JLIFF, Creating a JSON Serialization of OASIS XLIFF

Lossless exchange between asynchronous XML based and real time JSON based pipelines

David Filip, ADAPT Centre at Trinity College Dublin <david.filip@adaptcentre.ie>

Phil Ritchie, Vistatec <phil.ritchie@vistatec.com>

Robert van Engelen, Genivia <engelen@genivia.com>

Abstract

JLIFF [JLIFF] is the JSON serialization of XLIFF. Currently [JLIFF] only exists as a reasonably stable JSON schema [JLIFFSChema] that is very close to being a full bidirectional mapping to both *XLIFF 2* Versions. *XLIFF* is the XML Localization Interchange File Format. The current OASIS Standard version is *XLIFF Version 2.1* [XLIFF21]. The major new features added to [XLIFF21] compared to *XLIFF Version 2. 0* [XLIFF20] are the native *W3C ITS 2.0* [ITS20] support and the Advanced Validation feature via NVDL and Schematron. This paper describes how XLIFF was ported to JSON via an abstract object model [LIOM]. Challenges and design principles of transforming a multi-namespace business vocabulary into JSON while preserving lossless machine to machine interchange between the serializations are described in this paper. While we do explain about the *Internationalization (I18n)* and *Localization (L10n)* business specifics, we are also striving to provide general takeaways useful when porting XML based vocabularies into semantically and behaviorally interoperable JSON serializations.

Table of Contents

This research was conducted at the ADAPT Centre, Trinity College Dublin, Ireland.

The ADAPT Centre is funded under the SFI (Science Foundation Ireland) Research Centres Programme (Grant 13/RC/2106) and is co-funded under the European Regional Development Fund.

Introduction

In this paper and XML Prague presentation, we will explain about *JLIFF* [JLIFF], the JSON serialization of *XLIFF*. JLIFF is designed to start with the 2.0 version number and bidirectional mapping for both XLIFF 2.0 [XLIFF20] and XLIFF 2.1 [XLIFF21] is being built in parallel in the initial version. The design is extensible to support any future XLIFF 2.n+1 Version. The *XLIFF 2* series has been designed to be backwards and forwards compatible. XLIFFF 2.n+1 versions can

add orthogonal features such as the Advanced Validation added in [XLIFF21] or the Rendering Requirements [XLIFFRender] to be added in XLIFF 2.2. All XLIFF 2.n specifications share the core namespace `urn:oasis:names:tc:xliff:document:2.0`.

OASIS XLIFF OMOS TC only recently started developing the [JLIFF] prose specification that should be released for the 1st public review by April 2019. It is however clear that the specification largely mimics the logical structure of the XLIFF 2 specifications. JLIFF is designed to mirror XLIFF including its numerous modules, albeit via the abstract object model. The modular design of XLIFF 2 makes critical use of XML's namespaces support. Each XLIFF 2 module has elements or attributes defined in namespaces other than the core XLIFF 2 namespace. This allows the involved agents (conformance application targets) to handle all and only those areas of XLIFF data that are relevant for their area of expertize (for instance *Translation Memory* matching, *Terminology*, *Text Analysis* or entity recognition, *Size and Length Restrictions*, and so on). Now, how do you handle multi-namespace in JSON that doesn't support namespaces? This is covered in The design of JLIFF.

The data formats we are describing in this paper are for managing *Internationalization* and *Localization* payloads and metadata throughout the multilingual content lifecycle. Even though corporations and governments routinely need to present the same, equivalent, or comparable content in various languages, *multilingual content* is usually not consumed in more than one language at the same time by the same end user. Typically the target audience consumes the content in their preferred language and if everything works well they don't even need to be aware that the monolingual content they consume is part of a multilingual content repository or a result of a *Translation*, *Localization*, or cultural adaptation process.

Thus *Multilingualism* is transparent to the end user if implemented properly. To achieve end user transparency the corporations, governments, inter- or extra-national agencies need to develop and employ *Internationalization*, *Localization*, and *Translation* capabilities. While *Internationalization* is primarily done on a monolingual content or product, *Localization*, and *Translation* when done at a certain level of maturity -- as a repeatable process possibly aspiring to efficiencies of scale and automation -- requires a persistent *Bitext* format. Bitext in turn requires that *Localizable* or *Translatable* parts of the source or native format are *Extracted* into the Bitext format, which has provisions for storing the Translated or Localized target parts in an aligned way that allows for efficient and automated processing of content during the Localization roundtrip.

Our paper presented to XML Prague 2017 [3] made a detailed introduction of XLIFF ([XLIFF21prd02] as the then current predecessor of [XLIFF21] backwards compatible with [XLIFF20]) as the open standard *Bitext* format used in the Localization industry. This paper describes how the complete open transparent Bitext capability of XLIFF can be ported to JSON environments using the JLIFF format. We also demonstrate that JLIFF and XLIFF can be used interchangeably, effectively allowing to switch between XML and JSON pipelines at will.

Lay of the land

I18n and L10n Standards

The foundational Internationalization Standard is of course [Unicode] along with some related Unicode Annexes (such as [UAX9]). However, in this paper we are taking the Unicode support for granted and will be looking at the domain standards W3C ITS 2.0 [ITS20] and OASIS XLIFF [XLIFF21] along with its upcoming JSON serialization [JLIFF] that are the open standards relevant for covering the industry process areas outlined in the second part of the Introduction.

For a long time, XML has been another unchallenged foundation of the multilingual content interoperability and hence practically all Localization and Internationalization standards started as or became at some point XML vocabularies. Paramount industry wisdom is stored in data models that had been developed over decades as XML vocabularies at OASIS, W3C, LISA (RIP) and elsewhere. Although ITS is based on *abstract metadata categories*, *W3C ITS 1.0* [ITS10] had only provided a specific implementable recommendation for XML. The simple yet ingenious idea of ITS is to provide

a reusable namespace that can be injected into existing formats. Although the notion of a namespace is not confined to XML, again [ITS10] was only specifically injectable into XML vocabularies.

[ITS20] provides local and global methods for metadata storage not only in XML but also in [HTML5], it also looked at mapping into non-XML formats such as [NIF], albeit in a non-normative way. Because native HTML does not support the notion of namespaces, [ITS20] has to use attributes that are prefixed with the string `its-` for the purpose of being recognized as an HTML 5 module. In [JLIFF], we are using `its_` to indicate the ITS Module data.

[ITS20] also introduced many new metadata categories compared with [ITS10]. ITS 1.0 only looked at metadata in source content that would somehow help inform the Internationalization and Localization processes down the line. ITS 2.0 brought brand new and sometimes complex metadata categories that contain information produced during the localization processes or during the language service transformations that are necessary to produce target content and are typically facilitated by Bitext. This naturally led to a non-normative mapping of [ITS20] to [XLIFF12] and to [XLIFF20] (that was then in public reviews). Thus ITS 2.0 became a very useful extension to XLIFF. And here comes the modular design that allows to turn useful extensions into modules as part of a dot-release. Not only module data is better protected but also describing a data model addition as part of the broader spec gives an opportunity to tie lots of loose ends that are at play when using only a partially formalized mapping as an extension.

