# From XML to UDL: a unified document language, supporting multiple markup languages

Hans-Jürgen Rennau, parsQube GmbH
<hans-juergen.rennau@parsqube.de>

**Abstract**

A proposal is made how to extend the XML node model in order to be compatible with JSON markup as well as XML markup. As XML processing technology (XPath, XQuery, XSLT, XProc) sees instances of the node model, but does not see syntax, it is thus enabled to handle JSON as well as XML. The extended node model is dubbed a Unified Document Language, as it defines the construction of documents from building blocks (nodes) which can be encoded in various markup languages (XML, JSON, HTML).

## Table of Contents

# Introduction

Is an XML document a string or a tree of nodes? Although in many situations it can be regarded as both, the “node view” is certainly more essential. It ignores syntax and sees the information content. Specialized programs (parser and serializer) provide for the translation between document string and node tree. General processing technologies – e.g. XQuery - ignore syntax. This principle is the very foundation of their power. Technologically speaking, the name “extensible markup language” is questionable, as it emphasizes the surface, rather than the content.

The node view of XML is the result of an evolutionary process. The XML specification [W3C XML] (1998) itself does not use the word “node”. The tree structure is still implicit, hidden behind the rules of well-formedness. The infoset specification [W3C Information Set] (2004) defined the XML document as a tree of information items, which is similar to a tree of nodes. The XDM [W3C XDM] (2007) rounded the tree model off, pruning it and extending it by replacing character children by a further node kind, the text node. This completed node model was at the heart of the technological leap which led to XPath 2.0, XSLT 2.0 and XQuery 1.0.

Taking the evolution into consideration, one might wonder about the relationship between XML and JSON. It is an obvious fact that they are two markup languages. But if XML is essentially not a syntax, but an information language backed by a syntax – then we should regard JSON as an information language plus syntax, too, and we should explore the relationships between their information models, rather than dwell on the difference of syntax.

Both models are tree models for hierarchical data. Why don’t we have one single, unified model for hierarchical data? If we had one, XML and JSON would not be two languages, but two syntactical styles – a difference that ceases to exist during data processing, between parsing and serializing the data. A uniform data model would enable unified data design approaches, and data processing could be handled by one single set of technologies. XML and JSON parsers would act like adapters. Unfortunately, such a unified model does not exist, due to incompatibilities. Although the XML model is larger and more complex, it is not a superset of the JSON model. XML lacks arrays and maps and XML names cannot be arbitrary strings.

These incompatibilities throw a new light on XML. In 2011 Jonathan Robie concluded [Robie]:

> *The dream of one universal markup language is now past. JSON is clearly here to stay, and it is becoming the format of choice for data interchange.*

A possible response to this perception is a new dream: the dream of one universal information language, backed by several syntax variants aka markup languages. If XML is not as universal as it looked a few years ago – might we extend the language (no pun intended), restoring the universality? This dream is an illusion if we regard XML as a final version. In past years, any considered improvements of XML were too insignificant to warrant the disruptions which a new version might entail. But now we face new proportions, dealing with the issue of universality. We should explore our chances to regain universality by extending the XML model in a moderate, backward compatible way. Eventually, the lessons JSON taught and teaches us might be to a huge benefit of XML – if we attempt to learn them and act accordingly.

# Distinction between markup and document language

XML processing technology operates on node trees which capture the information content of XML documents. The nodes may be constructed from XML markup text, or in other ways. An XML document *is* a tree of nodes, and it may be *represented* by markup. Therefore one might distinguish a **document language** from a **markup language**. The former is a system which defines basic units of information, possible relationships between such units and rules how they can be combined into composite entities, the document. A markup language is a set of rules how to encode a document as a string. To complete the picture, one might add the concept of an **information language**, which models information in a more general way – documents and their building blocks, the material that may be inserted into or extracted from a document, and perhaps yet other forms of information. The triple XML / Infoset / XDM may be viewed as a stack consisting of a markup language, a document language and an information language. This stack of languages is the foundation of technologies – XPath, XQuery, XSLT, XProc – which enable to address and process information with amazing simplicity and efficiency.

