XLIFF 2 Extraction and Merging Best Practice, Version 1.0

Edited by: David Filip and Ján Husarčík
WG chaired by: Rodolfo M. Raya, Andreas Galambos

TAPICC T1/WG3

Additional artifacts

This prose specification is one component of a Work Product that also includes:

- Extraction and merging examples from https://galaglobal.github.io/TAPICC/T1/WG3/prd01/extraction_examples/

  An unstable editorial version of the examples might exist at https://galaglobal.github.io/TAPICC/T1/WG3/extraction_examples/

Related work

This note provides an informative best practice for XLIFF 2 Specifications:

- XLIFF Version 2.1 [XLIFF-2.1]
- XLIFF Version 2.0 [XLIFF-2.0]
- ISO 21720:2017 [ISO XLIFF]

Status

This Informational Best Practice was last revised by TAPICC T1/WG3 or the TAPICC Steering Committee on the above date. The level of approval is also listed above. Check the “Latest version” location noted above for possible later revisions of this document.

Contributions to this deliverable or subsequent versions of this deliverable can be made via the GALA TAPICC GitHub Repository subject to signing the TAPICC Legal Agreement.

Citation format

When referencing this specification the following citation format should be used:

[XLIFF-EM-BP]

XLIFF 2 Extraction and Merging Best Practice, Version 1.0 Edited by David Filip and Ján Husarčík. 28 June 2018. Public Review Draft 01. https://galaglobal.github.io/TAPICC/T1/WG3/prd01/XLIFF-EM-BP-V1.0-prd01.xhtml. Latest version: https://galaglobal.github.io/TAPICC/T1/WG3/XLIFF-EM-BP-V1.0-LP.xhtml.

Notices

The Translation API Cases and Classes (TAPICC) initiative is a collaborative, community-driven, open source project to advance API standards in the Localization industry. The overall

purpose of this project is to provide a metadata and API framework on which users can base their integration, automation and interoperability efforts.

The usage of all deliverables of this initiative - including this specification - is subject to open source license terms expressed in the BSD-3-Clause License and CC-BY 2.0 License, the declared applicable licenses when the project was chartered.

- The 3-Clause BSD License (BSD-3 Clause): https://opensource.org/licenses/BSD-3-Clause
- Creative Commons Legal Code (CC-BY 2.0): https://creativecommons.org/licenses/by/2.0/legalcode

28 June 2018

Abstract

This Informational Best Practice specification targets designers of XLIFF *Extracting* and *Merging* tools for content owners. It gathers common problems that are prone to appear when *Extracting XLIFF Documents* from HTML, generic XML, or Markdown. This specification shows why some *Extraction* approaches will cause issues during an *XLIFF Roundtrip*. This best practice guidance provides better thought through alternatives and shows how to use many of advanced XLIFF features for lossless Localization roundtrip of HTML and XML based content.

Table of Contents

Terminology and Concepts

Apart form terminology and concepts defined here, this specification makes heavy use of terms defined in the XLIFF Standards [XLIFF-2.1] such as: *Extractor*, *Merger*, *Translation*, *XLIFF Document*, *XLIFF-defined*, etc.

context hints
: *XLIFF-defined* attributes on structural or inline elements providing additional contexts, such as `disp` or `equiv`. Attributes `fs` and `subFs` defined in the XLIFF Format Style Module are also considered *context hints*.

inline codes
: `<sc/>`/`<ec>` pairs, orphaned `<sc/>` or `<ec/>`, well formed `<pc>`, standalone `<ph/>` and `<cp>` are *inline codes* used to represent native format inline markup in *XLIFF Documents*.

Note

Inline codes can reference original data in the *XLIFF Core* <originalData> standoff element.

markers

`<sm/>`/`<em/>` pairs and well formed `<mrk>` are *XLIFF-defined* inline marker elements designed for inline annotations of content with metadata.

Note

Markers are distinct from *inline codes* (see). Markers can be combined with *Module* or Extension based standoff elements for rich metadata that would be complicated or impossible to display inline.

1. Introduction

This Informational Best Practice targets designers of XLIFF *Extracting* and *Merging* Tools for content owners. *XLIFF Roundtrip* designers of all kinds will benefit, no matter if they design their *XLIFF Extractor/Merger* for corporate or blog use.

Extraction and *Merging* behavior is out of the normative scope of OASIS XLIFF Specifications. Although those specifications do provide some guidance for *Extractor* and *Merger Agents*, XLIFF TC did not attempt to prescribe how exactly to use XLIFF to represent native content. This is mostly because XLIFF is a native-format-agnostic Localization Interchange Format.