One of the main reasons why [XLIFF20] is not backwards compatible with [XLIFF12] is that the OASIS XLIFF TC and the wider stakeholder community wanted to create XLIFF 2 with a modularized data model. [XLIFF20] has a small non-negotiable core but at the same time it brings 8 namespace based modules for advanced functionality. The modular and extensible design aims at easy production of "dot" revisions or releases of the standard. *XLIFF Version 2.0* [XLIFF20] was intended as the first in the future family of backwards compatible XLIFF 2 standards that will share the maximally interoperable core (as well as successful modules surviving from 2.0). XLIFF 2 makes a distinction between modules and extensions. While module features are optional, *Conformant XLIFF Agents* are bound by an absolute prohibition to delete module based metadata (MUST NOT from [BCP14]), whereas deletion of extension based data is discouraged but not prohibited (the SHOULD NOT normative keyword is used, see [BCP14]). The *ITS Module* is the biggest feature that was requested by the industry community and approved by the TC for specification as part of [XLIFF21].

So in a nutshell, the difference between XLIFF 2.1 and XLIFF 2.0 can be explained and demonstrated as the two overlapping listings of namespaces.

Example 1. Namespaces that appear both in XLIFF 2.1 and XLIFF 2.0

```
urn:oasis:names:tc:xliff:document:2.0           <!-- Core [http://docs.oasis-o
urn:oasis:names:tc:xliff:matches:2.0            <!-- Translation Candidates Mo
urn:oasis:names:tc:xliff:glossary:2.0           <!-- Glossary Module [http://d
urn:oasis:names:tc:xliff:fs:2.0                 <!-- Format Style Module [http
urn:oasis:names:tc:xliff:metadata:2.0           <!-- Metadata Module [http://d
urn:oasis:names:tc:xliff:resourcedata:2.0       <!-- Resource Data Module [htt
urn:oasis:names:tc:xliff:sizerestriction:2.0    <!-- Size and Length Restricti
urn:oasis:names:tc:xliff:validation:2.0         <!-- Validation Module [http:/
```

Example 2. Namespaces that appear only in XLIFF 2.1

```
http://www.w3.org/2005/11/its                   <!-- ITS Module [http://docs.o
```

```
urn:oasis:names:tc:xliff:itsm:2.1                <!-- ITS Module [http://docs.oa
```

Example 3. Namespaces that appear only in XLIFF 2.0

```
urn:oasis:names:tc:xliff:changetracking:2.0      <!-- Change Tracking Module [ht
```

Apart from the 11 listed namespaces, both *XLIFF Core* and the *W3C ITS namespaces* reuse the `xml` namespace. This is still not all namespaces that you can encounter in an *XLIFF Document*. XLIFF 2 Core defines 4 element extension points (`<file>`, `<skeleton>`, `<group>`, and `<unit>`) and 4 more attribute extension points (`<xliff>`, `<note>`, `<mrk>`, and `<sm>`). Most of XLIFF's modules are also extensible by elements or by attributes. We will explain in the JLIFF design section how we dealt with the inherent multi-namespace character of XLIFF. Both module and extension data are allowed on the extension points with some notable distinctions and exceptions. Module attributes can be added not only at the above listed 4 extension points but can be also specifically allowed on `<pc>`, `<sc/>`, and `<ec/>`. Generally module namespaces based data is only considered Module (and hence absolutely protected) data when it appears on the extension points where it is explicitly listed in the prose specification which corresponds to where it is allowed by core NVDL and Schematrons. Core xsd is not capable of making this distinction.

The Notion of Extracting XLIFF Payload from Native Formats

Best practice for *Extracting* native content into [XLIFF21] has been recently specified as a deliverable of the GALA TAPICC Project in [XLIFFEMBP], see this publicly available specification for details on Extracting XLIFF. We will provide a condensed explanation of the *Extraction* concept here. The most suitable metadata category to explain the idea of Extraction from the native format with the help of ITS is *Translate*. This is simply a Boolean flag that can be used to indicate Translatability or not in source content.

Example 4. Translate expressed locally in HTML

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset=utf-8>
    <title>Translate flag test: Default</title>
  </head>
  <body>
    <p>The <span translate=no>World Wide Web Consortium</span> is
      making the World Wide Web worldwide!</p>
  </body>
</html>
```

Example 5. Translate expressed locally in XML

```
<messages its:version="2.0" xmlns:its="http://www.w3.org/2005/11/its">
  <msg num="123">Click Resume Button on Status Display or <panelmsg its:transla
      >CONTINUE</panelmsg> Button on printer panel</msg>
</messages>
```

Since it is not always practically possible to create local annotations, or the given source format or XML vocabulary has elements or attributes with clear semantics with regards to some Internationalization data categories such as Translate, in most cases, ITS 2.0 also defines a way to express a given data category globally.

Example 6. Translate expressed globally in XML

```
<its:rules version="2.0" xmlns:its="http://www.w3.org/2005/11/its">
```

```
  <its:translateRule translate="no" selector="//code"/>
</its:rules>
```

In the above the global `its:translateRule` indicates that the content of `<code>` elements is not to be translated.

XLIFF 2 Core has its own native local method how to express Translatability, it uses the `xlf:translate` attribute. Here and henceforth the prefix `xlf:` indicates this OASIS namespace `urn:oasis:names:tc:xliff:document:2.0`. Because XLIFF is the Bitext format that is used to manage the content structure during the service roundtrip in a source format agnostic way, XLIFF needs to make a hard distinction between the structural and the inline data. We know the structural vs inline distinction from many XML vocabularies and HTML. Some typical structural elements are Docbook `<section>` or `<para>` as well as HTML `<p>`. This is how XLIFF 2 will encode non-Translatability of a structural element:

Example 7. XLIFF Core @translate on a structural leaf element

```
<unit id='1' translate="yes">
        <segment>
          <source>Translatable text</source>
        </segment>
</unit>
<unit id='2' translate="no">
        <segment>
          <source>Non-translatable text</source>
        </segment>
</unit>
```

The above could be an *Extraction* of the following HTML snippet:

```
      <p translate='yes'>Translatable text</p>
       <p translate='no'>Non-translatable text</p>
```

The same snippet could be also represented like this:

Example 8. XLIFF representing ITS Translate by Extraction behavior w/o explicit metadata

```
<unit id='1'>
        <segment>
          <source>Translatable text</source>
        </segment>
 </unit>
```

However, it is quite likely that the non-translatable structural elements could provide the translators with some critical context information. Hence the non-extraction behavior can only be recommended if the *Extracting Agent* human or machine can make the call if there is or isn't some sort of contextual or linguistic relationship.

In case of the Translate metadata category being expressed inline, XLIFF has to use its *Translate Annotation*:

Example 9. XLIFF Core @translate used inline

```
<unit id='1'>
       <segment>
          <source>Text <pc id='1'/><mrk id='m1' translate='no'>Code</mrk></pc
```

```
        </segment>
</unit>
```

The above could be an Extraction of the following HTML snippet:

```
    <p>Text <code translate='no'>Code</code></p>
```

Also inline, there is an option to "hide" the non-translatable content like this:

Example 10. XLIFF representing ITS Translate by Extraction behavior w/o explicit metadata