### Use of the term "node model"

The node sub model of the XDM can be regarded as a refactoring of the infoset. The existence of two very similar, yet distinct tree models – a tree of information items, and a tree of nodes representing the information items – is not really necessary and probably due to a historical process. In this paper, it is the XDM/nodes sub model what is regarded as the document model, rather than the infoset. The term used will be “XML node model”, or simply “node model”.

Nevertheless, JSON has begun to replace XML in many applications. JSON is a simpler and terser markup language, and it is perfectly integrated with JavaScript objects. In many situations, JSON has clear advantages, when neither the loss of expressive power, nor the lack of processing technologies hurt. There is a growing awareness of the need to be flexible, to avoid overhead and use the right tools: the necessity to adapt the choice of markup language to the task at hand [Tennison].

Doubtless, the technological support for JSON will continuously evolve. Very doubtful, however, it is if it can ever achieve the level attained by XQuery 3.0 and XSLT 3.0. Maybe this will not be possible without a similar evolution, adding to the markup language a document language and an information language on which to base technology. I cannot image that this would be possible without reinventing many wheels, with the end result – in the best case – of a more limited version of XQuery and XSLT.

Let us explore the alternative: loosen the tight coupling between the XML markup language and the XML document language, extending the latter to become a unified document language (“UDL”) supporting multiple markup languages – XML, JSON, HTML, …

# The main idea

This section presents the main idea of UDL – Unified Document Language - in a suggestive way and without any precision. It should provide a conceptual backdrop for the remaining sections. What is a document from the “UDL point of view”?

A document is a tree of elements. An element has content, which is either text, or other elements, or both. An element has also two properties designed to identify individual elements and to indicate the semantics of the content. One property is the **element name**, the other one the **element key**. The name is a QName and can be chosen irrespective of the sibling names; the key is an arbitrary string and must not be equal to any sibling key. This duality implies three different styles how a document may be designed:

- name oriented – the elements are identified and described by names

- key oriented – the elements are identified and described by keys
- mixed – making use of both, names and keys

A document can be represented as a string, using a markup language. XML is a markup language well-suited for name oriented documents; JSON, on the other hand, is good at representing key oriented documents. JSON is limited in this respect that it cannot represent arbitrary documents. Only documents meeting certain constraints (using only unspecific standard names, no attributes, no mixed content) can be represented by JSON. XML, on the other hand, can represent any document, though not very elegantly in the case of key oriented documents.

The foundation of document processing is the XPath language, with a core designed for selecting nodes within a document. Its query syntax supports a stepwise navigation across the document, where each step filters a set of candidate nodes by a so-called node test. One node test – the name test - refers to the name property. Example:

```
a/b[.//c]
```

This is a selection wholly based on element names. Another node test – the key test – refers to the key property. Example:

```
#a/#b[.//#c]
```

This selection is based on element keys, rather than names. Apart from that, the logic is exactly the same. Of course, node tests and key tests can be mixed:

```
a/b[.//#c]
```

To generalize, element name and element key are just two properties which XPath expressions and languages built upon XPath (XQuery, XSLT, XProc) can reference in a similar way. XML documents and JSON documents are alternative styles of how to represent a UDL document as text string. A parser translates XML documents and JSON documents into UDL documents. A serializer translates a UDL document into an XML document or a JSON document. The translation into XML is always possible without loss of information. The translation into JSON deals with any loss of information as prescribed by serialization parameters.

The remaining sections present a detailed proposal how to implement the UDL by very limited extensions of the XML node model, XML markup, XPath and XQuery.

# Goals and non-goals

The XML node model shall be turned into a unified document language, so that XML processing technology - which is built on the node model, not on markup - becomes a unified processing technology. In particular, the document language must support JSON so that the processing technology (XPath, XQuery, XSLT, XProc) becomes applicable to JSON data as well as to XML data.