This specification gathers common problems that are prone to appear when *Extracting XLIFF Documents* from HTML, generic XML, or Markdown. This specification shows why some *Extraction* approaches will cause issues during an *XLIFF Roundtrip*, issues often so severe that *Merging* back of target content will not be possible without costly post-processing or could fail utterly. This best practice guidance provides better thought through alternatives and shows how to use many of the advanced XLIFF features for lossless Localization roundtrip of mainly HTML and XML based content. Often, there are no ultimate prescribed solutions, rather possible design goals are described and best methods how to achieve them proposed.

The concepts described in the next section are usually grouped by a common theme with no particular order.

2. Specification

2.1. XLIFF Structure

Taking time to consider not only what to *Extract*, but how to *Extract* it, and how to structure the *XLIFF Document* can significantly reduce number of issues during the roundtrip and enable usage of additional features offered by the XLIFF Standards.

2.1.1. File Structure

Native formats can contain structural elements dividing its content into parts, such as title, body, header and footer, or tables, lists and divs for markup languages; or windows, dialogs, and menus for software resources.

Representing native structural elements in XLIFF using potentially nested `<group>` elements can be useful for providing, and correctly scoping:

- additional context (`name`, `type`, `subType`, attributes from Format Style Module)
- restrictions (`canResegment`, `translate`, attributes from Size and Length Restriction Module)
- whitespace handling (`xml:space`)

- information from modules such as:
  - Metadata
  - Validation
  - ITS

Most of the above can still be achieved without using the optional `<group>` elements. It will be however at the cost of high redundancy of unit level metadata and possibly cause potentially illegal overload of some of the *XLIFF Core* features.

Example: group.

2.1.2. Role of the `<unit>` Element

Extractors set the XLIFF structure, which cannot be modified (an absolute prohibition expressed in the XLIFF Standards) during the roundtrip at the unit level or higher. Ensuring or not that the appropriate relationships drive *Extraction* from structures of the native format into XLIFF `<unit>` elements can make all the difference between hindering or crippling the roundtrip and making the most of XLIFF features in a compliant way.

Severe problems can be caused by both extremes: too many or not enough `<unit>` elements. Especially dangerous is Extracting every segment as a separate `<unit>` element as this will effectively prevent *Modifiers* downstream to change segmentation. Changing segmentation within logically self contained units is one of the key advantages of the XLIFF 2 structural data model that makes a distinction between the immutable high level structure (`<unit>` and higher) and the transient segmentation structure (`<segment>` elements within each `<unit>`) that interplays with the inline data model and the inline annotations' logic.

Example: mapping_to_unit.

2.1.3. Controlling Segmentation

Depending on *Extraction* rules for mapping of original document structures into *XLIFF Documents*, individual sentences within a paragraph; verses within a stanza; items or entries of a list; rows or cells of a table; items of a dialog window; and so on might be *Extracted* as segments of a single unit. While it is generally not advisable to perform segmentation at the time of *Extraction*, *Extractors* that *Extracted* multiple sentences, verses, entries, rows, and so on into a single non-segmented unit (a single `<segment>` element within each `<unit>`) and their corresponding *Mergers* need to expect that the *Modifiers* will need to transform them into individual segments within the same unit (multiple `<segment>` elements representing individual sentences, verses, and so on within each `<unit>`) during the roundtrip.

In cases where subsequent *Modifiers* cannot be reasonably expected to detect the segmentation logic, for instance due to the lack of knowledge of the original format logic, the content owner is advised to perform the segmentation and protection of that segmentation before sending their XLIFF Documents for the service roundtrip.

While it's generally desirable to be able to *Modify* segmentation within a unit during the roundtrip, doing so in some of the above cases might prevent *Merging*, cause build issues, or have negative impact on target product user experience.

Attribute `canResegment` can be used *with care* to control segmentation *Modification* behavior. Its values need to be controlled by rules derived from the structural and inline logic of the native format. For instance, more often than not it will make sense to set `canResegment` to `no` for:

- lists
- tables or table rows

- UI elements

Extracted as segments of a unit.

In UI elements and tables, it is likely that the available segmentation needs to be protected, on the other hand, it is advisable not to change the default `canResegment="yes"` for normal paragraph text and similar, see Role of the <unit> Element.

Importantly, preventing *Modification* of segmentation using the attribute `canResegment` (set to `no` when necessary) will *not* prevent reordering of segments within a unit using the `order` attribute on the `<target>` elements within the same unit. So in case an ordered list needs to be for instance alphabetically collated, translators can do so even in case the `canResegment` attribute is set to `no`. The segmentation logic of the native format remains protected without preventing collation. This would be all hampered if the *Extractor* decided to *Extract* each segment as a separate unit, which is the most evil practice that cannot be discouraged enough.

Example: mapping_to_unit.

2.2. Inline Codes

Guidance for processing standalone and spanning inline functional and formatting elements of localizable content can be summarized into the following list:

- Perform complete extraction
- Represent spanning code using `<sc/>` and `<ec/>` (or `<pc></pc>` where possible)
- Represent standalone code using `<ph>`
- Include all (even the outermost) *inline codes* in the *Extracted* content
- Additional details in the XLIFF2 prose.