```
<unit id='1'>
     <segment>
         <source>Text <ph id='1'/></source>
     </segment>
 </unit>
```

Again not displaying of the non-translatable content can be detrimental to the process, as both human and machine translation agents would produce unsuitable translations in case there is some linguistic relationship between the displayed translatable text and the content hidden by the placeholder code.

Because XLIFF has its own native method of expressing translatability, generic ITS decorators could not succeed. ITS processors can however access the translatability information within XLIFF using the following global rule:

Example 11. ITS global rule to detect translatability in XLIFF

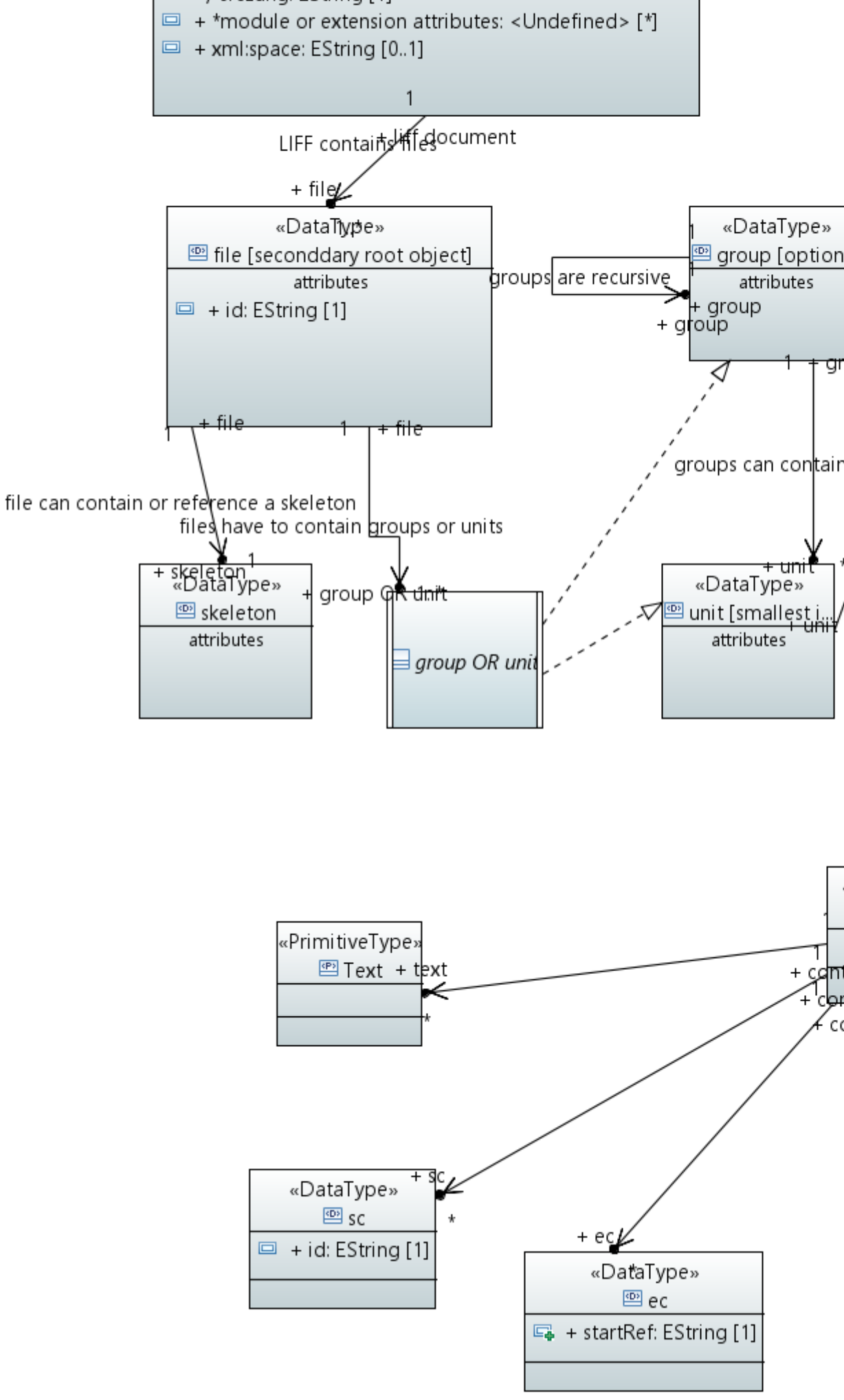
```
<its:rules version="2.0" queryLanguage="xpath">
      <!-- Rules for Translate -->
      <its:translateRule selector="//xlf:*[@translate='no']" translate='no'/>
      <its:translateRule selector="//xlf:*[@translate='yes']" translate='yes'
 </its:rules>
```

The above rule will correctly identify all XLIFF nodes that are using the `xlf:translate` attribute with one important caveat, Translatability annotations on pseudo-spans will be interpreted as empty <sm/> nodes. And pseudo-span Translatability overlaps with Translatability well-formed markup will not be properly interpreted, see [XLIFF21] http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html#translateAnnotation.

The abstract Localization Interchange Object Model (LIOM)

The Core Structure

This figure [LIOMFig] is a UML class diagram rendering of the abstract object model behind XLIFF 2 Core and hence also JLIFF core.


+ *module or extension attributes: <Undefined> [*]
+ xml:space: EString [0..1]
1
LIFF contains files
+ liff document
+ file
«DataType»
file [seconddary root object]
attributes
+ id: EString [1]
groups are recursive
«DataType»
group [option
attributes
+ group
+ group
+ file
+ file
groups can contain
file can contain or reference a skeleton
files have to contain groups or units
+ skeleton
«DataType»
skeleton
attributes
+ group OR unit
group OR unit
+ unit
«DataType»
unit [smallest i...
attributes
«PrimitiveType»
Text
+ text
+ sc
«DataType»
sc
+ id: EString [1]
+ ec
«DataType»
ec
+ startRef: EString [1]

The above can be described in a natural language as follows:

A *LIOM* instance contains at least one file. Each file can contain an optional recursive group structure of an arbitrary depth. Because grouping is fully optional, files can contain only a flat series of units. But also any file or group can contain either a flat structure of units or an array of groups and units (subgroups). Each unit contains at least one sub-unit of the type segment (rather than ignorable). Each sub-unit contains exactly one source and at most one target. Bitext is designed to represent the localizable source structure and only later in the process is expected to be Enriched with aligned target content. The content data type can contain character data mixed with inline elements. It is worth noting that XLIFF even in its XML serialization only preserves a treelike document object model (DOM) down to unit. Inline markup present in the source and target content data can form spans that can and often have to overlap the tree structure given by the well-formed `<mrk>` and `<pc>`, but also notably the structural `<source>`, `<target>`, `<segment>`, and `<ignorable>` tags. The `<unit>` tag separates the upper clean XML well-formed structure from the transient structure of segments where "non-well-formed" pseudo-spans formed by related empty tags (`<sc id="1"/>` and `<ec startRef="1"/>`, as well as `<sm id="2"/>` and `<em startRef="2"/>` pairs) need to be recognized and processed by *XLIFF Agents*. See [XLIFF21] Spanning Code Usage [http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html#spanningcodeusage] ("*Agents* MUST be able to handle any of the above two types of inline code representation") or Pseudo-span Warning [http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html#pseudo-spanWarning]. Since the equivalence of well-formed versions of the spanning codes `<pc>` and markers `<mrk>` with the above pseudo-spans is defined and described in XLIFF 2 itself, there is no need to include the well-formed versions of the inline tags in the abstract LIOM and non-XML serializations including JLIFF are free to use a fully linear inline data model.

The above class diagram shows that any LIOM instance has four options of logical roots. The original XML serialization, i.e. XLIFF 2, can only use the top level root object according to its own grammar. On the other hand, the abstract object model caters for use cases where LIOM fragments could be exchanged. Such scenarios include real time unit exchange between translation tools such as between a *Translation Management System (TMS)* and a translation or review workbench (browser based or standalone), a TMS and a *Machine Translation (MT)* engine, two different TMSes, and so on.

Based on the above, a LIOM instance can represent a number of *source files*, a single source file, a structural subgroup (at any level of recursion) or a smallest self contained logical unit that are intended for Localization.

The top level wrapper in the XML serialization is the `<xliff>` element, the top level object in the JSON serialization is an anonymous top level object with the required `jliff` property.

Example 12. XLIFF top level element