From this high-level objective several goals are derived.

- Extend the XML node model, enabling it to represent the information content of JSON documents as a tree of nodes.
- Define the serialization to/deserialization from JSON markup.
- Extend the XML markup language, enabling it to express the extended node model completely.
- Extend the XML markup language, enabling the combination of XML and non-XML markup.

- Extend the XPath language, enabling navigation of JSON documents with the same degree of terseness and flexibility.
- Extend the XQuery language, adding shorthand notation for the construction of JSON data.
- Make any changes to the XML node model in a backwards compatible way.
- Make any changes to the XML markup language in a backwards compatible way.
- Make any changes to the XPath language in a backwards compatible way.
- Make any changes to the XQuery language in a backwards compatible way.

These are non-goals.

- Do not attempt to define a mapping from JSON markup to XML markup (rather, define deserialization from / serialization to JSON).
- Do not attempt to achieve elegance concerning the XML markup representation of a node tree derived from a JSON document.
- Do not attempt to support characters which are valid in JSON but are not valid in XML.

# Concepts

The proposed approach is an elaboration of a small number of concepts.

## The node model as a unified document language

1. The XML node model is extended in such a way that any JSON document can be translated into a node tree and back again without loss of information.
2. XML markup continues to represent the complete node model – an extension of the node model must be accompanied by an extension of the XML markup language.
3. As JSON markup represents a subset of the node model, the concept of serialization is elaborated, defining distinct modes characterized by the acceptable loss of information.

## The node representation of JSON markup

1. JSON structures (objects and arrays) and their members are modelled as element nodes and their child elements, thus enabling continuous navigation along the descendant axis.
2. As JSON names can be arbitrary strings and must be unique among sibling name/value pairs, they must not represent node names, which are QNames and need not be unique among sibling elements. Rather, JSON names correspond to a new node property, [key]. As a consequence, element nodes have two properties related to discovery and content semantics: a required [name] property and an optional [key] property.
3. The contradiction implied by the facts that node names are required and JSON is incapable of encoding node names is solved by the concept of *defaulted node names*: the nodes represented by JSON markup do have a name which is an unspecific standard name that depends on the content model of the node (representing an object, an array, a simple value or a null value). A node which has been constructed from JSON markup can afterwards be renamed without constraints, like any node constructed in any way.

## Extensions of the XML markup language

1. The necessary extension of the XML markup language avoids new syntactical constructs – it completely relies on the semantics of predefined QNames (e.g. `udl:key`) used in pseudo-attrib-

utes (constructs which look like an attribute but do not represent a node) and pseudo-tags (which look like an element but do not represent a node).

2. XML markup should be “opened”, permitting the local insertion of non-XML markup.

# Extensions of the XPath language

1. The XPath language is extended by a third node test – the *key test*, which checks whether the candidate node has a given key. In a path step, the key test can be used as alternative to a name test or kind test, which means that key tests are freely combinable with navigational axes.

2. The syntax of a key test should be as simple as the syntax of a name test.

# Extensions of the XQuery language

1. Extensions of the XQuery language are not essential, as JSON data are element nodes and thus can be processed without any restrictions.

2. Nevertheless, the addition of some abbreviated syntax for the construction of "JSON style nodes" would be quite helpful.

# Proposal: extensions of XML, XPath and XQuery

This section describes the proposed extensions of XML, XPath and XQuery in detail.

# Extensions of the XML node model

The XML node model is extended by two new node properties, [model] and [key]. The result of these changes is a unified node model which can represent XML documents, JSON documents as well as nested combinations of JSON and XML fragments as a tree of nodes which is accessible to XPath navigation and, by implication, XQuery and XSLT processing.

***Details***

1. The node model is extended by a further node property: the **[key] property**. Only element nodes have a [key], which is possibly empty. The [key] must not be empty if the [parent] is an element whose [model] property (see below) has a value of "map". In any other case (i.e. if [parent] is empty, or is not an element node, or is an element whose [model] is "sequence") the [key] must be empty. The [key] of an element must not be equal to the [key] of any sibling element.

