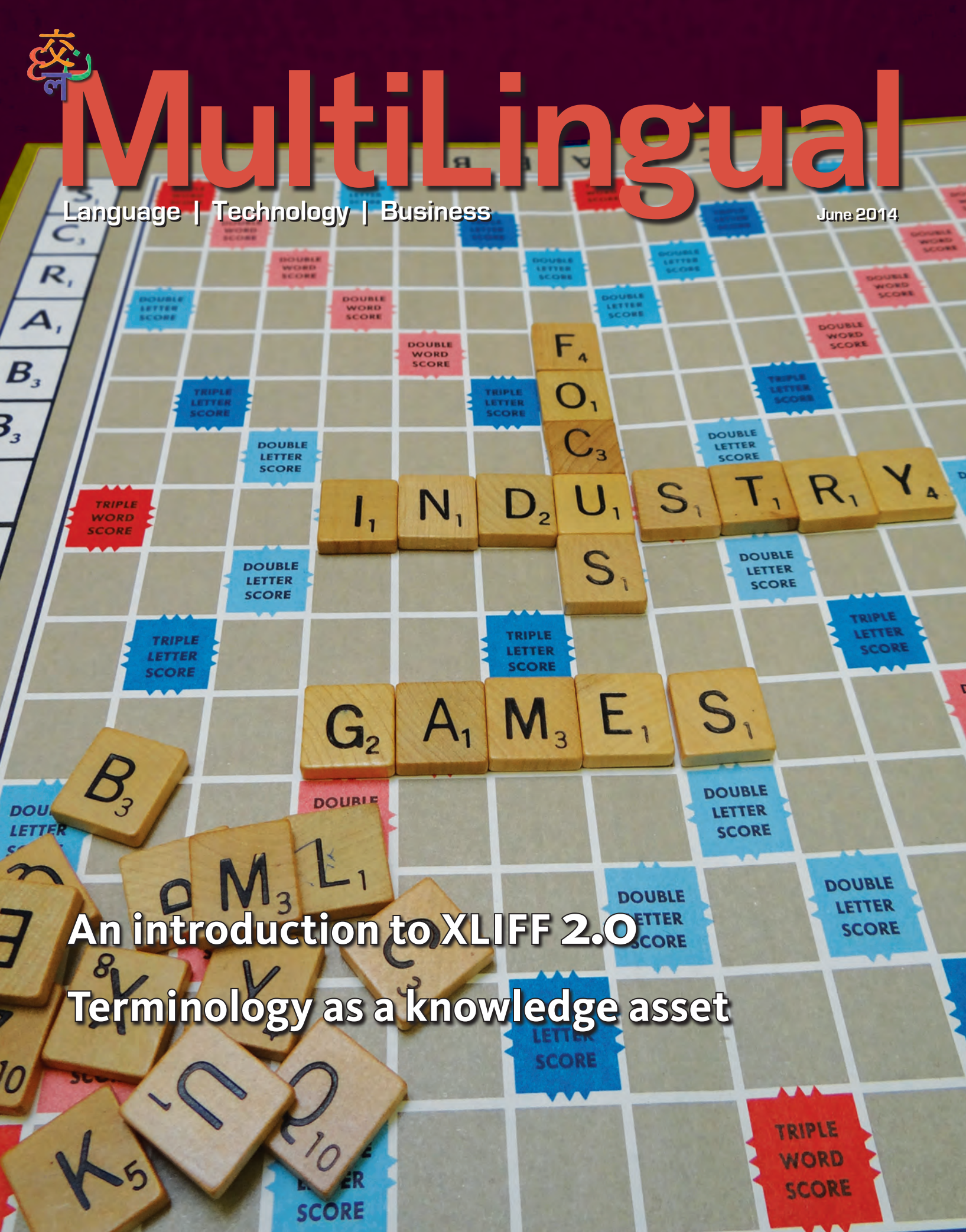
MultiLingual
Language | Technology | Business
June 2014
An introduction to XLIFF 2.0
Terminology as a knowledge asset

An introduction to XLIFF 2.0

Yves Savourel

The OASIS XLIFF technical committee published the XLIFF 1.2 standard in 2008. Since then, many translation tools as well as other types of applications have implemented support for the format. The technical committee started work on XLIFF 2.0 the following year by gathering requirements and feedback on 1.2. By the time you see this article, 2.0 should be an OASIS standard or very close to that status.

Yes, it did take a long time to get there. But developing a standard is a lot of work, and when it is done mostly without the benefit of sponsorship funds, it also requires a lot of time.