```
<xliff xmlns="urn:oasis:names:tc:xliff:document:2.0"
xmlns:uext1="http://example.com/userextension/1.0"
xmlns:uext2="http://example.com/userextension/2.0"
version="2.1" srcLang="en" trgLang="fr">
  <file … >
    <group … >
      /arbitrary group depth including 0/
      <unit … >
      [ … /truncated payload structure / … ]
      </unit>
    </group>
  <file>
</xliff>
```

Example 13. JLIFF anonymous top level object

```
{
    "jliff": "2.1",
```

```
    "@context": {
      "uext1": "http://example.com/userextension/1.0",
      "uext2": "http://example.com/userextension/2.0"
    },
    "srcLang": "en",
    "trgLang": "fr",
    "files | subfiles | sugbroups | subunits": [ … /truncated payload structure
}
```

Comparing the two examples above, it is clear that XLIFF in its original XML serialization doesn't have another legal option but to represent the whole project structure of source files. JLIFF has been conceived from the beginning as the JSON Localization *Fragment* Format, so that top level JLIFF object (`jliff`) can wrap an array of files (within the `files` object), an array of groups or units (within the `subfiles` or the `subgroups` object), or an array of sub-units (within the `subunits` object). Since the data model of a subfile and a subgroup is identical, `subfiles` and `subgroups` are instances of a common JSON schema type named `subitems`. The `subitem` type object simply holds an array of anonymous group or unit objects.

The `jliff` property values are restricted at the moment to `2.1` or `2.0`. The context property is optional as it is only required to specify [JSON-LD] context for extensions if present. This is a workaround to declaring the same extensions' namespaces as in the XLIFF top level example. The `srcLang` property is required while the `trgLang` property is only required when `target` objects are present.

LIOM Modules

LIOM modules originating in XLIFF 2.0

[XLIFF20] defined 8 namespace based modules, from which 7 survived to [XLIFF21], see the namespaces listing above. We won't be dealing with the deprecated Change Tracking Module.

The simplest [XLIFF21] Module is the Format Style Module [http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html#fs-mod] that consists just of two attributes `fs` and `subFs`. Obviously this very easy to express both in the abstract LIOM and in the JLIFF serialization. The `subFs` property is only allowed on objects where `fs` has been specified. At the same time `fs` values are restricted to a subset (see [XLIFF21] http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html#d0e13131) of [HTML5] tag names. While the Format Style Module only provides 2 properties, it is rather far reaching as these are allowed on most structural and inline elements of XLIFF Core. Information provided through `fs` and `subFs` is intended to allow for simple transformations to create previews. Previews are extremely important to provide context for human translators.

Apart from Format Style, all other Modules define their own elements. In general each Module's top wrapper is explicitly allowed on certain static structure elements and has a reference mechanism to point to a core content portion that it applies to. Glossary Module data can be also pointed to *vice versa* from the Core Term Annotation [http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html#termAnnotation].

The [XLIFF21] Translation Candidates Module [http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html#candidates] is only allowed on the unit level and serves for storing of locally relevant Translation suggestions. Those typically come from a TM or MT service. The Translation Candidate module reuses the core source and target data model, as the data is designed to be compared with the core contained source content and populate the core target content parts from the module target containers. While this primarily targets human translators selecting suitable translation candidates. Most TMSes have a function to populate or pre-populate core target containers with the module based suggestions based on some decision making algorithms driven by the match metada caried within the module. Those include properties such as `similarity`, `matchQuality`, `matchSuitability`, `type` of the candidate, but the only required property is a pointer identifying the relevant source content span to which the translation suggestion applies. Apart from reusing core, the module is extensible by the Metadata Module and by custom extensions.

The [XLIFF21] Glossary Module [http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html#glossary-module] is designed to provide locally relevant Terminology matches. But can be also used *vice versa* to store terminology identified by human or machine agents during the roundtrip. A mapping of XLIFF Core + Glossary Module to and form TBX Basic has been defined in [XLIFFglsTBXBasic]. This mapping is now being standardized also within OASIS XLIFF OMOS TC (the home of LIOM and JLIFF).

The [XLIFF21] Metadata Module [http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html#metadata_module] is perhaps the most suitable for being ported to JSON and other non-markup-language serialization methods. While being a module that in fact strictly protects it's content, it's also an extensibility mechanism for implementers who don't want to extend via their own namespace based extensions. The metadata structure is optionally recursive and is allowed on all XLIFF Core static structural elements (`file`, `group`, `unit`). It doesn't specify a referencing mechanism. It is simply an optionally hierarchically structured and locally stored set of key-value pairs to hold custom metadata. Because the data strusture is restricted to key-value pairs it provides at least some limited interoperability for the custom data as all implementers should be capable of displaying structural elements related key-value pairs data. Each Metadata Module object (element) has an optional id property (attribute) that allows for addressing from outside the module, either globally or within the LIOM instance.

The [XLIFF21] Resource Data Module is designed to provide native resource data either as context for the Translatable payload or as non-text based resource data to be modified along with the Translatable payload (if the resource data model is known and supported by the receiving agent), for instance GUI resources to be resized to fit the Translated UI strings. Resource data can be considered locally specific skeleton data and would be typically binary or generally of any media type. To tell receiving agents what type of data the module holds, there is a media type property (`mimeType` attribute).

The [XLIFF21] Size and Length Restriction Module [http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html#size_restriction_module] is a powerful mechanism that allows for defining any sort of size or storage constraints, even multidimensional. IT specifies a standard code point based restriction profile as well as three standard storage size restriction profiles. It also gives guidance how to specify arbitrary size or shape restriction profiles, for instance to control fitting restrictions in complex desktop publishing environments, custom embedded displays with font limitations, and so on.

The [XLIFF21] Validation Module [http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html#validation_module] provides an end user friendly way to specify simple Quality Assurance rules that target strings need to fulfill, mainly in relation to source strings. For instance a substring that appears in source must or must not appear in the target string. For instance, a brand name must appear in both source and target. Or on the contrary, a brand name must not be used in a specific locale for legal reasons.

The ITS Module

The [XLIFF21] ITS Module [http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html#ITS-module] specification is comparable in size with all the other Module specifications taken together. It defines native support or mapping, Extraction or ITS Rules parsing guidance for all 19 W3C ITS 2.0 [ITS20] metadata categories and for its [ITS20] ITS Tools Annotation [https://www.w3.org/TR/its20/#its-tool-annotation] mechanism.

Language Information [http://www.w3.org/TR/its20/#language-information] uses the [BCP47] data model via `xml:lang` to indicate the natural language of content. This is obviously very useful in case you want to source translations or even just render the content with proper locale specifics. This partially overlaps with XLIFF's own usage of `xml:lang` to specify `srcLang` and `trgLang`. An `urn:oasis:names:tc:xliff:itsm:2.1` namespace based attribute `itsm:lang` is provided to specify third language material inline. Both xsd and JSON Schema have an issue in validating [BCP47] tags. A regex or custom code based solution is recommended.

Directionality [http://www.w3.org/TR/its20/#directionality] has quite a profound Internationalization impact, it let's renderers decide at the protocol level (as opposed to the plain text or script level) whether the content is to be displayed left to right (LTR - Latin script default) or right to left (RTL

