SDLXliff files in OmegaT
State of the art

Thomas CORDONNIER

Table of Contents

Introduction

In OmegaT, the intermediate format in which the project memory is saved is not based on XLIFF, but on TMX, for historical reasons: first, because XLIFF was not fully specified when the project started. And second, because default OmegaT mode is to make association between source and target text: by default, the same phrase in the source, if it appears twice in the document, will have the same translation in both cases, until the translator explicitly requires for an "alternative" translation.

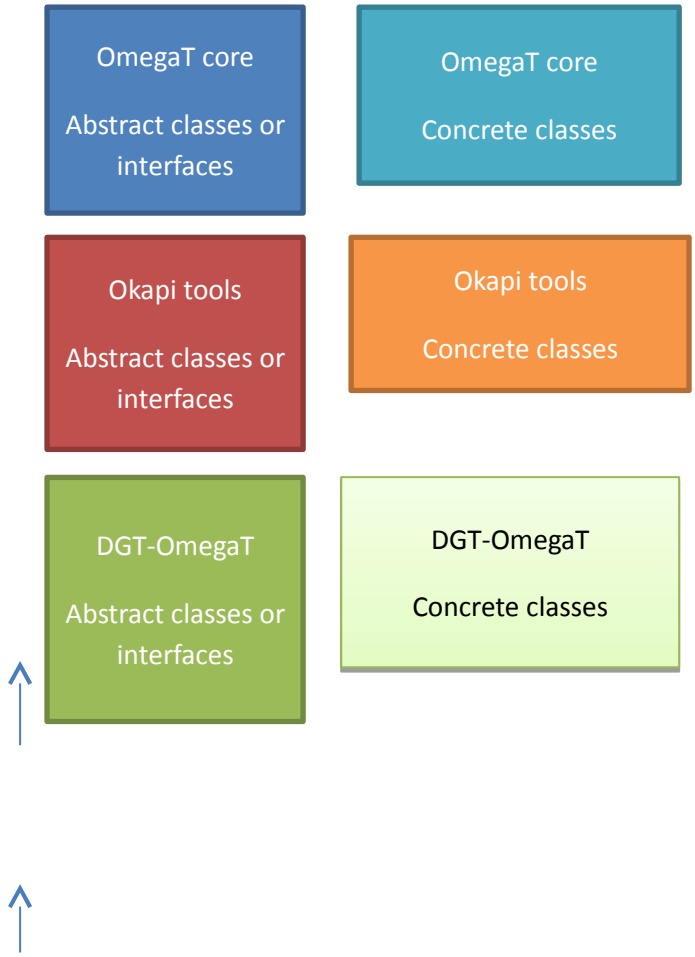
As a consequence, XLIFF format is only supported, inside OmegaT, via filters, i.e. as if it is the native format of the source and target documents.

Recently I was required to work on the possibility to use SDLXLIFF, the intermediate format of Trados Studio, based on XLIFF but with lot of additions, as a source/target format inside OmegaT. The reason is to make possible to collaborate between Trados and OmegaT users: it should be possible for an OmegaT user to make revision of a document translated using Trados, or the contrary, in both cases without losing any data, even concerning the formatting of the document. And OmegaT users also do not want to lose features they used with original DOCX documents just because they temporarily use SDLXLIFF as the native format. This document is a summary about improvements which have been made or are actually under development for that purpose.

Note: the start point of this document is the SDLXLIFF format, not OmegaT. Some of the problems described here, especially after page 18, may affect other CAT tools when they read SDLXLIFF files, as you can see in the small study of Heartsome, page 25. You can see this document as a study about compatibility of SDLXLIFF format with any CAT tool.

Colour scheme

The following colours will be used in the class hierarchy descriptions:

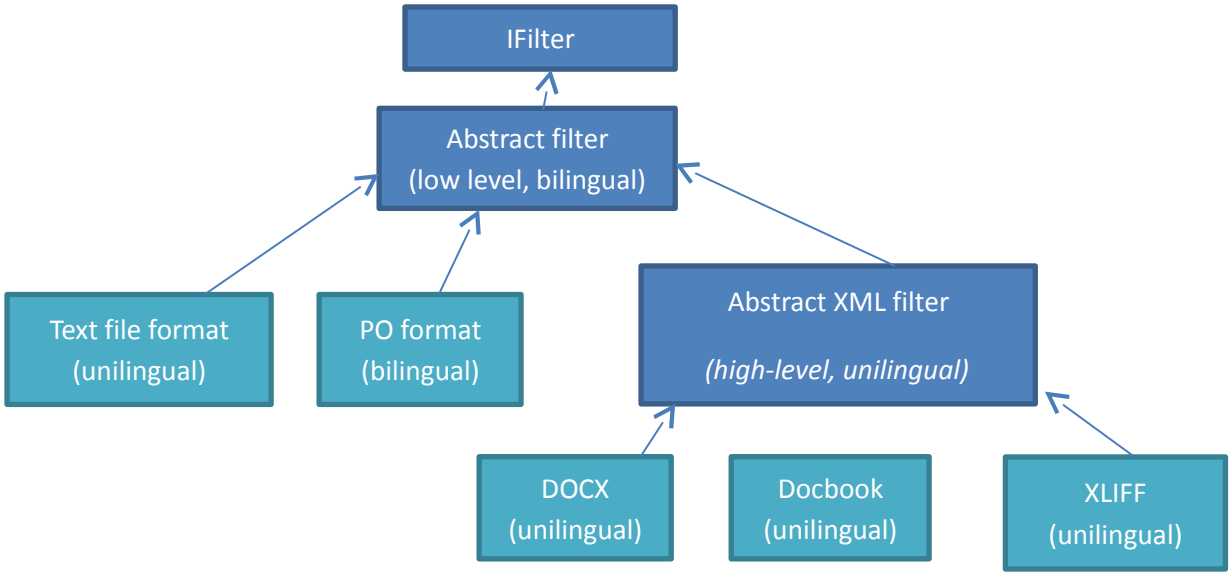


OmegaT filters for XLIFF

OmegaT filters architecture

Sorry for using programmer's terminology, but I do not see any other way to explain this.

OmegaT filters are classes implementing a common interface IFilter with the following hierarchy (only a few examples for illustration):



Existing filters for XLIFF format

Native filter

The core OmegaT application provides a filter based on the abstract XML intermediate. The intermediate class makes easier to write new filters with less code – you only have to declare which markups are used to begin or to end a segment, which attributes have to be considered as a segment to be translated (for example <a title> in XHTML) etc. But it has a big inconvenience: it has only been designed for **unilingual** document formats, such as Open XML (docx, Excel xlsx, powerpoint…), Open document format (libre office, odf), docbook, etc. It is totally impossible to implement with this intermediate class a filter which makes the distinction between source and target segments.

For that reason, the native XLIFF filter provided in the core is in reality unilingual: it will read and write only the contents of the <target> markup. Of course it means that it only works with a special kind of XLIFF files: these files have all segments translated with a translation identical to the source.

If you ask OmegaT team about this filter, they will say to you that such files are produced by the Okapi tools, a set of command-line batch tasks which can be integrated to OmegaT. The idea is the following:

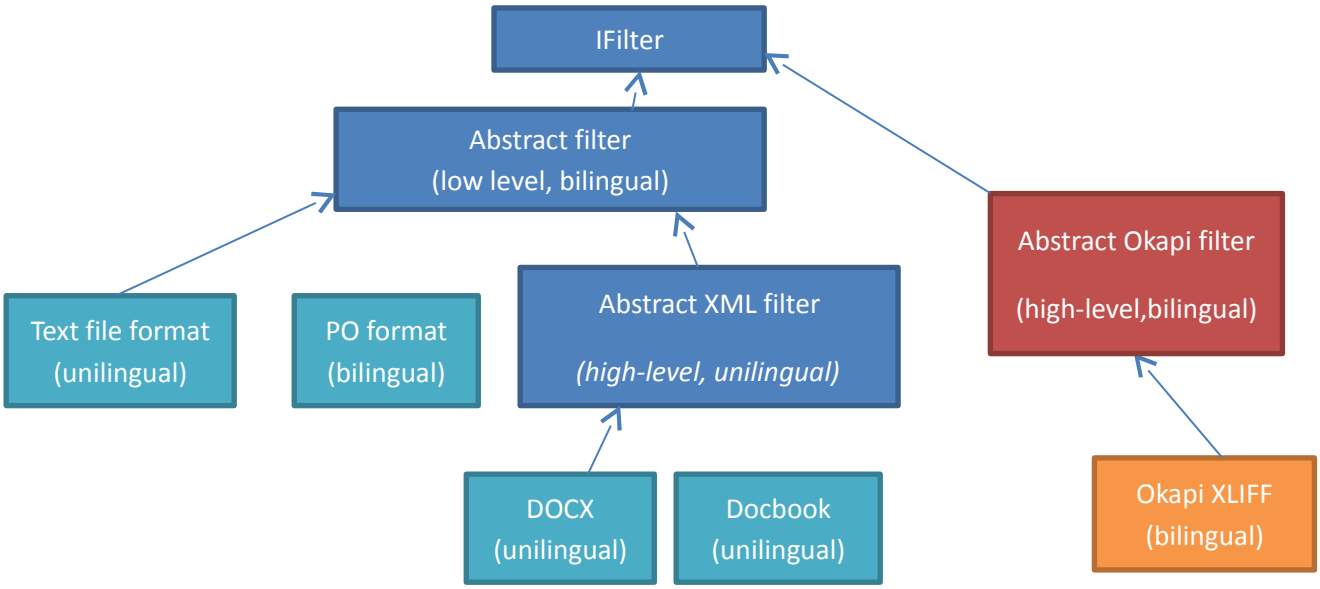
1. Using Okapi "tikal", you convert your native document, let's say a Word document, to XLIFF