2. The node model is extended by a further node property: the **[model] property**. Only element nodes have a [model], the value of which must be either "sequence" or "map". If the value is "sequence", the child nodes are an ordered collection and child elements must not have a [key]. Conversely, if the value is "map", the child nodes are an unordered collection, every child element must have a [key] and there must not be text node children containing a non-whitespace character. Note that the [model] can be regarded as a switch selecting one of two possible content models: sequence based (property value "sequence") or key based (property value "map"). In the former case element content is a sequence of child nodes; in the latter case element content is a map of child elements, using the child [key]s as map keys. The former case corresponds to "conventional XML", where content is always ordered by position.

3. **JSON simple values** are represented by (not nilled) element nodes which have simple content, or (in the case of a zero-length string) empty content and a [schema-type] xs:untypedAtomic.

4. **JSON null values** are represented by nilled elements.

5. **JSON objects** are represented by (not nilled) element nodes with [model] equal "map". By implication, such elements may or may not have child elements, but they have no text node children containing non-whitespace characters. The name/value pairs contained by the object are represented by the element children. Other child nodes (e.g. comments or whitespace-only text nodes) do not correspond to name/value pairs. Note that an empty object is represented by an element with [model] equal "map" and no child elements.

6. **JSON arrays** are represented by (not nilled) element nodes satisfying these constraints: (a) [model] equal "sequence", (b) the content is either empty or contains at least one child element; (c) there are no text node children containing non-whitespace characters. The array members are represented by the element children. Other child nodes (e.g. comments or whitespace-only text nodes) do not correspond to array members.

7. **JSON names** are represented by the [key] property of element nodes.

8. When a node tree is constructed from a JSON document, null values, simple values, arrays and objects are represented by elements which have **default node-names** ( `udl:null`, `udl:value`, `udl:array` and `udl:map`). As any element names in XML, these names do not have any built-in semantics: they do not signal that the element has been constructed from a JSON value, and they do not imply specific values of any node properties. After an update or if the node tree is constructed in any other way, the elements representing null values, simple values, arrays and objects may have any valid node name.

   Note that JSON names and XML names correspond to two distinct node properties which are utterly independent of each other. And also note the asymmetry: whereas JSON names are represented in XML markup by keys (via the `udl:key` pseudo-attribute, see next section), XML names cannot be represented in JSON markup at all. Lossless information mapping in both directions is nevertheless enabled by arbitrarily defining JSON markup to represent nodes with default names which are implied by other node properties.

# Extensions of the XML markup language

The extensions have two purposes: (a) express the new node properties; (b) support the use of non-XML markup within XML documents.

## Expressing the new node properties

The XML markup language is extended by rules how to represent the new node properties.

***Details***

1. A pseudo-attribute (`udl:model`) is introduced which indicates the value of the **[model] property**. Possible values are "sequence" and "map". The default value is "sequence", unless the element has an ancestor element with a pseudo-attribute `udl:defaultModel`, in which case the default is specified by the nearest ancestor with a `udl:defaultModel` pseudo-attribute.

2. A pseudo-attribute (`udl:defaultModel`) is introduced which sets the default value of [model] for the element itself and its descendants. The default value applies to the element itself and to its descendant elements unless the element in question has simple content (in which case [model] is always "sequence"), or has a [model] pseudo-attribute (which overrides the default) or has a nearer ancestor with a `udl:defaultModel` pseudo-attribute (which shadows any outer default values).

3. A pseudo-attribute (`udl:key`) is introduced which indicates the value of the **[key] property**. If an element without `udl:key` is child of an element whose [model] is "map", the [key] defaults to the local name of the element. Example:

```
<foo udl:model="map">
```

```
   <bar udl:key="bar">abc</foo>
</foo>
```

is equivalent to:

```
<foo udl:model="map">
   <bar>abc</foo>
</foo>
```

Note that non-empty [key]s are only allowed for elements with [parent].[model] equal "map". Accordingly, only such elements may have a `udl:key` pseudo-attribute. Example: if in the following markup

```
<foo udl:model="map">
   <bar udl:key="21">abc</foo>
</foo>
```

the value of `udl:model` were changed to "sequence", the markup would cease to be well-formed.