2.2.1. Representing Spanning Codes

Spanning codes in the original format are created by an opening code, the content, and the closing code. In HTML that can be `<bold>text</bold>`, in RTF `\b text \b0`.

In *XLIFF Documents*, such code can be always represented with an `<sc/>`/`<ec/>` pair, or with spanning `<pc></pc>`, only for well formed markup.

Ideally, the original format is documented well enough to instruct *Extractors* about the role of each *inline code*. For example, XML Schema allows to declare elements using the keyword EMPTY. This way, all elements that are not declared EMPTY, can be represented as described above. To further help the *Extraction* process, the following recommendation could be implemented in the original XML format: “For interoperability, the empty-element tag SHOULD be used, and SHOULD only be used, for elements which are declared EMPTY.”[XML].

Following this recommendation of the XML specification, an empty `<span>` ought to be encoded as `<span></span>` and therefore represented as an `<sc/>`/`<ec/>` pair in *XLIFF Documents*, unlike the always empty `<br/>` that has to be represented as a standalone placeholder code `<ph/>`.

This concept is illustrated by the *bad practice* example `spanning_as_ph`.

This kind of bad practice encoding doesn't inform the *Translating Agent* (human or machine *Modifier*) that the original code formed a span and effectively the original spanning code is not protected during the roundtrip. The standalone `<ph/>` codes can be switched or one of them omitted; simply, the span is likely to end up misplaced, malformed, or empty simply because the *Translation* editor cannot convey to the translator that the codes need to enclose a certain portion of the original content and what is the semantics of the original code span.

2.2.2. Outermost Tag Pairs

In some cases, the *inline codes* can enclose the localizable string in a way that could suggest omitting them in the *Extracted* text. For example, a paragraph containing only a link text, could be *Extracted* as the link text only, without the `<a></a>` decoration being represented. This relates to the previous *bad practice* example with the spanning tag represented as two empty `<ph/>` elements.

In case the `<a></a>` decoration is not represented, the translator loses valuable context (they cannot check the link), more importantly they don't know that the text is a link text, and moreover are unable to add any text outside of the link span, which might be advisable or even mandatory in certain locales.

Ideally, a consistent approach to all *inline codes* ought to be used during *Extraction*.

See the relevant *bad practice* example outermost_inline_excluded.

2.2.3. Incomplete Extraction of Inline Codes

Some implementers choose not to *Extract inline codes* at all and use one of the following approaches instead:

- CDATA sections as content of `<source>` and `<target>` elements (cdata)
- Escaping of native codes using XML entities (inline_codes_plain_text)

Doing one of the above can be used as a useful *interim Extraction* step when producing *XLIFF Documents* that are fit for roundtrip. However, it is strongly discouraged to send *XLIFF Documents* with the inline content not fully parsed for Localization roundtrip.

Such incomplete *Extraction* leaves *inline codes* unprotected and increases the risk of their corruption during the roundtrip, simply pushing the problem of *inline code* handling downstream.

According to the XLIFF [XLIFF-2.1] Standards, *Modifiers* can perform secondary parsing:

"*Writers may* preserve original CDATA sections" (meaning that it is entirely optional to preserve CDATA sections and that *XLIFF Writers* are not obliged to preserve CDATA sections)

and

text can be converted into inline codes.

Mergers have to accept XLIFF files with valid modifications, even though they may ignore the added codes.

Finally, it is considered an XML internationalization best practice to avoid CDATA sections in localizable content. This best practice is of course also valid for XLIFF `<source>` elements.

Strictly speaking, it is not illegal to create *XLIFF Documents* that contain CDATA sections or unparsed entities instead of fully parsed XLIFF inline content. However, considering all of the above, it is clear that unparsed inline content makes XLIFF Documents unfit for a fully interoperable roundtrip. Again, strictly speaking, *XLIFF Documents* with unparsed inline content are *capable* of roundtrip but all the effort that is saved on *Extraction* will cause unpredictable issues and hence even more effort when *Merging* back.

Implementers need to consider that *XLIFF Documents* with unparsed inline content are very likely to return with critical inline syntax or formatting corruptions that cannot be prevented on CDATA sections or entities that are both opaque to *XLIFF Modifiers*. Such corruptions are likely to prevent proper functionality of target content in the native environment. In case *XLIFF Modifiers* do perform the secondary parsing of content unparsed on *Extraction*, which is allowed by the standard, corruption will be prevented, however, *Mergers* will need to perform *unparsing* to facilitate merging back into the native environment, because XLIFF Modifiers are not and cannot be obliged to *unparse* back to

CDATA sections or entities not knowing the *Extraction* and *Merging* logic of the *XLIFF Document* originator.