What is new with 2.0? To start with, the new version is not backward compatible with 1.2. This allows for a new structure with a different representation of the segmentation, as well as an important feature often requested: modularity. The specification splits the format into a base namespace called the Core that all implementations must support and several specialized optional modules. This separation ensures the stability of the format while providing the possibility of future enhancements.

Structure

The Core includes the basic elements needed to store extracted content, add its translation, and merge it back into the original format. It also provides structural information.

As shown in Figure 1, the root `<xliff>` contains one file or more in the same language pair. Each file contains at least one unit or group. A unit corresponds to an extracted "paragraph" and contains at least one segment. A segment holds a source and an optional target, respectively containing the original and the translated content. An optional skeleton can be set for each file and holds (or points to) the data needed to reconstruct the original document. You can use `<group>` to represent the hierarchy of the document. Any sub-flow

```
<xliff xmlns="urn:oasis:names:tc:xliff:document:2.0"
 version="2.0" srcLang="en" trgLang="fr">
 <file id="f1">
  <skeleton href="myFile.skl"/>
  <unit id="u1">
   <segment state="initial">
    <source>Start here!</source>
   </segment>
  </unit>
  <unit id="u2">
   <segment state="initial">
   <source>Click the <ph id="1" type="image"
    subFlows="u1"/> button to start. </source>
   </segment>
   <segment state="translated">
    <source>You can also press the Enter key.</source>
    <target>Vous pouvez aussi appuyez la touche Entrée.</target>
   </segment>
  </unit>
 </file>
</xliff>
```

Figure 1: A simple XLIFF 2.0 document.

Yves Savourel works with ENLASO Corporation in Boulder, Colorado. He has been in the localization industry for more than 20 years, and has been involved in the creation of localization standards such as TMX and XLIFF, and is the author of XML Internationalization and Localization.

```
  <trans-unit id= "1">
   <source>First sentence. Second sentence.</source>
    <seg-source><mrk mtype="seg" mid="1">First sentence.
</mrk><mrk mtype="seg" mid="2">Second sentence.</mrk></
seg-source>
     <target><mrk mtype="seg" mid="1">Translated first
sentence. </mrk><mrk mtype="seg" mid="2">Translated second
sentence.</mrk></target>
  </trans-unit>
```

Figure 2: Segmentation in XLIFF 1.2.

```
  <unit id="1">
   <segment id="1">
    <source>First sentence. </source>
    <target>Première phrase. </target>
   </segment>
   <segment id="2">
    <source>Second sentence.</source>
    <target>Deuxième phrase.</target>
   </segment>
  </unit>
```

Figure 3: Segmentation in XLIFF 2.0.

```
  The two paragraphs to align:
  <p lang="en">He bought a yellow car. He loves yellow.</
p>
  <p lang="fr">Il adore le jaune. Il a acheté une voiture
jaune.</p>

  Their representation in XLIFF 2.0:
  <unit id="1">
   <segment>
    <source>He bought a yellow car. </source>
     <target order="2">Il a acheté une voiture jaune.</
target>
   </segment>
   <segment>
    <source>He loves yellow.</source>
    <target order="1">Il adore le jaune. </target>
   </segment>
  </unit>
```

Figure 4: Target order in XLIFF 2.0.

```
  <unit id="u1">
   <segment state="translated" subState="acme:post-edited">
    <source>They left the first day of the Month of the Red
Leaves.</source>
     <target>Ils partirent le premier jour du Mois des
Feuilles Rouges.</target>
   </segment>
   <segment state="reviewed" subState="acme:after-qa1">
    <source>The air was crisp and silent, portent of the
coming winter.</source>
     <target>L'air était froid et silencieux, présage de
l'hiver proche.</target>
   </segment>
  </unit>
```

Figure 5: Segments with different states.

content – an independent part of text that is embedded within another one, such as the text of the alt attribute of an HTML `<img>` within a paragraph – is represented in its own separate unit and linked to its referring inline code by a `subFlows` attribute.

Segmentation

As noted before, 2.0 drastically changes the way to represent segments. Figure 2 shows how 1.2 delimits segments using `<mrk>` elements with an `mtype` set to `seg`. Those `<mrk>` elements exist in the `<seg source>` and `<target>` elements.

In 2.0 the structure is reversed. As you see in Figure 3, the unit holds the segment elements, and each segment holds the source and target elements.

In 1.2 you can have content between the segment-delimiting `<mrk>` elements. In 2.0 there is a new element, `<ignorable>`, that holds any content outside the segments. In addition to the new representation, 2.0 comes with a new canResegment attribute that indicates if tools can change the existing segmentation in a given part of the document. You can set canResegment on `<file>`, `<group>`, `<unit>` or `<segment>`.