2. You translate the document in OmegaT
3. Using Okapi "tikal" again, you merge the translation with the source document to produce the target DOCX document back.

And if you ask OmegaT team about how to translate other XLIFF files via OmegaT, they will probably tell you that they won't produce a dedicated filter because there is already another existing one, provided by the Okapi team.

Okapi filter

Yes, the team which produces the tools we just talk about also provides a plugin which adds inside OmegaT the support for a lot of file formats, including XLIFF 1 and experimental version for XLIFF 2. Contrairly to OmegaT's native filter, this one is really bilingual – it is NOT based on "abstract XML" filter, but on their own abstract class directly connected to IFilter:



This filter is significantly more powerful than the native one, but as the abstract part depends on Okapi's API, rather than OmegaT's one, modifying it is more difficult, because I would have to learn a lot about Okapi's API.

For users, it has a small disadvantage which will be discussed in the next chapter.

Importing SDLXLIFF in OmegaT

Using the native filter

Before the translation, SDLXLIFF files contain <target> markups but without translation in it, not even a copy of the source. This is conform to XLIFF standard, the fact that the native filter does not understand this is the fault of OmegaT, not of SDL.

It is easy to copy the sources to translations, either via Trados Studio itself, or via a script – about these markups, SDLXLIFF is conform to XLIFF standard. But later this is not possible to make the distinction between non-yet translated segments and already translated ones. And this solution is not usable for revision, because the source would not be visible at all. That's why this solution has not been retained.

Using the Okapi filter

The Okapi filter correctly reads SDLXLIFF documents, makes the distinction between source and target, does not consider as translated a segment where <target/> exists but is empty, and is even capable of loading the notes and make them visible for OmegaT.

But specifically when reading a SDLXLIFF, and not an XLIFF coming from another tool, it has a big inconvenience, which is, as a paradox, due to the fact that it better interprets the XLIFF standard than what SDL does.

When a document with formatting is converted to XLIFF, info about bold, italics or colors are not saved in the XLIFF : instead, they are converted to a small set of markups, mainly two : <g> for paired markups (with beginning and end), and <x/> for those which have position, attributes, but which do not surround a piece of text. The XLIFF specification only says that these markups must have an identifier (attribute "id") which is required, but gives no information about how this id looks like, whenever it is an integer or not, and so on. The specification, in particular, does not say that this identifier must be unique – but it does not say the contrary as well. And SDL took a very strange decision: they use an integer which is unique for each markup, increased from 0 until the end of the document.

Note about XLIFF 2. In the new specification, id must be unique *at unit level*, which seems logical. But once again, they do not see that it should not be the case *at file level*: even if we switch to XLIFF 2 in the future, I strongly suspect that they will continue to profit from this ambiguity.

When loading such a document, the Okapi filter generates for OmegaT the markups it has read: `<g id="234">` will be translated to `<g234>`, which, strictly speaking, is perfectly correct. The problem is that a unique identifier has absolutely no chance to make a 100% match with another segment – neither from the translation memories, in TMX format, which will probably have tags such as `<g0>`, `<g1>` or `<g2>`, nor inside the document itself: even the same phrase cannot match, because <g12>Hello</g12> cannot be considered as identical to <g23>Hello</g23> !!!

Displaying source or target file

The Trados interface makes possible, during the translation, to view in real time the target file. This means, practically, that it re-compiles the target file regularly.

In DGT-OmegaT we wanted to add the same feature (it does not exist in the standard OmegaT). When the source file is in Word format, here is what happens:

- "View source" menu uses Java's Desktop class to ask Windows for which tool must be used to open the file; Windows answers with WINWORD, so it opens the file inside Word.
  This method has an inconvenient: the file is opened read-write, and we have no real way to prevent modifications in the file (while Studio opens a special read-only window for the document). It may be possible to use WORDVIEW instead, but not using java.awt.Desktop, this solution would not work when OmegaT is used under Linux (on the contrary, the version which is published in http://185.13.37.79/ will automatically switch to LibreOffice in this case… but it is still read-write)
- "View target" menu first calls for the menu "Create current target document", then applies the same algorithm as the previous one, but applied to the just generated target file.

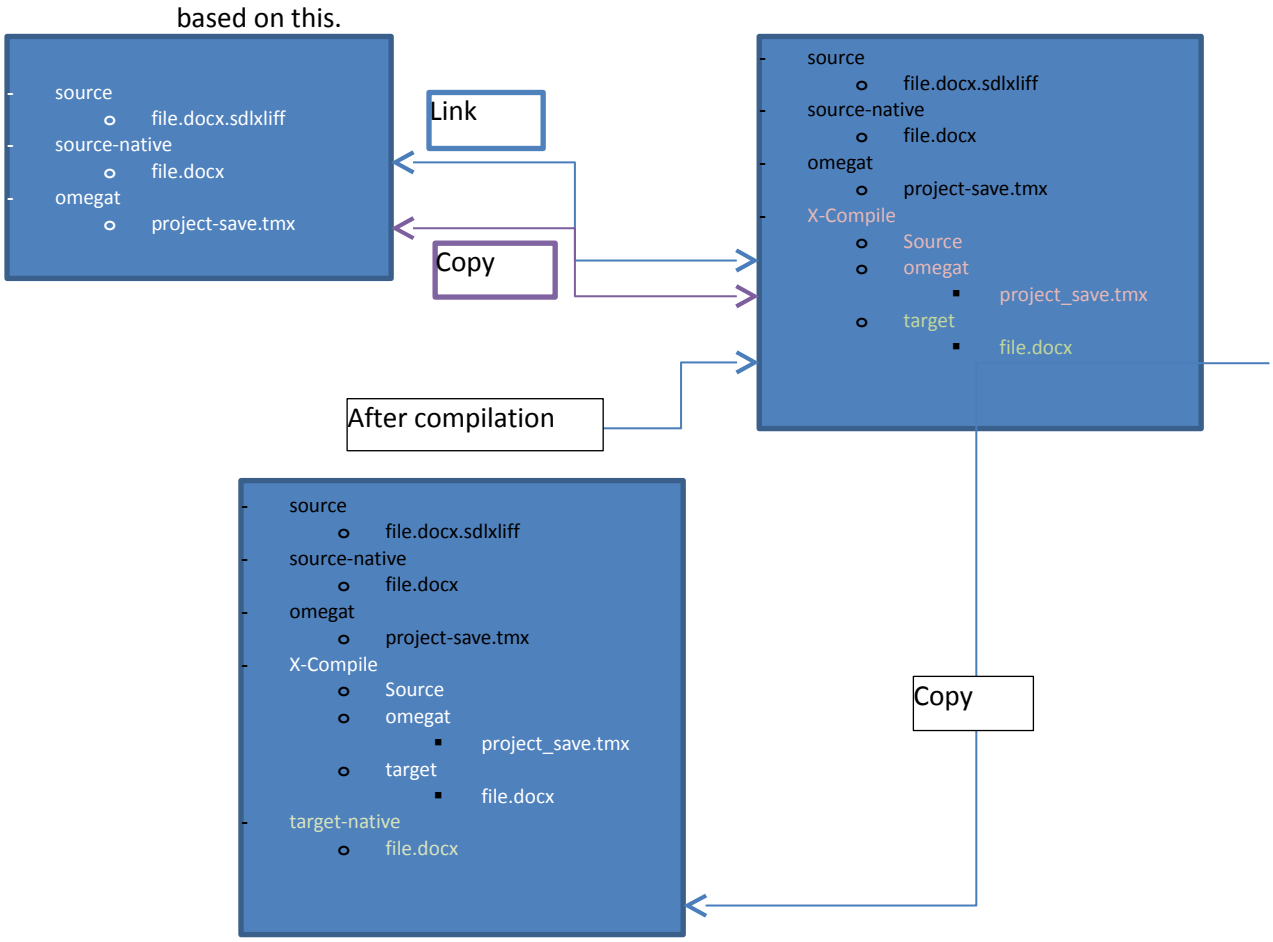
Now, if the document is in SDLXLIFF, what happens? If we apply the previous algorithm, Java's Desktop class will open the file with the application associated to this format, which is… Trados Studio. And this is not what we want. So, when I detect this case, I must change the algorithm:

- To "view source", I do the following:
  - Provide the original file in an alternative directory:
    - when a project has been generated by the Okapi tools, then «source» contains an XLIFF and «original» contains the Word document;
    - Look in directory «source-native» : we could have used "original" as well, but for SDLXLIFF files, the users did want to see "source" in the name, so I had to change it;
  - Look in `<file original='xxx'>` markup in the XLIFF code (this is a standard XLIFF markup, also supported by SDLXLIFF)
  - If the file is in SDLXLIFF, then it contains the original encapsulated as BASE64. Version 3 of DGT-OmegaT adds the possibility to extract it in «source-native», so that it can then switch to step 2.

- To "view target" it is more complicated. In any case, if the file contained in «source» is an XLIFF (or an SDLXLIFF), this means that it has been generated by another tool. Two techniques have been experimented:

1. *Cross-Compilation*

A new OmegaT project is generated, where "source" directory points to original project's "source-native" or "original", depending on which one does exist. Then we copy the project memory to this new project, and run OmegaT's compilation



Since the two projects contain the same segments, this should work. In practice this has two limitations:

- Segmentation in the XLIFF file is not the same as the one produced by OmegaT when reading the native file.
- Tags always differ between XLIFF filter and the native filter

Both problems have the same consequence: segments which differ will appear in the translated document in the native language, as if the user did not translate them. In its "auto-propagation" mechanism, OmegaT only has consideration for strictly identical segments.
On the other hand, the advantage of this technique, compared to the one which follows, is that you can run it without any extra tool, and in particular, you do not need Trados to do cross-compilation of an SDLXLIFF file. Also this is totally agnostic about the original tool: if it works for SDLXLIFF, it will work for MQXLIFF as well, without any need for re-configuration.

A partial solution for the segmentation problems is experimented in release 3.2 update 4: we consider that segmentation errors can be divided in 3 categories:

- Undercut : OmegaT produces less segments than Trados, 1 segment in the native file corresponds to 2 or more segments in XLIFF
- Overcut : OmegaT produces more segments than Trados, 1 segment in the XLIFF file corresponds to 2 or more segments in the native file when read by OmegaT

- Mixed : for example, 2 segments in XLIFF corresponds to 3 segments in native file, or the contrary

Considering the fact that the project_save.tmx is currently based on the XLIFF file, here is what we experiment now:

- For undercut: each segment from the native which is not yet translated file (read via the OmegaT filter for native format) is compared with all segments from project memory; if one segment in project memory corresponds to beginning of the current segment, we take the translation and we try to find the rest of the current segment in the project memory. Then we concatenate all found strings, leaving the last one in native language if it was not found.
  This algorithm has quadratic complexity $O(n^2)$, it can take some time for large files.
- For overcut: each segment from the project_save.tmx is submitted to the segmenter for source language; if it produces more than one segment, do the same for target language; then, if both give same number of subsegments, we make the hypothesis that they are correctly aligned, so we add the corresponding entries in the project_save.tmx
  This algorithm has linear complexity, it does not take lot of time.

For the moment we provide nothing for mixed situation, and there is still the problem with difference in tags, for which we may try, in the future, to use the same algorithm as in

2. *Call the original tool*

Omegat first generates the target (sdl)xliff file in `target` folder. Then the file is copied into a Trados project, and we call Trados to compile this project and copy the compiled file to `target-native`.

This technique solves the two previously mentioned problems: since the XLIFF file is read by the same tool which has been used to generate it before translation, the tool uses the same segmentation and tag system.

Trados's executable is not designed to be called on command line, or inside a script. For that reason, I needed to develop a small batch tool, which calls the Studio API to launch in command-line the task "Translate", which is nothing more than the equivalent of OmegaT's "Create current target document".

Actually I only implemented this for SDLXLIFF, so for Trados. Even if the exact location of the batch executable can be configured inside OmegaT, this does not mean that you can replace it by another batch tool without any change.

If I want to add the same feature for another CAT tool, it depends whenever its executable can accept parameters in command-line or not. But in any case, I have to hard-code the correspondence between OmegaT directories and the parameters to be passed to the native tool. So this cannot be totally generic.

But the thing which is particular to licensed tools like Studio is that this tool, to give the permission of using its API, requires that it is installed inside the directory of Studio itself… meaning, of course, that you must have an installation of the tool, with the correct license!

My developments: what can be done and what cannot

The OmegaT native filter loads SDLXLIFF with segment-based markups (the number in the markup is reset to 0 at each new segment), so it does not have the problem just mentioned before. But modifying it to be bilingual is strictly impossible: it would imply either to modify the abstract XML filter in order to make them possibly bilingual, or to rewrite the part of XLIFF filter linked to abstract XML.

Modifying the Okapi filter to use segment-scope numbers is probably possible, but it will be difficult: Okapi filters are a part of a bigger architecture which I do not master, and I am not sure whenever it is possible to modify only this filter without impacting other parts of the tools suite. We have sent to the Okapi team a ticket (https://bitbucket.org/okapiframework/omegat-plugin/issues/20/numerotation-of-tags-in-xliff-filter) to ask them for a study of the problem, but we did not receive a real answer, 5 months later.

Important note: despite all the mentioned inconvenient, the support of SDLXLIFF is correct: after a translation or a revision, if you "compile" the file in OmegaT and then import the generated target into Studio, it is perfectly recognized and you can even continue to work with Studio.

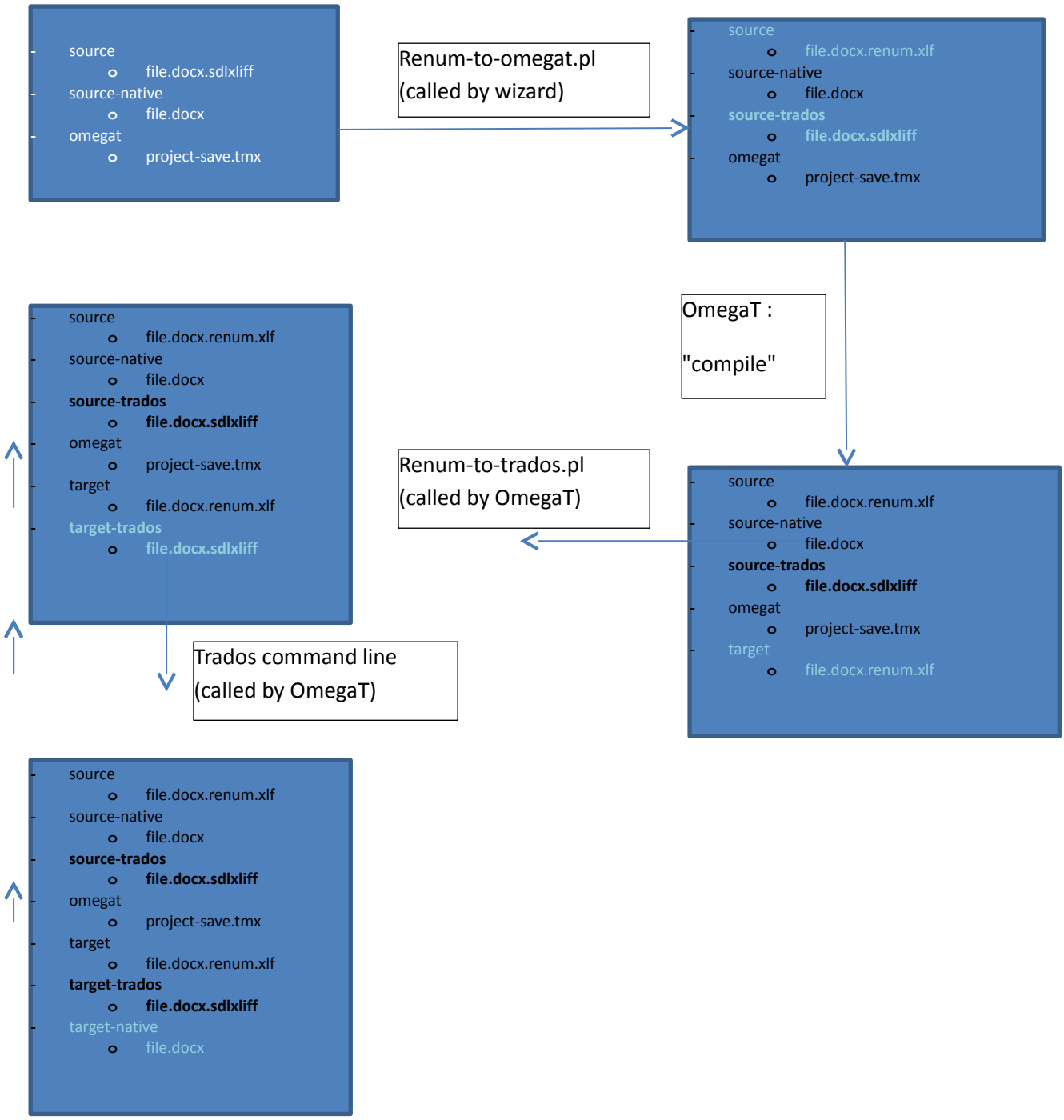
So I finally preferred to study two alternative solutions which I will detail now.