- Arabic or Hebrew script default). But the Unicode Bidirectional Algorithm [UAX9]as well as directionality provisions in HTML and many XML vocabularies changed since 2012/2013, so the ITS 2.0 specification text is actually not very helpful here. This obviously doesn't affect the importance of the abstract data category and of having proper display behavior for bidirectional content. LIOM contains core `srcDir` and trgDir and `dir` properties that allow values `ltr`, `rtl`, and `auto`. The default auto determines directionality heuristically as specified in [UAX9]. Directionality in XLIFF is given by a high level protocol in the sense of [UAX9]. All objects that can have the directionality property in LIOM either determine the directionality of their descendants (as higher level protocol) or act as directionality isolators inline.

Preserve Space [http://www.w3.org/TR/its20/#preservespace] indicates via `xml:space` whether or not whitespace characters are significant. If whitespace is significant in source content it is usually significant also in the target content, this is more often then not an internal property of the content format, but it's important to keep this characteristics through transformation pipelines. The danger that this category is trying to prevent is the loss of significant whitespace characters that could not be recovered. This data category is considered XMLism or SGMLism. It should be preserved at LIOM level. However, even XLIFF21 recommends pre-Extaction normalization of whitespace and setting of all inline content whitespace behavior to `preserve`. This is also the option best interoperable with JSON where all payload whitespace is significant.

ID Value [http://www.w3.org/TR/its20/#idvalue] indicates via `xml:id` a globally unique identifier that should be preserved during translation and localization transformations mainly for the purposes of reimport of target content to all the right places in the native environment. XLIFF id mechanism is NMTOKEN rather than NCName based. However, usage of `xml:id` to encode this metadata category can only be exprected in XML source environments. Therefore, XLIFF and LIOM use an unrestricted string mechanism (`original` on file, `name` on group and unit) to roundtrip native IDs.

Terminology [http://www.w3.org/TR/its20/#terminology] can simply indicate words or multi-word expressions as terms or non-terms. This is how the category worked in [ITS10]. In [ITS20], Terminology can be more useful by pointing to definitions or indicating a confidence score, which is especially useful in cases the Terminology entry was seeded automatically. Terminology belong exclusively to categories that come from the native format. Together with Text Analysis it can be actually injected into the content during any stage of the lifecycle or roundtrip and is not limited to source. However, it is very important for the localization process, human or machine driven, to have Terminology annotated be it even only the simple Boolean flag. Core [LIOM] doesn't have the capability to say that a content span is not a term, therefore the negative annotation capability is provided via the ITS Module.

Text Analysis [http://www.w3.org/TR/its20/#textanalysis] is a sister category to Terminology that is new in [ITS20]. It is intended to hold mostly automatically sourced (possibly semi-supervised) entity disambiguation information. This can be useful for translators and reviewers but can also enrich reading experience in purely monolingual settings. This is fully defined in the ITS Module as there is no equivalent in core [LIOM] or [XLIFF21].

Domain [https://www.w3.org/TR/its20/#domain] can be used to indicate content topic, specialization or subject matter focus that is required to produce certain translations. This can be for instance used to select a suitably specialized MT engine, such as one trained on an automotive bilingual corpus in case an automotive domain is indicated or. In another use case, a language service provider will use a sworn translator and require in country legal subject matter review in case the domain was indicated as legal. Although ITS data categories are defined independently and don't have implementation dependencies, Domain information is well suited for usage together with the Terminology and Text Analysis datacats. As the Domain [https://www.w3.org/TR/its20/#domain] datacat doesn't have local markup in the W3C [ITS20] namespace, [XLIFF21] had to define a local `itsm:domain` attribute that is also taken over as a local property by [LIOM] and [JLIFFSchema].

MT Confidence [http://www.w3.org/TR/its20/#mtconfidence], Localization Quality Issue [http://www.w3.org/TR/its20/#lqissue], Localization Quality Rating [http://www.w3.org/TR/its20/#lqrating], and Provenance [http://www.w3.org/TR/its20/#provenance] - all new categories in ITS 2.0 - can be only produced during Localization transformations; specifically, during Machine Translation,

during a review or Quality Assurance process, during or immediately after a manual or automated Translation or revision.

MT Confidence [http://www.w3.org/TR/its20/#mtconfidence] gives a simple score between 0 and 1 that encodes the automated translation system's internal confidence that the produced translation is correct. This score isn't interoperable but can be used in single engine scenarios for instance to color code the translations for readers or post-editors. It can also be used for storing the data for several engines and running comparative studies to make the score interoperable first in specific environments and later on maybe generally. This overlaps with the LIOM Translation Module's property `matchQuality`.

Localization Quality Issue [http://www.w3.org/TR/its20/#lqissue] contains a taxonomy of possible Translation and Localization errors that can be applied in annotations of arbitrary content spans. The taxonomy ensures that this information can be exchanged among various Localization roundtrip agents. Although this mark up is typically introduced in a Bitext environment on target spans, marking up source isn't exclude and can be very practical, especially when implementing the feedback or even reporting a source issue. Importantly, the issues and their descriptions can be Extracted into target content and consumed by monolingual reviewers in the native environment. This is fully defined in the ITS Module as there is no equivalent in core [LIOM] or [XLIFF21].

Localization Quality Rating [http://www.w3.org/TR/its20/#lqrating] is again a simple score that gives a percentage indicating the quality of any portion of content. This score is obviously only interoperable within an indicated Localization Quality Rating system or metrics. Typically flawless quality is considered 100 % and various issue rates per translated volume would strike down percentages, possibly dropping under an acceptance threshold that can be also specified. This is fully defined in the ITS Module as there is no equivalent in core [LIOM] or [XLIFF21].

Provenance [http://www.w3.org/TR/its20/#provenance] in ITS is strictly specialized to indicate only translation and revision agents. Agents can be organizations, people or tools or described by combinations of those. For instance, Provenance can indicate that the Reviser John Doe from ACME Language Quality Assurance Inc. produced a content revision with the Perfect Cloud Revision Tool. This is fully defined in the ITS Module as there is no equivalent in core [LIOM] or [XLIFF21].

In spite of [XLIFF21] using the W3C namespace for the ITS Module, there is a systematic scope mismatch between the XLIFF defined ITS attributes and the ITS defined XML attributes. Because [ITS20] has no provision to parse pseudo-spans, it will necessarily fail to identify spans formed by XLIFF Core `<sm/>` and `<em/>` markers.

In XLIFF, Modifiers can always transform `<mrk id="1">span of text</mrk>` into `<sm id="1"/>span of text <em startRef="1"/>`, which is fundamentally inaccessible by ITS Processors (or other generic XML tooling) without extended provisions. Unmodified or unextended ITS Rules will find the `<sm/>` nodes, if those nodes do hold the W3C ITS namespace based attributes or native XLIFF attributes that can be globally pointed to by ITS rules, yet they will fail to identify the pseudo-spans and will consider the `<sm/>` nodes empty, ultimately failing to identify the proper scope of the correctly identified datacat. XLIFF implementers who want to make their XLIFF Stores maximally accessible to ITS processors are encouraged to avoid forming of `<sm/>` based spans, it is however often not possible. Had it been possible, XLIFF would have not needed to define `<sm/>` and `<em/>` delimited pseudo-spans in the first place.

Principal reasons to form pseudo-spans include the following requirements: 1) capability to represent non-XML content, 2) need for overlapping annotations, 3) capability to represent annotations overlapping with formatting spans as well as 4) annotations broken by segmentation (which has to be represented as well formed structural albeit transient nodes).

The design of JLIFF

The design of JLIFF follows the design of XLIFF 2 closely albeit making use of abstractions described under LIOM Core. As with XLIFF 2, the JLIFF Core corresponds to the abstract Localization Interchange Object Model, LIOM. One of the primary goals of JLIFF is compatibility with XLIFF 2

to allow switching between XML and JSON based pipeline at will as stated in the Introduction. While JLIFF is structurally different compared to XLIFF 2 due to the much simpler JSON representation format, compatibility is made possible through the following mappings:

1. As a general rule, JSON object property names are used to represent XLIFF elements and attributes, with the exception of element sequences that must be represented by JSON arrays. JSON object properties should be unique and are unordered, whereas this does not generally hold for XML elements;

2. It was decided to use JSON arrays to represent element sequences, for example a sequence of `<file>` elements becomes an array identified by the JSON object property `"files": […]` where each array item is an anonymous file object that contains an array of `"subfiles": […]`. It was decided to use plural forms to refer to arrays in JLIFF and as a reminder of the structural differences between XML and JSON;

3. To store units and groups that exist within files, JSON object property `"subfiles": […]` is an array of unit and group objects representing XLIFF `<unit>` and `<group>` elements, where an anonymous unit object is identified by a `"kind": "unit"` and an anonymous group object is identified by `"kind": "group"`;

4. Likewise, `"subunits": […]` is an array of subunits of a unit object, where a segment subunit is identified as an object with `"kind": "segment"` and a ignorable object is identified as `"kind": "ignorable"`;

5. A subset of XSD data types that are used in XLIFF are also adopted in the JLIFF schema by defining corresponding JSON schema string types with restricted value spaces defined by regex patterns for `NCName`, `NMTOKEN`, `NMTOKENS`, and `URI/IRI`. The latter is not restricted by a regex pattern due to the lax validation of URI and IRI values by processors;

6. Because JSON intrinsically lacks namespace support, it was decided to use qualified JSON object property names to represent XLIFF modules, which is purely syntactic to enhance JLIFF document readability and processing. For example, ITS module properties are identified by prefix `its_`, such as `"its_locQualityIssues"`. Generally undersore "_" is used as the namespace prefix separator for modules (unlike custom namespace based extentions);

7. JLIFF extensions are defined by the optional JSON-LD context `"@context": {…}` as a property of the anonymous JLIFF root object. [JSON-LD] offers a suitable replacement of XML namespaces required for extension identification and processing. A JSON-LD context is a mapping of prefixes to IRIs. A JSON-LD processor resolves the prefix in an object property name and thus creates a fully qualified name containing the corresponding IRI qualifier;

8. To identify JLIFF documents, the anonymous JLIFF root object has a required property `"jliff": "2.0"` or `"jliff": "2.1"`;

9. One of the decisions taken in relation to element mappings was not to explicitly support well-formed `<pc/>` and `<mrk/>` elements, therefore `<mrk/>` is mapped to `<sm/>` and `<em/>` pairs, and `<pc/>` is mapped to `<sc/>` and `<ec/>` pairs. See also LIOM Core.

These mapping decisions were verified against the LIOM and XLIFF 2 while developing the JSON schema for JLIFF. In addition, to validate the approach several XLIFF examples were translated to JLIFF and validated by the JLIFF JSON schema. A reference implementation is being developed for lossless translation of XLIFF into JLIFF and back, except for support for `<pc/>` and `<mrk/>` elements as explained above.

Further design considerations worth noting include:

1. While JSON supports the Boolean type values `true` and `false`, it was decided to use string based enumerations of `yes` and `no` in JLIFF to represent XLIFF attributes of the `yesNo` type. There are two main reasons for this. Firstly, the omission of a Boolean value is usually associated with the value false by processors and applications. By contrast, the absence of a value should not default to false in JLIFF. The absence of a value has an XLIFF driven meaning. In fact the XLIFF default of the `yesNo` attributes is `yes`. Absence of of an attribute typically indicates permission.

Hence we defined the yes defaults in the JLIFF JSON schema, which would have conflicted with the defaulting behavior of the JSON Boolean type. Secondly, the object property `canReorder` of the `ec`, `ph`, and `sc` objects is a three-valued enumeration with the `yes`, `no`, and `firstNo` values, necessitating the use of a JSON string type with enumeration rather than a JSON Boolean type in JLIFF.

2. Almost all JSON schema types defined for JLIFF correspond one-to-one with JSON object property names in JLIFF documents. This design choice reduces efforts to comprehend the JLIFF JSON schema structure for implementers versed in XLIFF. For example, the `files` property mentioned earlier has a corresponding `files` type in the JLIFF JSON schema, which is an array that references the `file` schema type. However, this schema design deviates in one important aspect that is intended to avoid unnecessary duplication of the schema types for the properties `subfiles` and `subgroups` that share the same data model. It was decided to introduce the schema type `subitems` to represent the value space of both `subfiles` and `subgroups`. We also added named types to the schema that have no corresponding property name, to break out the JSON structure more clearly. For example, `elements` is an array of mixed types, which is `one of` (`element-text`, `element-ph`, `element-sc`, `element-ec`, `element-sm`, and `element-em` in the schema. Note that `element-text` is a string while the other types are objects.

3. JLIFF considers LIOM Modules an integral part of the JLIFF specification, meaning that all Modules are part of the single JSON schema specification of JLIFF [JLIFFSchema]. The decision to make Modules part of the JLIFF specification is a practical one that simplifies the processing of Modules by processors, as Modules are frequently used (albeit different subsets based on individual needs and specializations) by implementers. By contrast, extensions are registered externally and included in JLIFF documents as `userdata` objects. A `userdata` object contains one or more extensions as key-value pairs: each extension is identified by a qualified property with an extension-specific JSON value. The prefix of the qualified property of an extension is bound to the IRI of the extension using the JSON-LD `@context` to identify the unique extension namespace IRI. Processors that are required to handle extensions should resolve the prefix to the extension's objects fully qualified names as per JSON-LD processing requirements. Otherwise extensions can be ignored without raising validation failures. This approach offers an extensible and flexible mechanism for JLIFF extensions. While the JSON-LD workaround for namespaces was considered a suitable solution for extensions, it was considered heavy weight and to complex for modules that are used regularly. The "context" of modules is considered a shared JLIFF agent knowledge documented in the prose specification rather then being resolved each time when module data need processed, hammering OASIS servers that would need to hold the canonical context files required for proper JSON-LD processing.

Reference Implementation

It was an early goal to work on a concrete implementation of JLIFF in parallel to the development of the schema. It would give us an early opportunity to find and work through any design flaws or limitations. Fortunately, the JliffGraphTools [JGT] reference implementation has been open source since early on.

It was a wish that JLIFF should be easy to implement and serialize/deserialize using well-known JSON libraries.

As explained in the JLIFF Design section, there is a difference in the structure of JLIFF and XLIFF in inline markup. In XLIFF, inline markup is nested within the segment text as descendant elements of the `<source>` and `<target>` elements. In JLIFF the segment text and inline markup are stored as an array of objects property of the unit's source and target properties. This has an impact on how rendering tools may display strings for translation, see below on the approach taken in [JGT].

Having got the priority of JSON serialization and deserialization working we then looked at roundtripping, i.e. the capability to create an XLIFF output from a JLIFF file legally changed by a Translation agent. JliffGraphTools [JGT] supports bidirectional serialization between XLIFF and JLIFF and it is this library which powers the [Xliff2JliffWeb] web application made public at http://

xliff2jliff.azurewebsites.net/. Unfortunately the public web application only implements the JLIFF output capability at the time of writing this.

At present when segments for translation are rendered in [JGT], there is an option to flatten the array of text and inline markup objects and render them in a way which is based upon the approach taken in the [OkapiFwk] XLIFF library. That is inline markup tags are converted to coded text which uses characters from the private use area of [Unicode] to delimit inline markup tags. See [OkapiFwk] http://okapiframework.org/devguide/gettingstarted.html#textUnits.

The following program listings demonstrate [JGT] can be used to exchange the flattened fragments instead of the fully equivalent JLIFF. This capability is based on JLIFF having been designed to support any of the four logically possible LIOM roots, see LIOM Core Structure.

Example 14. Simple XLIFF input