2.2.4. Representing Multiple Subsequent Codes

As original *inline codes* can occur in clusters, for instance as nested formatting, implementers could be tempted to combine such markup on *Extraction* and represent it as a single inline element.

This kind of *Extraction* is likely to prevent potentially desirable *Modification* of *inline codes*, affecting *Translation* quality. It will also prevent usage of fine grained code metadata (for instance context, display, and editing hints) or automated format validation during the roundtrip.

On the other hand, some potential benefit can be perceived in reducing markup inside segment content, which is useful in CAT tools that cannot properly display the inline codes (render information available through original data or context hints). In such tools, less markup reduces the visual clutter and makes the translatable text more readable. This can be solved by proper choice of CAT tools (short term) or by large buyers requesting that offending tool vendors do support proper rendering of *inline code* data and metadata (mid and long term).

Implementers need to consider the pros and cons of both approaches and use the one that best matches their business need.

For examples see multiple_codes_represented_as_single.

2.3. Target Content in Extracted XLIFF

This section focuses on reasons whether or not to populate the `<target>` element during *Extraction* or *Enriching* and when to do so, if at all.

Generally, one should omit the `<target>` element, unless there is an added value and also in cases where the specification offers another dedicated solution. Proper support of the state machine during the whole roundtrip helps *Agents* to process and validate the *XLIFF Documents* as intended.

When looking at the situation from the *microservices architecture* point of view, the *Extractor*/*Merger* ought to be implemented as just that — a single purpose *Extraction*/*Merging* service that delegates any other operations, such as segmentation or *Enriching* to other specialized services.

Output of such extractor would be a *target language* agnostic *XLIFF Document* with source content only, possibly with additional modules/extensions which could not be generated after extraction, for example Size and Length Restriction Module or Format Style Module.

Unless the implementer has a specific need to create *target language* specific instances of the extracted *XLIFF Document*, for instance by *Enriching* with translation candidates, the *Extracted XLIFF Document* could and ought to be sent downstream for the Localization roundtrip as-is.

2.3.1. Inserting Source Content into `<target>`

The copy of the source content in `<target>` elements generally does not provide any advantage during the XLIFF roundtrip. On the contrary, it brings disadvantages such as needlessly increasing the size of the *XLIFF Document* or enforcing existence of the `trgLang` attribute with a specific BCP 47 compliant value. Populated `<target>` elements are also likely to prevent segmentation modification, unless the target content is intentionally removed (which service providers are understandably hesitant to do). Not the least issue is that the source content copied to the target actually is *not* in the target language indicated by the BCP 47 tag on the XLIFF root element, which can cause a host of other processing issues. The *bad practice* of populating `<target>` elements with source content used to facilitate parsing and editing of XLIFF in *Translation* editors or generic XML editors that didn't have XLIFF support and could only open for translation certain elements in generic XML formats. As such, this practice is strongly discouraged.

Bad practice example: source_in_target.

2.3.2. Inserting Possible Translations into <target> elements

Enriching Agents can use translation memories, machine translation engines, or other means to obtain suitable translation candidate strings in the target language to be used later in the roundtrip, for example as suggestions for translators, to achieve better leverage, or to get higher consistency with previous translations.

Using the <target> element for storing such translation candidates limits the number of the possible proposed translations to a single one per segment. Moreover, this way it's not possible to pass critical metadata about the translation candidate, such as its origin, similarity, or quality (all those are available in a dedicated module), causing interoperability issues for *Agents* without prior knowledge of the workflow.

Inserting translation candidates into <target> elements during *Extraction* or *Enriching* constitutes an illegal overload of the core element with a clearly set purpose. The Translation Candidates Module was designed exactly to provide translators with multiple translation candidates along with metadata that facilitate decision making and effective reuse. Moreover, the module can address sub-segment matches.

Example: pre-populated_target.

2.3.3. State Machine

The XLIFF specification contains attributes for managing a segment state machine. The attributes used are <state> and <subState>. The <subState> attribute can only be used as long as the <state> attribute is used. The <state> attribute is for high level interoperability. The <subState> attribute allows implementers to define private sub-state machines that can give more fine-grained sub-states based on their private workflow needs.

The <state> attribute defines just a high level four states state engine. The values are `initial`, `translated`, `reviewed`, and `final`. Although this attribute is *optional* on the element, it is assumed as having the default value `initial` whenever not used explicitly. There are some important advantages to using the state machine explicitly. Importantly, <target> elements are optional in the `initial` state. So if you want to even enforce <target> existence in your deliverables you should be using at least the high level four states state engine provided by the *Core* attribute `state`. Setting the `state` attribute of a to `translated` or later does enforce <target> existence within that segment.