Renumerotation script

First, I must precise that contrarily to what lot of users of the Okapi filter think, even outside DGT (for example : the comment from Didier Briel here https://bitbucket.org/okapiframework/omegat-plugin/issues/11/implement-the-same-inline-code) the number generated by Okapi filter is not necessarily sequential: it is only identical to what is in the source file, so it will be sequential only if it is the case in the source file. I did a first test: doing a manual renumerotation in SDLXLIFF file, with segment-scope numbers, then I loaded it in OmegaT and saw exactly the numbers I had entered.

So, the idea is to couple OmegaT with a pair of Perl scripts, like this:

- Before import, a script transforms `.sdlxliff` file to `.renum.xlf`, which is identical except that all tag numbers are reset to 0 on each new segment
- After compilation of the target file, a second script does the conversion in the reverse direction.


source
file.docx.sdlxliff
source-native
file.docx
omegat
project-save.tmx
Renum-to-omegat.pl
(called by wizard)
source
file.docx.renum.xlf
source-native
file.docx
source-trados
file.docx.sdlxliff
omegat
project-save.tmx
OmegaT :
"compile"
source
file.docx.renum.xlf
source-native
file.docx
source-trados
file.docx.sdlxliff
omegat
project-save.tmx
target
file.docx.renum.xlf
target-trados
file.docx.sdlxliff
Renum-to-trados.pl
(called by OmegaT)
Trados command line
(called by OmegaT)
source
file.docx.renum.xlf
source-native
file.docx
source-trados
file.docx.sdlxliff
omegat
project-save.tmx
target
file.docx.renum.xlf
target-trados
file.docx.sdlxliff
target-native
file.docx

The "renumeroted" document would not load correctly in Studio, but I already did some tests and after the conversion in the reverse direction, the rebuilt SDLXLIFF file correctly runs inside Studio. One inconvenient of this solution is that even if I can integrate the reverse renumerotation during the "compilation" in OmegaT, the first step must occur before OmegaT can even see the document. As most users would not want to run manually a script in command line, this solution requires updating the workflow tool (actually the "wizard"). That's the reason why this solution is still not in test environment, even if it is ready from my side.

Special notes: supporting 2 or more file formats at same time

Displaying source and target language files

Trados enables to display in real time the source file in its native format (almost if this format is Word). In OmegaT we added this feature in DGT version, but it is not so powerful: it simply starts a new instance of the native editor, without even the possibility to set it as read-only.

With (sdl)xliff as a source document, this feature initially does not work anymore, because there is no native editor for XLIFF. But as I already wrote the code, this was easy to implement the possbility, when the document is in XLIFF-style format, to open the native file itself, in condition that it exists. The directories "native", in the previous schema, are here for this purpose. OmegaT is able to extract the file encapsulated in a sdlxliff, or to open the file declared in `<file>` markup instead. But to generate the target native file, then I need to call Studio: I had to write a special executable file for that – and SDL imposes to put it in Studio directory.

Intermediate file formats

Normally a CAT tool works with one source file format and one intermediate for its project memory. Each tool gives a list of source formats accepted via filters, and has one intermediate format (SDLXLIFF for Trados, TMX for OmegaT)

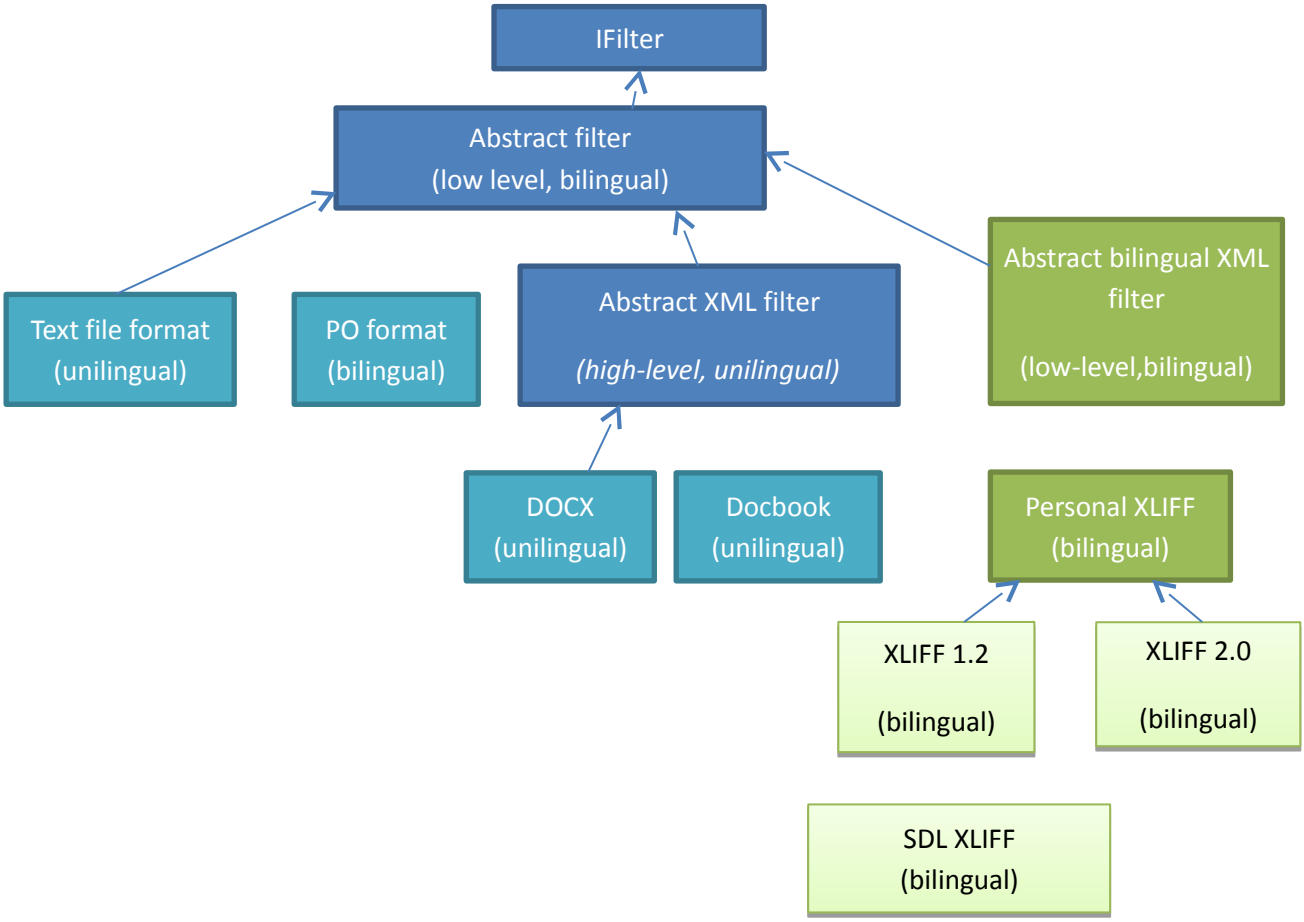
With my directories source-native and target-native, I implement a concept which no CAT tool used before, as far as I know : a second-level source file. And renumerotation introduces a 3rd level: renumeroted file vs source in SDLXLIFF.