```
<?xml version="1.0"?>
<xliff xmlns="urn:oasis:names:tc:xliff:document:2.0" version="2.1" srcLang="en"
<file id="f1">
 <unit id="u1">
  <originalData>
   <data id="d1">[C1/]</data>
   <data id="d2">[C2]</data>
   <data id="d3">[/C2]</data>
  </originalData>
  <segment canResegment="no" state="translated">
   <source><ph id="c1" dataRef="d1"/> aaa <pc id="c2" dataRefEnd="d3" dataRefSt
   <target><ph id="c1" dataRef="d1"/> AAA <pc id="c2" dataRefEnd="d3" dataRefSt
  </segment>
  <ignorable>
   <source>.  </source>
  </ignorable>
 </unit>
</file>
</xliff>
```

Example 15. [XLiff2JliffWeb] -> Fully equivalent JLIFF

```
{
  "jliff": "2.1",
  "srcLang": "en-US",
  "trgLang": "fr-FR",
  "files": [
    {
      "id": "f1",
      "kind": "file",
      "subfiles": [
        {
          "canResegment": "no",
          "id": "u1",
          "kind": "unit",
          "subunits": [
            {
              "canResegment": "no",
              "kind": "segment",
              "source": [
                {
                  "dataRef": "d1",
                  "id": "c1",
                  "kind": "ph"
```

```
                },
                {
                  "text": " aaa "
                },
                {
                  "dataRef": "d3",
                  "id": "c2",
                  "kind": "ec"
                },
                {
                  "text": "text"
                },
                {
                  "kind": "ec"
                }
              ],
              "target": [
                {
                  "dataRef": "d1",
                  "id": "c1",
                  "kind": "ph"
                },
                {
                  "text": " AAA "
                },
                {
                  "dataRef": "d3",
                  "id": "c2",
                  "kind": "ec"
                },
                {
                  "text": "TEXT"
                },
                {
                  "kind": "ec"
                }
              ]
            },
            {
              "kind": "ignorable",
              "source": [
                {
                  "text": ".  "
                }
              ],
              "target": []
            }
          ]
        }
      ]
    }
  ]
}
```

Example 16. [XLiff2JliffWeb] -> "Flattened" JLIFF

```
{
  "jliff": "2.1",
```

```
  "srcLang": "en",
  "trgLang": "fr",
  "subunits": [
    {
      "canResegment": "no",
      "kind": "segment",
      "source": [
        {
          "dataRef": "d1",
          "id": "c1",
          "kind": "ph"
        },
        {
          "text": " aaa "
        },
        {
          "dataRef": "d3",
          "id": "c2",
          "kind": "ec"
        },
        {
          "text": "text"
        },
        {
          "kind": "ec"
        }
      ],
      "target": [
        {
          "dataRef": "d1",
          "id": "c1",
          "kind": "ph"
        },
        {
          "text": " AAA "
        },
        {
          "dataRef": "d3",
          "id": "c2",
          "kind": "ec"
        },
        {
          "text": "TEXT"
        },
        {
          "kind": "ec"
        }
      ]
    }
  ]
}
```


Discussion and Conclusions

In the above we tried to show how we ported into JSON XLIFF 2, a complex business vocabulary from the area of Translation and Localization. While the paper deals in detail only with XLIFF and

JLIFF, we believe that important topics were covered that will be useful for designer who will endeavor porting their own specialized multi-namespace business vocabularies into JSON.

The major takeway we'd like to suggest is not to port directly from XML to JSON. It is worth the time to start your exercise with expressing your XML data model in an abstract way, we used UML class diagram as the serialization independent abstract method.

XML serializations are as a rule fraught with "XMLism" or "SGMLism". While some of the XML capabilities such as the multi-namespace support are clear XML advantages that will force the JSON-equivalent-designer into complex and more or less elegant workarounds and compromises. Some other XML traits and constraints are arbitrary from the point of view of other languages and serialization methods.

To name just a few examples of XMLism that doesn't need maintained in JSON. You don't need to support well formed versions of inline markup in JSON, it is easier to serialize everything linearly. In JSON, all payload space is significant, so you don't need to keep the `preserve` | `default` flag in your JSON serialization. Instead make sure that all inline data is normalized and set to `preserve` in your XML data. JSON data types are much poorer than XML datatyes, nevertheless, you can make up for this with relative ease with the usage of regular expression patterns in your JSON schema. For instance

Example 17. NCName pattern in JSON schema