The last aspect of segment representation is ordering. In 1.2 the source and target segments are in separate containers and they are linked by the values of the mid attributes, so there is no need for a specific ordering mechanism. In 2.0 the source and target are within each segment. With such structure there is no implicit way to change the order of the target parts. So the new attribute order in `<target>` can be used if the target segments need to be in a different order than the source segments. Figure 4 shows an example of this.

The `<segment>` element also holds two attributes to indicate its status: `state` and `subState`. The `state` attribute can have the values `initial` (the default), `translated`, `reviewed` and `final`. If a process needs additional values, you can refine each state using a custom value specified in `subState`. The value of `subState` must start with a prefix that identifies the "authority" defining the values. This helps to avoid a clash between different toolsets. Figure 5 illustrates how the two attributes work. The `state` attribute allows a minimal level of interoperability, while

```
HTML content:
<B>Bolded</B> text<BR>

XLIFF representation:
<unit id="u1">
 <originalData>
  <data id="d1">&lt;B></data>
  <data id="d2">&lt;/B></data>
  <data id="d3">&lt;BR></data>
 </originalData>
 <segment>
<source><pcid="1"dataRefStart="d1"dataRefEnd="d2">Bolded</
pc> text<ph id="2" dataRef="d3"/></source>
 </segment>
</unit>
```

Figure 6: Unit with inline codes and their original representation.

```
<unit id="u123">
 <mtc:matches>
    <mtc:match id="1" ref="#m1" type="mt" origin="MS-
Translator-Hub" similarity="100">
   <source>He is a good friend of mine.</source>
   <target>Il est un bon ami à moi.</target>
  </mtc:match>
  <mtc:match id="2" ref="#1" type="tm" origin="myTM"
similarity="97">
   <source>Good friends</source>
   <target>Bons amis</target>
  </mtc:match>
 </mtc:matches>
 <segment>
 <source><mrk id="m1" type="mtc:match">He is a <pc id="1"
type="fmt">good friend</pc> of mine.</mrk></source>
 </segment>
</unit>
```

Figure 7: A unit using the Translation Candidates module.

`subState` provides some flexibility for customization.

Inline elements

The representation of the extracted content has been completely redesigned in 2.0. The content is made of the text itself as well as three types of inline elements: invalid XML character representations, inline codes and annotations.

XML cannot represent most control characters as normal text, even when using the standard escape notation. Nevertheless, because such characters may exist in extracted text (for example in software strings) XLIFF must be able to represent them. The `<cp/>` element takes care of that. For instance, `<cp hex="001b"/>` represents the character Escape `(U+001B)`.

XLIFF also must be able to store the original inline codes of the extracted document. For example, if the document is extracted from HTML and a paragraph contains a span of bolded text, you should be able to store the two `<b>` and `</b>` HTML tags. This is done using the `<ph/>` or the `<pc>` element, depending on whether the code stands alone or has content. If you cannot use a `<pc>` element because it overlaps another inline code or an annotation, or goes across segments, then you fall back to the `<sc/>` and `<ec/>` elements. The metadata stored in those elements is the same as in `<pc>`: switching back and forth between the two notations is lossless. You can also use the `<sc/>` and `<ec/>` notation when one opening or closing end of a paired code exists without its corresponding closing or opening end in the same unit.

You can store the native codes outside the content itself, in a list of data elements linked to the inline codes through their IDs, as displayed in Figure 6. With this representation, all text nodes in the source and target elements contain true text, while in 1.2 the content is a mix of text and codes often difficult to separate because they are all seen as text nodes by the XML parsers.

You can decorate each inline code with various attributes. For example, `dispStart / dispEnd / disp` provide a display representation of the code; `equivStart / equivEnd / equiv` provide a plain text equivalent for processing (word count, for example). The inline codes come also with a set of editing hints: `canCopy, canDelete, canOverlap,` and `canReorder` provide a set of directives that translation environments and other processing tools must honor. In 2.0 there are also provisions for handling added inline codes (such as when you need to put extra formatting in your translation).

Annotations are information added on top of the content and not necessarily present in the original document. For example, an annotation can be a comment from a translator on a specific portion of the text. Annotations have a greater role in 2.0 than in 1.2 because they can be used to anchor references to or from modules and extensions.

Annotations are represented with the element `<mrk>`, but you can fall back on `<sm/>` and `<em/>` when an annotation overlaps another one, overlaps inline codes, or spans across segments. Each annotation has the mandatory `id` and `type` attributes. The annotation can also have a `translate`, a `value` and a `ref` attribute. The semantics of the two last attributes vary based on the type of annotation. There are three predefined annotation types: `generic` (to tell if a span of text is translatable or not); `term` (to indicate that a span of text is a term and optionally to provide related information); and `comment` (to associate a short comment or a note with a span of text).