## Supporting non-XML markup

The extensions enable the use of non-XML markup either embedded in XML markup or completely replacing it.

***Details***

1. The XML syntax model is extended by permitting **alternative markup languages**. An alternative language can be used in three different scopes: (a) the content of an element, (b) a document section of arbitrary length, representing any number of sibling nodes, (c) the complete document. Three languages are supported: `xml`, `json`, `telem`, a slightly simplified version of XML using JSON-like constructs for simple elements meeting certain constraints. See the section called “ Mixing markup styles ” for details.

2. The XML syntax model is extended by a pseudo-attribute, `udl:markup`, which specifies the markup language used to represent the content of an element. If the value is not `xml`, the child nodes of the element are the nodes constructed from the markup found in the text content. Only element tags and the pseudo-tag `udl:markupSection` (see below) may have this pseudo-attribute. Possible values are: `xml`, `json`, `telem`; default value is `xml`. Example:

```
<temperatures y="2012"
              udl:markup="json"><![CDATA[
   "2012-08-01" : 33.2,
   "2012-08-02" : 28.9,
   "2012-08-03" : 30.0,
   "sites" : ["AB", "DK", "PP"],
   "anno" : {"automatic" : true, "reference" : false}
]]></temperatures>
```

   Note that the scope of the alternative markup language is the content of an element and that the alternative representation is preceded and followed by the XML start and end tag of the element. Thus the markup of the example corresponds to an element with name “temperatures”, which has one attribute and five child elements. An XML document can use different markup languages in different elements.

3. The XML syntax model is extended by a pseudo-tag `udl:markupSection`, which delimits a markup section, a section of the document text which uses a particular markup language. When

constructing the node tree, the pseudo-tag and its contents represent the nodes constructed from the contained markup. The markup language is identified by the `udl:markup` pseudo-attribute contained by the pseudo-tag. In the following example, the pseudo-tag represents five nodes which are constructed from the JSON markup:

```
<udl:markupSection udl:markup="JSON"><![CDATA[
   "2012-08-01" : 33.2,
   "2012-08-02" : 28.9,
   "2012-08-03" : 30.0,
   "sites" : ["AB", "DK", "PP"],
   "anno" : {"automatic" : true, "reference" : false}
]]></udl:markupSection>
```

Note that the pseudo-tag does itself not represent a node – it has a purely delimiting function. Any non-XML markup may be used which is supported by the parser. Besides JSON, a parser may support an implementation-defined set of further markup languages or domain specific languages.

4. The **XML declaration** is extended by a further field: `markup`. Possible values are: `xml`, `json`, `html`; default is `xml`. Depending on the value, the text following the XML declaration will be interpreted as XML markup, JSON markup or HTML markup. Example:

```
<?xml markup="json" encoding="ISO-8859-1"?>
{
   "title" : "JSON and XML",
   "year" : 2012
}
```

5. The XML markup language is augmented by a rule how to parse a **non-XML document without XML declaration**. If the first non-whitespace character of the text is not the "<" character, the document text is interpreted as non-XML markup. More precisely, it is interpreted as the default non-XML markup which is expected to be JSON, although implementation-defined alternatives might be considered. Example: the text

```
{
   "title" : "JSON and XML",
   "year" : 2012
}
```

is a valid UDL document.

# Extensions of the XML serialization model

The serialization model must be extended in order to support JSON output.

1. When the serialization method is `xml`, serialization produces conventional XML markup, augmented by the pseudo-attributes `udl:key`, `udl:model` and `udl:defaultModel` where appropriate.