If you ask another editor to support the standard XLIFF 1 or 2, they probably will do it via a filter at first level. But doing this does not give all the features of the native format: at minima you will lose the possibility to display source and target files (they are not Word anymore).

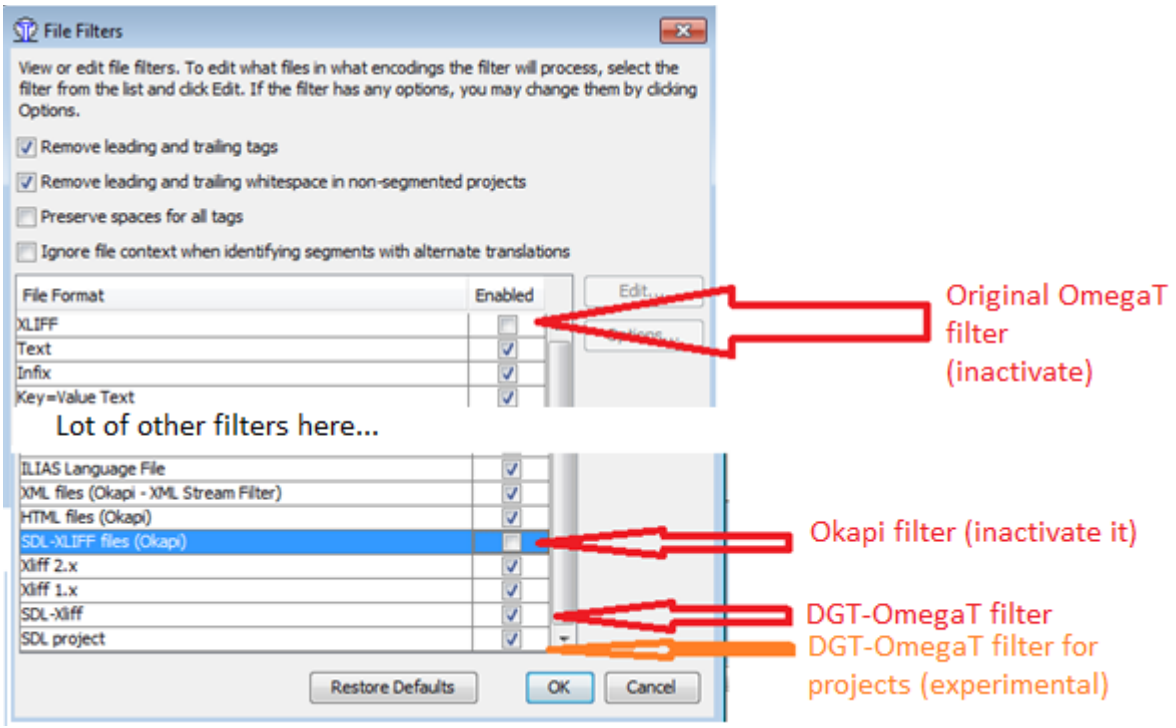
Studio API enables to create new file filters, and also to extend an existing filter for new features (we used it for Tag wiper, in the past). Unfortunately, even if this API is connected to "view source and target" features, there is no possibility to get access to the Word filter from the XLIFF one, so it will not be possible to connect the XLIFF filter with the viewer for the native format. Maybe this is a fact you could use when saying in call for tender that XLIFF support is required.

Writing a new filter

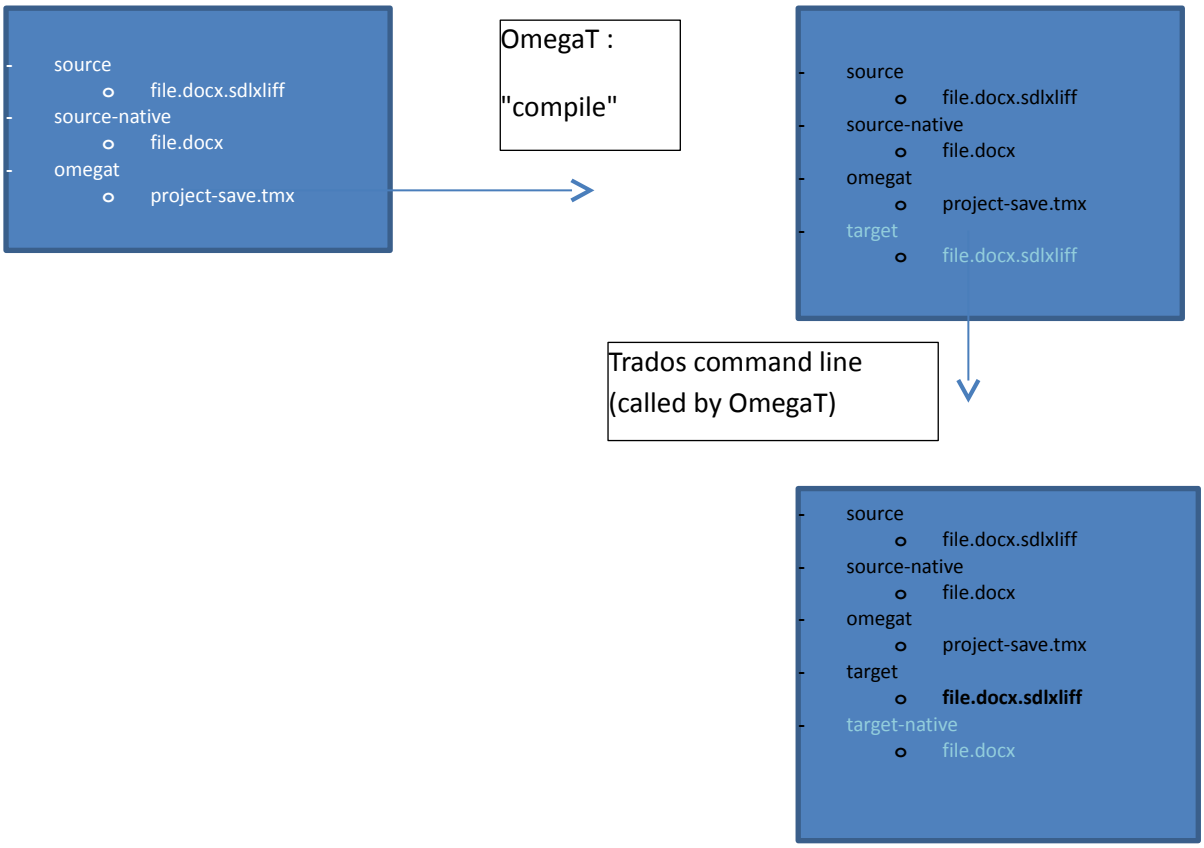
Another approach is to try to do the same as Okapi team did: writing a totally new filter, from scratch, based on the low-level abstract filter, rather than the XML one which is not bilingual:



This solution does not require intermediate "renumerotation" scripts, you simply select them as a replacement for "native XLIFF" or "Okapi XLIFF" in the list of available filters of OmegaT:



However, there is still the need for using the Trados command line:



The only inconvenience is that writing such a filter is, by comparison, a very long work, especially because I need to start from the same low-level basis (abstract filter) as for very different formats such as plain text or GetText's PO. And even my own abstract XML class remains low-level (you must use Java StAX events, rather than only declaring markups) so implementing another XML filter based on it would remain harder than when it is based on OmegaT's original one – but it is bilingual.

Actually I succeeded to write a filter which seems reasonably compliant with XLIFF standard. But I still have lot of work for supporting all SDLXLIFF-specific markups – if only I can understand each of them. In any case this solution will be better than the previous one, but only once it is finished!

Other problems with SDLXLIFF

Initially I tested the two previous approaches only for one problem – the numerotation. But in the meantime we discovered other problems with SDLXLIFF and I had a look in both approaches to see what can be done.

This chapter describes what we already discovered, but remains open in case we find other problems later.

Support for notes

Standard XLIFF 1

The Okapi filter is already capable of displaying, in OmegaT's "comment" window, the notes which are inserted in an XLIFF file with the standard markup <note>. I already added this possibility in my own filter.

But first, we can notice that this does not work in the reverse direction: during compilation, notes in the document are kept as is, meaning that those who already existed in the source will be kept in the target, but that those which have been added by the translator will not be added. Here we can say that this is OmegaT's fault: the abstract filter (low-level) does not even give the possibility to insert new notes in a target file, even if the target format does support them.

I added the possibility to read notes in the source in my own filter, and then, for the moment experimentally I succeeded to implement the possibility to add notes in target files at abstract filter level. With my own XLIFF filter, this seems to work. I hope it has no bad consequences for other formats!