```
"NCName": {
      "description": "XSD NCName type for xml:id interoperability",
      "type": "string",
      "pattern": "^[_A-Za-z][-._A-Za-z0-9]*$"
    }
```

Namespaces support workarounds in JSON are worth an extra mention. While JSON doesn't support namespaces *per se*. We identified the JSON-LD methods for introducing and shortening fully qualified names quite useful as a namespaces support surrogate. For practical reasons (like prevention of hammering of OASIS servers to read XLIFF module context files) we decided to use the full blown JSON-LD method for expanding prefixes into fully qualified names only for extensions. We decided to use an arbitrary "_" (underscore) prefix separator to make the XLIFF modules human discernable. There goes the disadvantage of losing the modularity of XLIFF modules in JLIFF, yet we felt that JSON-LD-coding of each of the modules data would be very complex and heavyweight with minor benefits to outweigh the drawbacks.

Bibliography

[1] S. Saadatfar and D. Filip, Advanced Validation Techniques for XLIFF 2. in Localisation Focus, vol. 14, no. 1, pp. 43-50, April 2015. http://www.localisation.ie/locfocus/issues/14/1

[2] S. Saadatfar and D. Filip, Best Practice for DSDL-based Validation. in XML London 2016 Conference Proceedings, May 2016. https://xmllondon.com/2016/xmllondon-2016-proceedings.pdf#page=64

[3] D. Filip, W3C ITS 2.0 in OASIS XLIFF 2.1, in XML Prague 2017 - Conference Proceedings, Prague, 2017, vol. 2017, pp. 55–71. http://archive.xmlprague.cz/2017/files/xmlprague-2017-proceedings.pdf#page=67

[BCP14] S. Bradner and B. Leiba, Eds., Key words for use in RFCs to Indicate Requirement Levels and Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words, IETF (Internet Engineering Task Force) 1997 & 2017. http://tools.ietf.org/html/bcp14

[BCP47] M. Davis, Ed., Tags for Identifying Languages, IETF (Internet Engineering Task Force) http://tools.ietf.org/html/bcp47 .

[HTML5] S. Faulkner at al. Eds., HTML 5.2 W3C Recommendation 14 Dec 2017. https://www.w3.org/TR/html52/

[ITS10] C. Lieske and F. Sasaki, Eds.: Internationalization Tag Set (ITS) Version 1.0. W3C Recommendation, 03 April 2007. W3C. https://www.w3.org/TR/its/

[ITS20] D. Filip, S. McCance, D. Lewis, C. Lieske, A. Lommel, J. Kosek, F. Sasaki, Y. Savourel, Eds.: Internationalization Tag Set (ITS) Version 2.0. W3C Recommendation, 29 October 2013. W3C. http://www.w3.org/TR/its20/

[JGT] P. Ritchie, JLIFF Graph Tools. Vistatec, 2019. https://github.com/vistatec/JliffGraphTools/commit/74ffde990d8dd6d6d5d3f80d78e76ea8b0dc8736

[JLIFF] D. Filip and R. van Engelen, *JLIFF Version 1.0 [wd01]*. OASIS, 2018. https://github.com/oasis-tcs/xliff-omos-jliff/commit/7e63e0d766bb7394f9dccaa93d7fa54bf1a394d3

[JLIFFSchema] R. van Engelen, *JLIFF Version 1.0, JSON Schema [wd01]*. OASIS, 2018. https://github.com/oasis-tcs/xliff-omos-jliff/commit/2ed3b57f38548600f1261995c466499ad0ade224/>

[JSON-LD] M. Sporny, G. Kellogg, M. Lanthaler, Eds. JSON-LD 1.0, A JSON-based Serialization for Linked Data W3C Recommendation 16 January 2014. https://www.w3.org/TR/2014/REC-json-ld-20140116//>

[L10nStandards] D. Filip: Localization Standards Reader 4.0 [v4.0.1], Multilingual, vol. 30, no. 1, pp. 59–73, Jan/Feb-2019. https://magazine.multilingual.com/issue/jan-feb-2019dm/localization-standards-reader-4-0/

[LIOM] D. Filip, *XLIFF 2 Object Model Version 1.0 [wd01]*. OASIS, 2018. https://github.com/oasis-tcs/xliff-omos-om/commit/030828c327998e7c305d9be48d7dbe49c8ddf202/>

[NIF] S. Hellmann, J. Lehmann, S. Auer, and M. Brümmer: Integrating NLP using Linked Data. 12th International Semantic Web Conference, Sydney, Australia, 2013. http://svn.aksw.org/papers/2013/ISWC_NIF/public.pdf

[OkapiFwk] Y. Savourel et al., Okapi Framework. Stable release M36, Okapi Framework contributors, August 2018. http://okapiframework.org/

[UAX9] M. Davis, A. Lanin, and A. Glass, Eds.: UAX #9: Unicode Bidirectional Algorithm.. Version: Unicode 11.0.0, Revision 39, 09 May 2018. Unicode Consortium. http://www.unicode.org/reports/tr9/tr9-39.html

[Unicode] K. Whistler et al., Eds.: The Unicode Standard. Version 11.0 - Core Specification, 05 June 2018. Unicode Consortium. https://www.unicode.org/versions/Unicode11.0.0/UnicodeStandard-11.0.pdf

[XLIFF12] Y. Savourel, J. Reid, T. Jewtushenko, and R. M. Raya, Eds.: XLIFF Version 1.2, OASIS Standard. OASIS, 2008. Y. Savourel, D. Filip, R. M. Raya, and Y. Savourel, Eds.: XLIFF Version 1.2. OASIS Standard, 01 February 2008. OASIS. http://docs.oasis-open.org/xliff/v1.2/os/xliff-core.html

[XLIFF20] T. Comerford, D. Filip, R. M. Raya, and Y. Savourel, Eds.: XLIFF Version 2.0. OASIS Standard, 05 August 2014. OASIS. http://docs.oasis-open.org/xliff/xliff-core/v2.0/os/xliff-core-v2.0-os.html

[XLIFF21] D. Filip, T. Comerford, S. Saadatfar, F. Sasaki, and Y. Savourel, Eds.: XLIFF Version 2.1. OASIS Standard, 13 February 2018. OASIS http://docs.oasis-open.org/xliff/xliff-core/v2.1/os/xliff-core-v2.1-os.html

[XLIFF21prd02] D. Filip, T. Comerford, S. Saadatfar, F. Sasaki, and Y. Savourel, Eds.: XLIFF Version 2.1. Public Review Draft 02, February 2017. OASIS http://docs.oasis-open.org/xliff/xliff-core/v2.1/csprd02/xliff-core-v2.1-csprd02.html

[XLIFFEMBP] D. Filip and J. Husarčík, Eds., XLIFF 2 Extraction and Merging Best Practice, Version 1.0. Globalization and Localization Association (GALA) TAPICC, 2018. https://galaglobal.github.io/TAPICC/T1/WG3/rs01/XLIFF-EM-BP-V1.0-rs01.xhtml/>

[XLIFFglsTBXBasic] J. Hayes, S. E. Wright, D. Filip, A. Melby, and D. Reineke, Interoperability of XLIFF 2.0 Glossary Module and TBX-Basic, *Localisation Focus*, vol. 14, no. 1, pp. 43–50, Apr. 2015. https://www.localisation.ie/resources/publications/2015/260

[XLIFFRender] D. Filip and J. Husarčík, Modification and Rendering in Context of a Comprehensive Standards Based L10n Architecture, Proceedings ASLING Translating and the Computer, vol. 40, pp. 95–112, Nov. 2018. https://www.asling.org/tc40/wp-content/uploads/TC40-Proceedings.pdf