Using the high level states `reviewed` and `final` gives you even more control over the progressive validation of the *XLIFF Documents* you're roundtripping. All of the states `translated`, `reviewed`, and `final` will trigger validation of the inline data model within <target> elements, which is not being validated in the `initial` state where even the existence of <target> elements is not assumed. Violations of the inline data model including Editing hints are being tested in all states more advanced than `initial`. Those violations are considered "Warnings" at `translated` and `reviewed` states. Only in the `final` state, those violations will become actual "Errors" that render the *XLIFF Document* invalid.

2.4. Editing and Context Hints

The XLIFF specification provides a number of attributes that allow to manage the behavior and validation of structural and inline elements; such as controlling the localizability of text; protecting non-deletable inline codes, or preserving their order; controlling the segmentation modification; or providing additional context to other agents downstream.

The default values of the editing hints and potential need to set them otherwise need to be considered when creating *Extraction* rules to prevent issues which can be only identified by automated validation with editing hints set as intended.

2.4.1. Non-deletable Inline Codes

Original source text can contain functional inline codes apart from formatting ones, such as software placeholders to be replaced during runtime. Removal of these placeholders or other functional code, either intentional or accidental, during the *Translation* roundtrip can produce valid *XLIFF Documents* that will nevertheless fail to merge back, cause build failures later on, or create other functional issues in the *Translated* product.

The XLIFF specification provides the editing hint `canDelete` with its default value set to `yes` that is thus automatically used or can be explicitly set on any inline code. For most of the formatting codes, the default value `yes` is fine, so that there is no need to set the attribute explicitly most of the times. The default value means that the codes can be removed during localization as the translators see fit. A typical example is the need to remove italics or bold formatting codes in Chinese or Japanese target content. These languages don't use typographical methods of emphasis and non-deletable formatting codes tend to complicate life of translators into such languages. On the other hand, *Extractors* need to take care to set the `canDelete` attribute to `no` explicitly whenever an inline code is critical for *Merging* back of the *XLIFF Document*, their build process, or product functionality.

2.4.2. Preserving Order of Codes

In case the order or nesting of inline codes in the original document is prescribed (for instance by a schema), it has to be preserved in the target content during the localization roundtrip to prevent *Merge* issues, or validation fails after *Merging*.

The attribute `canReorder` on the inline code determines, whether each code can be moved before, or after another code. Again, the default value of this attribute is `yes` meaning that the inline codes can be reordered as the translators see fit.

This attribute is used to create and protect non-reorderable sequences of inline codes if necessary for proper inline code functionality.

Example 1. Example of a non-reorderable source sequence of inline codes

```
...
<source><pc id="1" canCopy="no" canDelete="no"
 canReorder="firstNo">
            <pc id="2" canCopy="no" canDelete="no"
 canReorder="no">this is linktext</pc>
            <ph id="3" canCopy="no" canDelete="no" canReorder="no"/
>
        </pc>
</source>
...
```

Since this attribute is supported by native XLIFF validation artifacts (*XLIFF Core* Schematron Schemas), potential reordering of the `<pc/>` and `<ph/>` tags in the corresponding `<target>` element will be called out when validating `<segment>` elements with the state more advanced than `initial`. See also the Sate Machine section.

Example: editing_hints_canReorder.

2.4.3. Providing Context

The *Agents* in the roundtrip, human or machines, need enough information to make appropriate decisions regarding operations on inline codes, and how the codes impact the adequacy and fluency of the target text, the context in short.

These additional metadata can be provided using the *context hints* attributes: `disp` (`dispStart`/`dispEnd`), `equiv` (`equivStart`/`equivEnd`), `type`, and `subType`.

The XLIFF Standards provide the `<originalData>` that ought to be used to store the native content of the represented inline elements. Its `<data>` child can even contain CDATA sections that are strongly discouraged in XLIFF content.

Original data, as a sort of internal skeleton, are likely to be intended to facilitate collaboration between the *Extractor* and the *Merger*, not necessarily suitable for use by other roundtrip *Agents*. The original data content can be, for instance, too long to render correctly in the CAT tool.

Extractors are encouraged to populate the value of `equiv` attributes with a suitable plain text representation of the original data and the corresponding `disp` attributes with a user-friendly variant of the same.

Alternatively, the same task could be performed further downstream by an *Enricher* that can understand the content of the `<originalData>`. Separating features of *Extractors* and *Enrichers* is in line with SOA and microservices architectures.

Example: context_hints.

2.4.4. Considerations for Using Spanning Codes

Compared to the `<pc>` pair tag, which can be only used to represent well-formed spanning codes within a single `<segment>`, the more universal `<sc/>`/`<ec/>` pair can handle segmentation changes, span across segments, other codes, or annotations, and even represent orphaned native inline codes. Sounds great, so why use `<pc>`at all?