SDLXLIFF

Unfortunately, SDLXLIFF does not use the standard <note> markup, provided by the XLIFF standard. When a user of Trados Studio adds a "comment" in the document, a markup is inserted in the source or target segment itself to indicate the scope, and then the comment itself is in a dedicated location… in the beginning of the file!

In one sense, I understand SDL's position: the 3 options to indicate the scope of a note proposed by the XLIFF standard 1.2 – source, target and undefined – are clearly insufficient. But the solution they propose is not correct. First it is not conform to XLIFF standard – the standard allows extensions only at the *end* of a section, not at the beginning! Second, it obliges all algorithms to parse the document twice – one for the notes, one for the text. Clearly there is a reason for using a non-standard behaviour but not for having the comments grouped in a section separated from the text - unless the reason is to make the things harder for other CAT tools. Even if attribute note/@annotates cannot be extended, they should have added a own-namespace attribute for this, it would be more compliant with the standard.

Modifying one of the existing filters for that purpose would be difficult.

For the renumerotation, I already could add in the Trados→OmegaT script a conversion between SDL "comments" to XLIFF "notes". Then, the Okapi filter is capable to read these notes (as it would do for standard XLIFF) and display them in the “Comments” window. In the reverse direction, you need to add the `project_save.tmx` (project memory) as parameter to the script: then all notes which are present in this project memory will be converted into SDLXLIFF comments.

For the new filter, the situation is the following: reading from the source is easy – I only have to rebuild the association between notes and segments. On the other direction, even if I can generate the corresponding markups, the fact that they have to be in the beginning of the document is a source of complication, since OmegaT filters are normally not designed to work in two passes. Actually it works, except that I cannot remove comments which were present in the source but are not valid anymore after the translation: they are present in the file, even if Trados ha no way to show them anymore.

In any case, OmegaT is not as powerful as Trados on this point: it provides no way to specify the scope of a note, while Trados provides a probably too precise one – almost from the point of view of XLIFF, which did not specify a way to attach a note to a single word inside the segment. If I finally succeed to implement something, it will probably be to attach OmegaT notes to the full segment, not less.

Standard XLIFF 2

XLIFF 2 has an even worse support for notes in segments than XLIFF 1 : notes can only refer to a unit (the full paragraph), not even to a segment! For that reason, the Okapi filter decided to remove them completely.

They also added a new type for <mrk> : <mrk type="comment" comment="xxx">. This is probably the best solution, except that having a long comment in an attribute, rather than a reference to a text which comes before or after – but in the current unit, not in the beginning of the document as SDL did!

Support for status

Trados, contrarily to OmegaT, makes a distinction between draft translation and validated one (appears with a pen in the editor). Unfortunately, as usual they use for that a custom set of markups. This time they have no excuse, because the corresponding attribute does exist in XLIFF (target/@state) and it explicitely accept user-defined values (http://docs.oasis-open.org/xliff/v1.2/os/xliff-core.html#state).

In any case, both native and Okapi filters compile the document without changing the state attribute, even if the text is translated in the target while it was not in the source.

Now in renumerotation it is possible to add the corresponding markup (the one Trados understands, not the standard one) in target file. With the new filter it is done during compilation.

Unsegmented SDLXLIFF files

Let's remember that the main reason why OmegaT users would have to use SDLXLIFF files is to collaborate with users of Trados Studio. And they want to see the same things – in particular the same list of segments.

The XLIFF format is supposed to be already segmented, by definition. But in fact, the standard has the support for two levels of segmentation: paragraph and phrase. This is true in XLIFF 1 as well as XLIFF 2, even if I admit that the way it works with XLIFF 2 is better.

Let's take an example in XLIFF 1.2, also valid for SDLXLIFF:

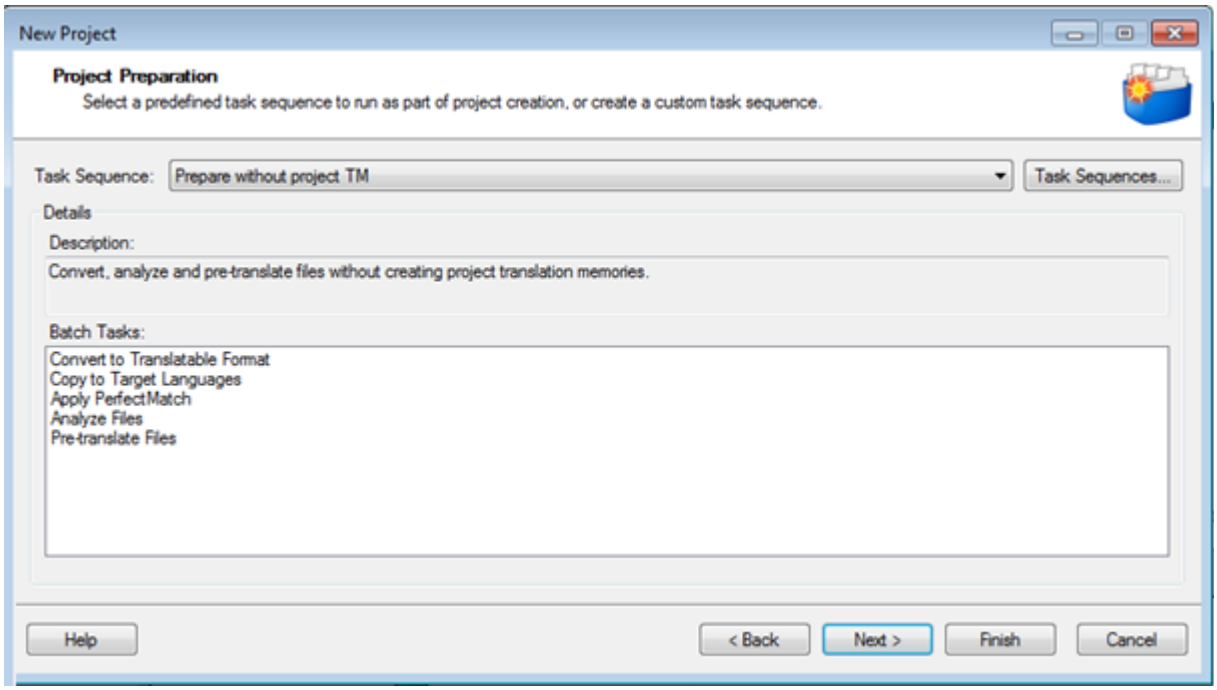
```
<trans-unit>
    <source>This is a paragraph. It contains two phrases.</source>
    <seg-source>
      <mrk type="seg" mid="1">This is a paragraph.</mrk>
      <mrk type="seg" mid="2">It contains two phrases.</mrk>
    </seg-source>
    <target>
      <mrk type="seg" mid="1">Ceci est un paragraphe.</mrk>
      <mrk type="seg" mid="2">Il contient deux phrases.</mrk>
    </target>
</trans-unit>
```

Such a document will work perfectly in OmegaT : when a trans-unit contains seg-source, both native and Okapi filters can use it and generate two segments, in this example, instead of one.

Unfortunately, during the "project creation" inside Trados Studio, creation of <seg-source> is not implicit: if you look inside a Trados project, and more precisely in the directory corresponding to source language, you may notice that there is not only the original file, but also a SDLXLIFF file which seems useless. In reality this file is correct but does not contain the <seg-source>. Trados users, on the internet, name these files "unsegmented SDLXLIFF": they are not translated, which is logical, but they contain one <trans-unit> by paragraph, without segmentation as <seg-source> entries.

Personally I do not see any interest for such a file. People on the internet say that they can be used as a basis for multi-target translation – you give the same English source to French and German language translator, for example. But for me it does not justify why the file is not segmented: even if Trados Studio enables to use different segmentation rules for different translators, and even if it allows to the user to cut a segment or to merge two into one, the role of XLIFF is to unify segmentation, not the contrary!!!