You can also define your own type of annotation and specify the role of

the <mrk> attributes for that custom type. Extension attributes can be used in <mrk> (and <sm/>). Those are the only inline elements where you can use extension attributes.

Fragment identifiers

Linked data are an important and increasing part of today's technologies. So it is imperative for 2.0 to provide a well-defined and interoperable mechanism to point to places inside an XLIFF document.

Unlike in traditional XML documents, an XLIFF document has several sets of IDs that are not necessarily unique within the whole document. So you cannot use something like ref="myFile.xlf#id1" because id1 may be the identifier of several elements. XLIFF has always had this particularity and finally 2.0 addresses it by defining its own fragment identifier syntax. While uncommon for XML documents, there is provision for this type of definition in the MIME Type specification.

The syntax provides a way to specify both the ID value and the context of the element being pointed to. You achieve this by telling in which file, group or unit the element occurs. For instance, if you want to point to the lone <ph/> element listed in Figure 1 you would use: ref="myFile.xlf#f=f1/u=u2/1", which means: the element with id="1" that is inside the element <unit id="u2">, which is itself inside the element <file id="f1">.

Each category of identifier, except one, has an assigned prefix letter (f for <file>, u for <unit> and so on). IDs with the prefix t are used for inline elements in <target>, while IDs without prefix are used for inline elements in <source> and for the and <ignorable> elements. Each module also has its own prefix. For example, to point to the second <match> element of the Translation Candidates module shown in Figure 7 (assuming the <unit> is in the same <file> as the previous example) you would use: ref="myFile.xlf#f=f1/u=u123/mtc=2", where mtc corresponds to the prefix for the Translation Candidates module.

The page lynx.okapi-xliff.cloudbees.net/fragments lets you try out the syntax to better understand how the mechanism works. The FragmentID Decorator utility available at xmarker.com/xliffutilities is also a good way to discover XLIFF fragment identifiers.

Modules

In addition to the Core, 2.0 provides eight specialized modules: Translation Candidates; Glossary; Format Style; Metadata; Resource Data; Change Tracking; Size and Length Restriction; and Validation. Each module has its own namespace and can evolve independently from the Core and the other modules. This modular aspect of 2.0 is very important because it allows tools to continue working with future versions of XLIFF without any change, as long as the Core and the modules they use do not change. In a sense, 2.0 is the first release of a composite standard where you implement and use only the blocks you need.

Because this introductory article focuses on the Core, we will not go through all modules. It is useful, though, to see how they work by looking at one example. Quite a few applications will want to use the Translation Candidates module. This module associates the text to translate with possible translations suggested through various mechanisms. In 1.2 we have this functionality with <alt-trans> (with its alttrans type attribute set to "proposal", which is the default). While <alt-trans> can be used for other things, in 2.0 the Translation Candidates module is strictly reserved for this function. The module's namespace is identified by the URI urn:oasis:names:tc:xliff:matches:2.0 and it uses the fragment identifier prefix mtc (which is also its suggested namespace prefix).

As shown in Figure 7, a <match> element is associated with a span of content using its ref attribute. If the entry matches a span already delimited by an existing inline element (like the <pc> element in the example), the reference can point to that existing element. Otherwise, it can use the dedicated translation candidate annotation defined by the module. This mechanism allows the representation of the candidates for the whole segment as well as for substrings within the segment or even across several segments. More and more tools offer such a capability, but that is not supported in 1.2.

With each match, you can indicate the similarity with the original source text, the quality of the translation and a suitability score that you construct from different parameters. It is currently not possible to have standard values for those attributes across different tools because each tool uses different algorithms for evaluating the candidates.

Extensions are the elements and attributes that are not part of the XLIFF standard but allowed in selected places of an XLIFF document. They use their own namespace and are very similar to modules. Not allowing extensions would make the standard much less usable – you want to be able to use extensions for features that are not yet in the standard. In 2.0 they can be seen as a way to develop new modules. In fact, implementers will find they can easily treat extensions like the modules they do not support. How you can use extensions is restricted by constraints and processing requirements.

One example of this is how extensions must define their identifiers: They can only use attributes named `id` or the `xml:id` to denote IDs. The ID values must be unique within the immediate file, group or unit that encloses the element that holds the extension, and the value must be compatible with the type NMTOKEN. Extensions must also register the prefix they use in the fragment identifier notation. It is easy to implement these restrictions. They are in place to provide some degree of interoperability even with the tools that have no knowledge of the extensions.