The fact that `<sc/>`/`<ec/>` pairs do support for overlapping codes will, however, create an issue in a situation where `<sc>`/`<ec>` pairs are used to represent multiple well-formed spanning codes without setting their `canOverlap` attribute to `no`. In such cases, the well-formedness of the original codes is not protected and can be corrupted during the roundtrip. It will be impossible to prevent this corruption with native XLIFF methods unless the `canOverlap` and possibly `canReorder` attributes are properly set. So in case of representing well-formed native markup, using the `<pc>` pair tag is likely to be easier for the *Extactor*. On the other hand, it is important to consider that transforming the `<pc>` pair tag into an `<sc/>`/`<ec/>` pair is always allowed. So the *Mergers* need to prepared to handle `<sc/>`/`<ec/>` even in case their corresponding *Extractor* used only the `<pc>` pair tag.

Example: editing_hints_canOverlap.

2.5. Miscellaneous

There are many other good or bad practice concepts that do not belong to any of the above categories. Some of them are listed in this section:

2.5.1. Value of the Attribute `id`

Implementers could be tempted to store values of resource Ids in the `id` attribute of XLIFF structural or elements. It should be stressed that the XLIFF `id` attribute is intended for internal and external addressing of XLIFF fragments. While the XLIFF `id` value is restricted to *NMTOKEN*, the native format might not have such a restriction. Invalid characters in XLIFF `id` attributes will render the whole *XLIFF Document* invalid. Although this would typically be discovered as soon as the first validation occurs, it can still be costly to fix in a large or a long running project.

Note

A real life example:

A long running software localization project used its resource Ids as XLIFF `id` attributes in the role of the guaranteed key for In-Context Exact TM leverage. After a while, a new developer changed the style for generating resource Ids. The new Id did not fit the

NMTOKEN restriction, which led to a complex redesign of the localization and TM leverage workflow.

The XLIFF `name` attribute is designed to store the original identifier of the resource, it can be any string without restrictions. On the `<file>` element, the `original` attribute can be used for the same purpose.

Example: id_and_name.

2.5.2. Whitespace Handling

Whitespaces can be important inside nodes such as `<pre>`, containing for instance code samples, and *Modifying* them during the roundtrip is not desirable.

Whitespaces (more than one of the whitespace characters in a sequence) are, however, generally insignificant in the text nodes of markup formats, such as XML or HTML, and can be changed anytime (even not intentionally as an XLIFF transform) by, for example, reformatting and indentation (so called pretty-printing) without affecting the layout of the rendered document.

Thus, one cannot indiscriminately either preserve, or normalize. Since most of TMS and CAT tools penalize whitespace discrepancies, the leverage could be negatively affected if whitespaces are *Modified*, and layout of nodes with significant whitespace could be corrupted.

The general best practice, also taking into account the ITS Preserve Space data category is the following:

The *Extractor* itself ought to

1. Normalize native content where possible, not relying on other *Agents* in the roundtrip to do so and

2. protect XLIFF structural elements with `xml:space` set to `preserve`.

Please note that the XLIFF default for `xml:space` is `default`. Therefore, it is important to ensure that content with mixed whitespace behavior is normalized and then protected with `xml:space` explicitly set to or inherited as `preserve`. The default *XLIFF Core* behavior is only useful if all whitespace is globally insignificant.

Additional details in the XLIFF spec.

Example: xml_space_preserve.

2.5.3. Protecting Non-localizable Content

There are cases, when it's necessary to prevent localization of inline content parts otherwise exposed to localization, be it brand names, or functional inline code, such as software placeholders.

XLIFF offers two options for :

- translate annotation
- `<ph>`

each of them having different purpose and offering different features and options.

Careful consideration is necessary to decide which way to protect a particular non-translatable string type, as the two methods are neither equivalent nor interchangeable.

Usually, the translate annotation is suitable for protecting linguistically significant content, e. g. non-localizable brand or product names, while `<ph>` is better for representing standalone codes without a syntactic role, typically standalone formatting artifacts such as `<br/>`.

Note

XLIFF Core validation artifacts do not support validating translate annotations by design. It is the ultimate decision of the *Modifier* if a span annotated as non-translatable will indeed stay unchanged based on linguistic and context considerations. If the validation of the non-translatable annotations is necessary, it needs to be added for instance as custom validation code or a custom Schematron rule, based on particular business needs and validation infrastructure options.

Placeholders and variables to be replaced with syntactically significant content on runtime are a particularly difficult use case to address. The functional variable usually doesn't provide much of a context for the translator even in case when not replaced by the `<ph>` element and just surrounded by the do-not-translate annotation. The best way to represent such variables is to use `<ph>` with the `type` attribute set to `ui` and the `subType` attribute set to `xlf:var`. See the XLIFF Constraints for the `subType` attribute. In general, the `<ph>` elements need to be accompanied by appropriate context and editing hints. The `disp` is suitable to display (in a CAT tool GUI) an example value that is likely to be used on runtime. The same or a related value can be used also for the plain text equivalent (rendering) hint `equiv`.