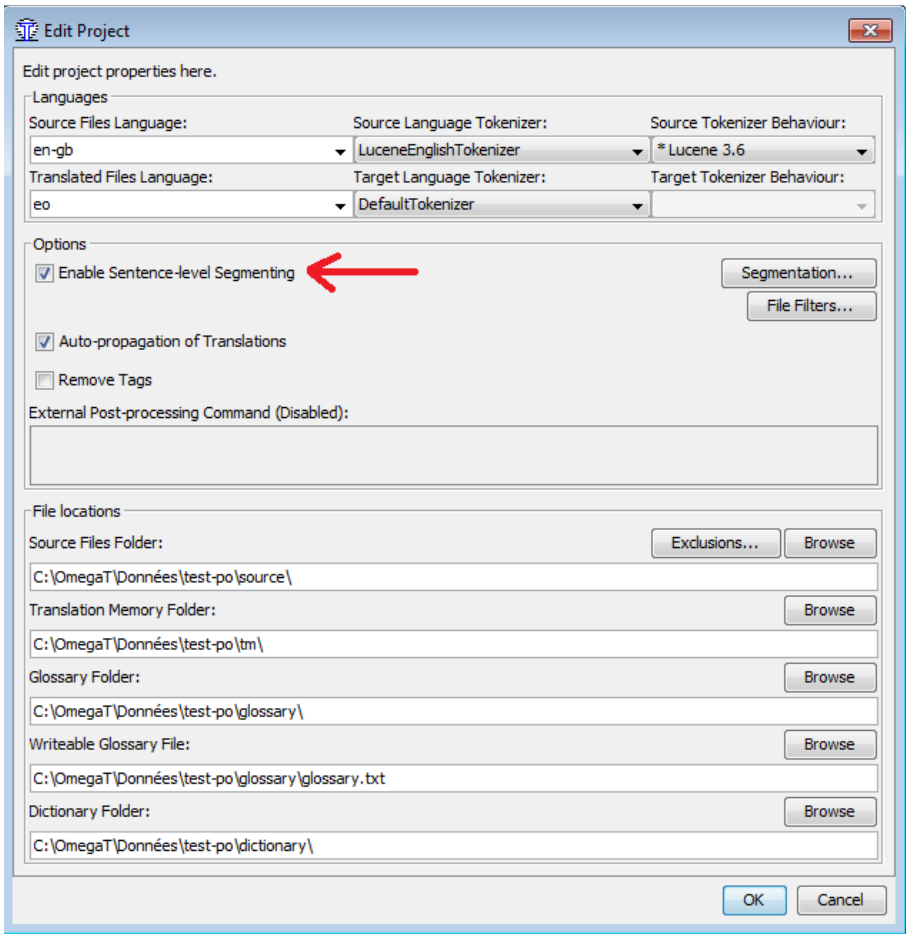
If you open the Trados GUI, create a project and import a DOCX file into it, after the project creation you will have the unsegmented file in the source-language directory, and normally, a correctly segmented file in the target. But that is true if, and only if, you have something like "Pre-translate files" in the list of batch tasks to be executed during preparation:



If it is not the case, the file in the target directory will be identical to the source. It seems also that the CAT client, which is actually the most used way to create projects inside DGT, does not execute the pre-translate task. For Studio itself it changes nothing: when the project is loaded it automatically re-applies its own segmentation rules (The only case where it could cause problems is if you take this SDLXLIFF and try to import it in another Studio project: unless this project has exactly the same segmentation rules, the aspect inside Studio will differ). But when you open such a file in OmegaT, it depends. OmegaT is perfectly capable of re-segmentation – but we are absolutely not sure to have the same rules: first, because Studio's "translation rules" format is not fully documented, so there can be some non-standard behaviour which are not reproducible using another tool; and second, because even in our internal teams we are not always informed by Studio team when they change the rules!

For that reason, we actually need to detect such files, and open them inside Studio to obtain a segmented file. For the future, I already developed a batch task runner, i.e. a command-line tool which can be called via an external interface such as the wizard. But all of this works only if the tool is installed, and if we have Studio in the PC: this solution is totally not applicable for freelance users, unless you want to force them to buy a Studio license.

Warning: by default, OmegaT considers that documents are not segmented, or if they are, it is at paragraph level. This is logical if we consider that most file formats are not segmented, but XLIFF is one of the rare exceptions. You must inactivate segmentation when you create a project in XLIFF format:



If you don't, then OmegaT will apply its own segmentation rules **in addition to** what is already registered: it can happen that you obtain more segments in OmegaT than in Studio. This does not have consequences about file parsing – during target generation, it will rebuild original segmentation and Studio will not see the difference. You can also decide to use unsegmented file in OmegaT and let it apply its own rules. But in both cases, having different segmentation in OmegaT means that you lose one of the advantages of using SDLXLIFF instead or native DOCX: interoperability. If segments are different, it can also prevent from finding them correctly in the translation memory.

Non-translatable segments

I do not know the exact reason, but SDLXLIFF files contain a lot of segments which should not be translated: they do not appear when you load the document in Trados editor.

XLIFF has support for this feature via the attribute translate='no'. Even if Okapi filter ignores it, it is easy to implement it, either in the renumerotation (we can remove them completely, and retrieve them back from the source SDLXLIFF during reverse conversion). And of course it can be implemented in the new filter as well.

Unfortunately, once again things are not so simple. Some segments do not appear in the editor even if the corresponding attribute is not present. And for the moment I did not succeed to find a general

rule to find which segments are concerned. I am pretty sure that these are segments containing only tags, no text – but I also saw examples of segments which did satisfy this criteria but appeared in Trados editor, so the condition seems necessaire, but not sufficient. For that reason I still do not have a general solution for this problem: even if I know how to remove segments, in both methods, the problem is that I do not know exactly, which segments are concerned.

In any case, the only consequence is that you will see in OmegaT segments which you won't see in Studio. But if you translate them as identical to the source, the target SDLXLIFF will be perfectly readable by Studio: once again, this is only a factor of different behaviour, not a factor of incompatibility.

Extra formatting

Normally, users want to have the same formattings (i.e. bolds, italics, underline… in the same position) in the source and in the translation. But our users found situations where they want to add an italics in the target in a location where it was not formatted in the source: for example, to give emphasis to the fact that this is a word in the original language, which should not be translated.

XLIFF provides markups to do this:

```
<g id='xxx' ctype='bold'>text in bold</g>
```

But as usual, SDL decided not to follow the standard. First, they put the formatting in the `id` attribute. Even if the standard says that `id` is required, here SDL is a little bit hypocrite : for all other tags they use a globally unique identifier, which causes us many problems, but for this case, they use a non-unique one (if you add more than one italics in the file, all will use `id="italic"`). And as if it were not already so complicated, if you add, via another tool, a <g id="italic"> in a file which did not contain it before, then Trados will say that this tag does not exist, because it is not declared in its proprietary `<tag-defs>` markup!

So, finally the only solution is to add our own declarations in `<tag-defs>`, then we are allowed to use them in the segments. For the moment it seems to work, but we did not test a lot of cases…

Significant tailing spaces and tabulations

This case occurs probably rarely, but it is particularly hard to solve.

Taken from the XML standard description (https://www.w3.org/TR/xml/#sec-white-space):

"In editing XML documents, it is often convenient to use "white space" (spaces, tabs, and blank lines) to set apart the markup for greater readability. **Such white space is typically not intended for inclusion in the delivered version of the document.** On the other hand, "significant" white space that should be preserved in the delivered version is common, for example in poetry and source code."

In practice, we have found some documents where leading spaces or tabulations were found in the beginning or in the end of a segment. When it occurs, some XML parsers will ignore them, or

eventually collapse multiple spaces into only one, because they consider them as separators or indentation rather than really significant part of the text. But then, when you open in Studio the target SDLXLIFF file generated by another CAT tool, these spaces which really appeared in the source SDLXLIFF document do also disappear when reading the target document! And even worse, if from Studio you then generate the target native file, these spaces also disappear from the target native file!

In OmegaT, and probably in other CAT tools as well, it would be very difficult to modify the filters to preserve spaces: the error is not at XLIFF level, but at XML level : strictly speaking, these documents are not anymore valid XML ones, because they use spaces which should be preserved but without telling us (with `xml:space="preserve"`) that it is the case! And as a result, the same application (OmegaT) has different behaviour, depending on which XML parser is used at Java level!

For the moment, the only solution seems to modify the source files before giving them to our favorite CAT tool : either adding `xml:space="preserve"`, or englobing tabs inside CDATA – because in CDATA, the XML specification says that spaces ARE preserved. Writing a new filter we wanted to avoid the requirement to call an extra tool before OmegaT… but now it sounds impossible.

Tests: using XLIFF with other CAT tools

From the previous chapters I could conclude that OmegaT's support for XLIFF, and then for SDLXLIFF, is initially poor, but that due to the fact that it is open source, it is relatively easy to extend it to match our needs.

One could be surprised by the fact that XLIFF is not the native format of OmegaT. As I said in the introduction, this can be explained by the fact that OmegaT project started long time before the first specification of XLIFF was ready. But before asking us to modify OmegaT to use XLIFF rather than TMX, let's ask us the right question: ***do other CAT tools really make use of XLIFF, as it should be?***

Let's consider separately the CAT tools which are directly integrated in an editor, such as old Trados (integrated in Word), Anaphraseus (for OpenOffice) or Wordfast (also for Word). These tools cannot use an intermediate format at all. On the other hand, we cannot consider that they are really without intermediate format: if you open a file partially translated with the editor in a computer where the CAT tool is not installed, you will see a lot of strange markups which should be normally removed by the macros: this means that these tools *do* have an intermediate format, which is a variant of the editor's format, only partially compatible with it. But in any case, this cannot be XLIFF.