Conformance

As Andrew Pimlott pointed out during a TAUS conference three years ago in his presentation "XLIFF 2.0: Great (Processing) Expectations," the new version of XLIFF needed clear constraints and processing requirements that implementers can easily follow and rely upon. In 2.0 we have plenty of those and an application must abide by them to be conformant.

For example, when a tool un-segments content already segmented, it must first honor the `canResegment` flag and then follow the processing requirements defined for merging existing segments, such as: what resulting values to set for the attributes that have different values on two segments being joined.

Conforming to the specification is both simpler and more difficult than in 1.2. It is simpler because 2.0 spells out what you need to do. It is more complicated because there are many more things that you need to do.

You cannot validate the constraints that come with 2.0 using only schemas. Some of the verification must be done programmatically. You can use Lynx, a free Java open-source tool, to validate the Core and some of the modules. There is also an online version of Lynx accessible at lynx.okapi-xliff.cloudbees.net/validation. Bryan Schnabel's utilities available at xmarker.com/xliffutilities also provide validation, including for all the modules.

Often software developers try to follow Postel's Law: "Be liberal in what you accept; be conservative in what you send." The second part of the law is always good advice. The first part, though, is perhaps not so good. While it may make sense in some cases, it is not always a good thing to try to accommodate invalid documents. XLIFF is not HTML or e-mail: there are not many thousands of applications generating XLIFF output or people coding XLIFF manually. There is no justification for generating invalid XLIFF. In our context, expecting a valid document must be the norm. That said, you will get invalid data because mistakes always happen. When that occurs: first be sure that you do detect the problem and second (whether you accept the document or not) make sure that you provide feedback to the sender. Interoperability works by cooperating. In today's world of web services and automation workflows, it is relatively easy to report errors. So help promoting better interoperability, adapt Postel's Law to "Be loud and vocal about the wrong XLIFF you get; be conservative in what you send."

Challenges and implementations

No format is perfect and 2.0 has a number of disadvantages compared to 1.2. For example, it is wordier and the extracted documents can get quite large. Compression can alleviate that issue.

Also, the use of annotations, while powerful, makes things a bit more complicated for simple content. This is the

price to pay for the benefit of having a format that follows generic patterns to allow more features and future evolution with some stability.

Another drawback of 2.0 is that it does not support some of the features found in 1.2. Remember that modularity is at the center of the new specification and the missing functionality can be part of new modules.

What tools use XLIFF 2.0? There are not many yet, because 2.0 is only now finishing its journey through the standardization process. Even so, there are already some implementations. The University of Limerick has a number of XLIFF 2.0-aware applications: Localisation Knowledge Repository, a tool used to help authors produce more localizable documents; LocConnect, a tool for managing translation workflows; Workflow Recommender, a tool to determine the best workflow for an XLIFF input; and Service Mapper, a tool integrating several machine translation engines. Together they implement the Core and three modules: Metadata, Glossary and Translation Candidates.

As noted before, Bryan Schnabel offers several 2.0 open-source tools. In addition to the FragmentID Decorator utility that can also perform validation (xmarker.com/xliffutilities), the DITA XLIFF Roundtrip project (sourceforge.net/projects/ditaxliff) offers extraction and merging functions for DITA documents; and the XLIFF Roundtrip tool (sourceforge.net/projects/xliffroundtrip) provides extraction and merging functions for XML documents in general.

The Okapi XLIFF Toolkit project (code.google.com/p/okapi-xliff-toolkit) provides a library to read, write and manipulate 2.0 documents. It also includes a validation tool, with various utilities and options. The library is used in several of the other Okapi components: You can extract and merge 2.0 documents using Rainbow or even edit 2.0 documents directly in the popular OmegaT translation editor using the Okapi Filters plug-in.

Finally, the ITS Interest Group is mapping ITS 2.0 (the Internationalization Tag Set W3C Recommendation) to XLIFF 2.0 through a future module, so the many metadata of ITS can be used seamlessly in XLIFF documents. The Okapi library already has some support for this mapping.

It will take time for 2.0 to replace 1.2. But as the developers start looking at the new version, they will find that it offers a better foundation for their tools: a stable base (the Core) with modules that they can implement, or even create, as they need.

These developers may also notice that the overall definition of 2.0 takes a step toward an important change: the specification often defines a processing model rather than just a format. The localization industry will always need a standard way to represent extracted content; however, in a world where the shift of storage solutions to databases and cloud-based repositories increases, exchange formats need to be complemented by specifications for object models, common APIs and protocols. **M**