Example: ph_and_mrk.

2.5.4. Merging Translated Content

Modifiers can perform various *valid* transformations during an XLIFF roundtrip. XLIFF compliant *Mergers* need to be able correctly handle all of them, as those changes are canonical validity preserving operations. See in particular the clause 2.e. of the XLIFF Conformance section.

These operations are (in order of importance or severity of issues caused if ignored):

- Converting CDATA sections and parseable text (such as XML entities) into XLIFF *inline codes*.
- Segmentation *Modification*,
- Equivalent conversion of `<pc>` elements into `<sc>`/`<ec>` pairs (always allowed and possible) and *vice versa* (only possible with well-formed spanning codes),
- Adding, and removing of *inline codes* (taking into account the editing hints and their Processing Requirements),
- Content *Enrichment* with Annotations and *Context Hints*,
- Performing of any other changes allowed by Processing Requirements.

Extraction not following best practices usually just shifts the problems further downstream, forcing other *Agents* to mitigate the inherited issues, more often than not leading to unexpected, undesirable, or unpredictable results that will trip over the *Merging* after the roundtrip.

Additional guidance is also available in the XLIFF Best Practice for *Mergers*.

Example: merging.

2.5.5. Selecting Language Tags

Agents in the roundtrip, machine and human, need to be able to sufficiently identify the languages used in *XLIFF Documents*. The two main languages (the source and the target language) of the XLIFF bitext are primarily specified by the attributes `srcLang` and `trglang`. The optional `xml:lang` attributes on the `<source>` and `<target>` elements are directly inherited from `srcLang` and `trglang` respectively and are in fact provided only for generic XML Processors interoperability. The attribute `itsm:lang` serves just to define inline foreign language spans via annotations if necessary.

What language tag to use is usually not an issue for languages like Slovak (`sk`) or Czech (`cs`) that are spoken predominantly in one country and encoded exclusively using one script. This becomes a more prominent question for languages used in different regions, such as English (`en-GB`, `en-US`); using various scripts, for example Uyghur (`ug-Arab`, `ug-Cyrl`, `ug-Latn`); or having multiple variants like Basic English (*en-basiceng*).

The use cases for the correctly set fine-grained language tags vary from simple, such as spell-check, which will behave quite differently for `en-GB`, compared to `en-basiceng`; to more complex, like using `fr-FR` as a reference language for a *Translation* into `fr-CA` .

MT engines will return Serbian encoded with the Cyrillic script (`sr-Cyrl`) output when the request contains the language tag *sr* albeit the user might have meant and expected Serbian written with the Latin script (`sr-Latn`). Human translators would hopefully ask which of the two was the desired one or just provide `sr-Cyrl` as the machines. More dangerous than translators would be undocumented internal mapping tables and custom business rules facilitating communication between roundtrip actors that could assume different defaults when not given an unequivocal language tag, or worse ignore a valid fine-grained language tag they don't cover.

The XLIFF Standards prescribe that the BCP 47 language tags are to be used as values for attributes specifying human natural languages used in the *XLIFF Documents*. The Unicode Consortium offers a tool available online, which can help to perform basic validation of selected language tags.

Generally, language tags need to be carefully chosen for source, target, and reference languages in *XLIFF Documents*, and it is worth your time to consult external resources, or language experts; even more so, if you are not familiar with the defaults for the language in question.

Example: language_tags.

2.5.6. Validation of Extracted Content

The native format can contain various reserved characters, or their sequences, for structural and inline markup, as well as for programmatic purposes. While not explicitly violating XLIFF Constraints and Processing Requirements, their incidence in the *Extracted* content could point out issues with the *Extraction* process.

One could implement a *sanity check* for the *Extractor* output that would identify potential problems by looking for such characters, or sequences of them. Failing such a sanity check would ideally interrupt the roundtrip as early as possible, allow for an update of the *Extraction* rules, and for redoing the *Extraction* in order to prevent problems further downstream. Not at least, *Extraction* would expose such control characters or their sequences to localization transformations that would most probably be harmful with regards to the *Merge* and build processes.

Example: sanity_check.

2.6. XLIFF Validations

Since XLIFF is a roundtrip oriented format that is supposed to facilitate complex workflows bringing together best of breed specialized *Agents*, it makes huge business sense to validate, to validate a lot, everywhere, and all the time. Successful validation on every input and output step is the critical factor for successful "blind", plug-and-play, or unsupervised interoperability. If you are designing a service architecture facilitated by XLIFF as the canonical data model, or if you are otherwise integrating many services that are consuming and outputting XLIFF, it makes sense to expose XLIFF validation as a reusable microservice that is freely available and even mandated at every input and output step in your ecosystem, service layer, or service bus. When your tool is supposed to receive *XLIFF Documents*, check first if the XLIFF you are trying to consume is indeed valid. Also if you are outputting *XLIFF Documents* that other *Agents* or service providers are supposed to consume, do check that you are outputting valid XLIFF. Don't force your service providers to accept invalid XLIFF, rather take their pushback as a signal that something might be wrong with your process. Do remember that problems pushed downstream will force various mitigation steps among a potentially large number of Agents

and those problems will resurface unpredictably shape-shifted no later than at the time when you will try to *Merge* the XLIFF back, or worse as the localized product's issues, if *Merging* and building miraculously succeed.