Another special case to be considered is the use of web CAT tools. These tools usually save their memory in a database, not in a file, because this is the best way to manage the fact that two people can modify the translation at same time. So, as for the previous case, these tools do not really need an intermediate format like XLIFF. But they should be able to export the memory in XLIFF format, and eventually to import it. However, as said in page 9, if XLIFF is considered as any input format, this is not sure that all features which are possible with the real native format will be available if XLIFF is treated exactly as if it were the native format.

So, you can understand that using XLIFF as "file filter" is not the same as using it as an intermediate format. Now we can go back to the initial question: ***do other CAT tools really use XLIFF as intermediate format?***

In Trados, the answer is clearly no. Not only because SDLXLIFF adds extensions to the XLIFF format: this is explicitly allowed by the specification. Real interoperability would mean that if I have an XLIFF file provided by another CAT tool, Studio should treat it exactly as if it was an SDLXLIFF, except for the features which are in the extensions, of course. This is not the case. If I replace, inside a project, the SDLXLIFF file by an XLIFF provided by another tool, based on the same native source file, this does not work: even if I rename it to sdlxliff, Studio clearly says that the format is not valid.

Now, if you ask SDL about XLIFF support, they will probably answer that there is a file filter for XLIFF 1, and even, since Studio 2017, for XLIFF 2.0 as well. This means that you use them as native format, not as intermediate, and as I said in the grey zone in page 9, features which depend from the native format, such as the synchronized source and target display, will at minima work differently, and most

probably, not work at all. In OmegaT, I could correct this because I have the source, but with Studio, even with a plugin, I am not sure that it is possible.

Before to blame Trados for this, let's have a look about another CAT tool which has a very similar user interface, even if internally it works very different way: Heartsome.

For those who don't know it, Heartsome was a commercial CAT tool which has finally be published as open source when the company bankrupted: https://github.com/heartsome/translationstudio8. Unfortunately, no community was created based on this tool, so if we want to use it, we have no choice but to use it as is, or to create a fork.

Like Trados, Heartsome uses its own variant of XLIFF, named HSXLIFF. But contrarily to SDLXLIFF, this format has really a very small set of added markups, so we could reasonably suppose that it could be compatible with XLIFF files coming from other CAT tools.

So, I tried to convert a file to XLIFF using the Okapi tools, then simply renamed it to HSXLIFF and replaced the file produced by Heartsome itself. And result: it works! We simply notice differences in segmentation, but except of that, the file is perfectly editable:

No.	eo
1	fr-BE
2	DGT/Omega-T for Omega-T users
3	Thomas CORDONNIER
4	Table of Contents
5	Introduction 2
6	Toolbar 3
7	Coulour codes and additions in editor 6
8	Auto-inserted text in the editor 6
9	Automatic machine translator using TMX file 7
10	Note about other machine translators 7
11	Search screens 8
12	Search in directory 8
13	Expression and word mode 8
14	Matches display template 9
15	Search in project 10
16	Text to search 10
17	Search scope 11
18	Search and replace 11
19	Pre-translate 12
20	Glossaries 14

HSXLIFF file produced by Heartsome

No.	en-GB
1	14.0000
2	European Commission
3	0
4	false
5	false
6	false
7	false
8	DGT/Omega-T for Omega-T users
9	Thomas CORDONNIER
10	Table of Contents
11	Introduction 2
12	Toolbar 3
13	Coulour codes and additions in editor 6
14	Auto-inserted text in the editor 6
15	Automatic machine translator using TMX file 7
16	Note about other machine translators 7
17	Search screens 8
18	Search in directory 8
19	Expression and word mode 8
20	Matches display template 9

File produced by Okapi tools

Unfortunately, then we lose the capacity of **re-generation of the target file**, and as a consequence, the "**preview**" feature. But this does not mean that we cannot use this solution: after translation, the file is saved correctly and can be used again with the tool which has been used to generate the initial version (Okapi tools in our sample). Almost we can conclude that Heartsome can be used in combination with other tools.

Actually, the process is not perfect: yes, you can call the original tool to regenerate the native file (word), but you have to do it outside Heartsome – either manually, or integrated in a workflow tool such as the wizard. This was also the case for OmegaT until I implemented the multi-level source files, and it is certainly possible to do the same for any open-source tool such as Heartsome. But here we can notice a strange lack in the XLIFF specification: you can specify a name for the native file,

eventually with a path, but there is nothing, no markup or attribute, to specify which tool was used to generate the XLIFF file. Of course you can detect markups which only exist in tool-specific XLIFF files, but some tools can use no specific markup (like Okapi tools), and even if it was the case, you would have to implement in your tool a class hierarchy where all concurrent tool has its own compatibility implementation! So, even if it is possible to implement multi-level source files, there is no way, using only XLIFF standard, to implement a chaining mechanism which is totally tool-independent, just because you cannot guess which tool to call in which order. Without such a mechanism, can we really say that XLIFF is a format for interoperability?

In any case, technically, I do not see why Trados completely refuses to open a project where the intermediate file has been produced by another CAT tool. This is really a potential problem of interoperability.

If really it is not possible to accept external XLIFF as an intermediate, almost the plugin architecture, which already makes possible to extend an existing file format with additional features, should enable connection with features like file display, in case the format we are extending is an intermediate format, rather than source one. Actually it seems that it enables to create a component for preview, but the problem is that the API does not give access to already existing ones: so, even if it is possible to write a XLIFF subfilter which calls the original tool to regenerate the file, this is not possible to ask Trados to open the generated file for preview, even if it is in a format like DOCX, for which it already has a preview component.

What about segmentation?

In previous chapters I regularly spoke about segmentation which differs from one tool to another. You probably know that there is a format named SRX (Segmentation Rules Exchange) which is supposed to be a way to share segmentation info between CAT tools, exactly as XLIFF is supposed to be the way to share translation info.

Unfortunately, SRX has a lot of limitations. Being a stateless algorithm, it makes some rules impossible to express (for example, the Euramis rule saying that we should never cut inside a parenthesis or quotation marks).

SRX specification has the advantage to be very short, but the inconvenience that it does not clearly define a lot of things: for example, it enables to specify whenever we should segment inside subflows or not, but it does not enable to define what a subflow is: in the example of Euramis, should parenthesis be considered as a subflow or not? Finally, in practice, I have found absolutely no SRX engine which had an implementation of subflows, making even more difficult to understand this definition.

Strictly applied, an SRX document would leave a lot of trailing spaces: as we generally need to put spaces after a dot, and as the cut occurs generally just after the dot and before the space, this means that except for the first segment, all following segments will begin with a space! When you try with a strict SRX engine, such as Ratel from Okapi tools, this is exactly what happens: you really see space in the beginning of each segment! On the contrary, most tools decide to remove this useless space, but it is done after the segmentation. OmegaT does it, I saw it in the code. Trados Studio seems to do this, but because I don't have access to the code, I have no way to see if it does it always (it should not be the case for languages like Chinese, but how to prove it except testing case by case?). The fact that this is done outside the SRX algorithm seems potentially problematic, because there could be bugs, or simply situations where it should not be done but the programmer did not know that. The possibility to explicitly request for space removal in the "rules file" should be very useful: it should ensure that all tools respect the same rule, almost if they want to be conform to the specification.

For all these reasons, it seems impossible to create a file which specifies segmentation rules in a format understandable by any tool, even with a plugin. There will always be differences, and without documentation by the tool owner, no possibility to anticipate all of them.

Conclusion

Interoperability between various CAT tools is not really ensured by SRX and XLIFF formats, but it is not really the fault of the tools: the specifications are sometimes not clear and commercial tools took profit from any ambiguity as a pretext to select the solution which would be the more difficult for concurrent to implement.

SRX and XLIFF were specified by an "association" (the term "consortium would be more appropriate) of CAT tools manufacturers, named LISA, which finally was closed when commercial manufacturers realized that their interest was to avoid any normalization, to limit interaction with free alternatives like OmegaT. Now all former LISA standards are maintained by various standardization organizations (mostly OASIS, but not only) but these organizations are not specialized about translation, and they seem mostly to keep the standards as is, without making improvements - probably not because they do not want, but most probably because nobody asks for it: if nobody wakes up, the situation is not ready to change and using various CAT tools in the same organization will progressively become a nightmare.