XLIFF is a format that has been blessed with multiple low level implementations, therefore you have also multiple options for XLIFF validation. Since XLIFF Version 2.1, most of the advanced validation checks that required custom code in XLIFF 2.0 can be validated using the XLIFF TC provided Schematron schemas that are in fact an integral normative part of the OASIS Standard. XLIFF TC provided artifacts (xsd, sch, and NVDL) can be used for validation in any generic XML editor. However, if you are trying to design an automated workflow you'd not typically rely on manual or semi-automated validation in XML editors, you ought rather try and build a service that is for instance using the xslt rendering of the Schematron schemas or create a service out of the command line Lynx tool that is part of the open source OAKAPI XLIFF Toolkit (java). The open source Microsoft XLIFF Object Model (.NET) contains built in validation (can be disabled). The fourth option to validate XLIFF is Bryan Schnabel's Xmarker. Advanced miscroservices architectures would ideally adopt more than one low level method of validation. This can be used either for redundancy or for double-checking of validation results, comparing of error messages for advanced trouble-shooting and so on.

3. Summary

This specification attempts (among other goals) to make the XLIFF Standards more accessible to content owners that are not necessarily looking into the full nitty-gritty of the XLIFF specs. It gives a general guidance how to handle constructs common in HTML and generic XML, it also provides some basic information on *Extraction* from Content Management Systems and software resources. The TAPICC WG3 and the Editors look forward to receiving feedback how to make this Informational Best Practice even more useful, potentially how and in what directions to expand its scope.

We haven't utterly failed if a multilingual publishing data flow designer took home what are the basic design principles behind XLIFF as the exchange bitext format, both at the structural and the inline levels. We hopefully managed to introduce and explain good business reasons for thoughtful, properly structured, and metadata rich XLIFF *Extraction* that will in turn facilitate fully automated, gold standard *Merge* and target build processes.

XLIFF *Extraction* can never make sense when perceived on its own, as an isolated process. Since XLIFF doesn't purport to standardize *Merging* without the full knowledge of the *Extraction* mechanism, all implementers that build *Extractors* will need to build *Mergers* in order to benefit from the exercise. In fact, for a corporate owner the *Extractor*/*Merger* will be considered a single application with two end points. Designers of such tools need to be acutely aware that every design compromise made at the *Extraction* endpoint will compromise the ability of downstream *Agents* to perform lossless bitext transformations and thus will in the end undesirably and sometimes unpredictably affect their own *Merging* endpoint that will have to receive the *XLIFF Documents* after a localization roundtrip.

Sometimes, some service providers do accept horrible and ugly "XLIFF" pretending that there is nothing to worry. In such cases, rest assured that the service provider had to complement the poor *Extraction*/*Merge* job with their own costly pre- and post-processing routines, or worse pushed them even further downstream onto the translators who may or may not be tech-savvy enough to preserve unprotected features that are critical for your build or runtime functionality. No matter if issues resurface in your localized product, you can rest assured that you are paying extra for solving issues that you could have solved easier on your own or you are not providing the translators with sufficient metadata to produce the best possible technical and linguistic in-context quality.

References

Normative references

[XML] W3C: Extensible Markup Language (XML) 1.026 November 2008 https://www.w3.org/TR/xml/

[XLIFF-2.1] Edited by David Filip, Tom Comerford, Soroush Saadatfar, Felix Sasaki, and Yves Savourel: XLIFF Version 2.113 February 2018 http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html-http://docs.oasis-open.org/xliff/xliff-core/v2.1/xliff-core-v2.1.html

[XLIFF-2.0] Edited by Tom Comerford, David Filip, Rodolfo M. Raya, and Yves Savourel: XLIFF Version 2.004 August 2014 http://docs.oasis-open.org/xliff/xliff-core/v2.0/os/xliff-core-v2.0-os.htmlhttp://docs.oasis-open.org/xliff/xliff-core/v2.0/xliff-core-v2.0.html

[ISO XLIFF] Edited by Tom Comerford, David Filip, Rodolfo M. Raya, and Yves Savourel: ISO 21720:2017 - XLIFF (XML Localisation interchange file format)November 2017 https://www.iso.org/standard/71490.html

[BCP 47] M. Davis: Tags for Identifying Languages, http://tools.ietf.org/html/bcp47 IETF (Internet Engineering Task Force).