2. When the serialization method is `xml`, the serialization may nevertheless insert non-XML markup into the document text, depending on serialization parameters. The non-XML markup is constrained to represent element contents – that is, every chunk of non-XML markup is scoped to represent the content of an element whose start and end tag delimit the chunk.

3. When the serialization method is `xml`, additional serialization parameters control the use of alternative markup within selected elements. Parameter `json-content-elements` contains a list

of expanded QNames, identifying the elements whose content shall be represented as JSON markup. In a similar way, parameter `telem-content-elements` identifies the elements to be rendered using the `telem` style. (For details see the section called “ Mixing markup styles ”).

4. The value range of serialization parameter `method` is extended by the value `json`. This value lets the complete document be serialized as JSON markup.

5. A new serialization parameter `info-loss` specifies how to handle information loss implied by the serialization. Special values relate to situations where JSON markup should be produced but a node to be serialized contains information which cannot be expressed by a JSON representation. (There are three cases: (i) mixed content, (ii) the use of attributes, (iii) the use of non-standard element names.) Three parameter values are supported: `json.strict`, `json.ignore-names`, and `json.projection`. In case of `json.strict` the serialization must be aborted; the value `json.projection` mandates a projection which simply ignores any information which cannot be represented; and the value `json.ignore-names` means that the QNames of XML elements are ignored, but any other incompatibility with the JSON model (e.g. the use of attributes) produces an unrecoverable error. (For details see the section called “ Serialization: controlling the loss of information ”.)

# Extensions of the XPath language

The extensions are designed to make the processing of JSON data as powerful and convenient as the processing of XML data. Namely, the [key] property can be checked by a *key test*, similarly to the checking of the node name by a node test.

1. The XPath language is extended by a new node test, an alternative to the existing name test and kind test: the **key test**. A key test checks for the candidate node if it has a key equal to a given key value. In path expressions, key tests can be combined with XPath axes in the same way as kind tests and name tests. The syntax of a key test is a # character immediately followed by the key value delimited by single or double quotes. If the key value contains only name characters, the quotes can be omitted. If quotes are used, occurrences of the actual quote character within the key value must be escaped by an entity or character reference. The characters & and < must always be escaped. Examples of path steps containing a key test:

```
#key1
#"key1"
#”key 2”
self::"#key 2"
descendant::#key3
parent::#"#key4"
ancestor::#'++14085! &amp; O&apos;Neill'
```

2. A new XPath function `fn:node-key` returns the [key] of a given node, or the empty sequence if the node has no [key]:

```
fn:node-key($node as node()?) as xs:string?
```

Example: the expression

```
string-join($x/ancestor-or-self::*/(concat(‘#’, fn:node-key(.)), ‘/’)
```

might return a result like `#a/#b/#c`.

3. A new XPath function `fn:node-model` returns the [model] of a given node, or the empty sequence if the node is not an element node. The [model] is represented as a string which is either "sequence" or "map":

```
fn:node-model($node as node()?) as xs:string?
```

4. The semantics of function `fn:deep-equal` is modified as follows: (a) if the arguments are element nodes with different [key]s or with different [models]s, the function returns "false"; (b) if both arguments are element nodes with [model] equal "map", the comparison ignores non-element children and ignores the order of element children.

5. The *abbreviated syntax* is extended by a more intuitive syntax for accessing array members by index, which hides the fact that array members are child nodes:

```
foo~[expr]
```

is equal to

```
foo/*[expr]
```

# Extensions of the XQuery language

As JSON items correspond to element nodes, there is no principal need to introduce new constructor expressions. In element constructors, the pseudo-attributse `udl:key` and `udl:model` are used in the same way as they are used in XML markup. In order to reduce verbosity, however, several abbreviated variants of element constructors are introduced.

1. **Map constructors** are a shorthand for constructing element nodes with name `udl:map` and [model] equal "map". Syntax:

```
{ Expr }

   is equivalent to:

<udl:map udl:model="map">{ Expr }</udl:map>
```

The children of the newly constructed `udl:map` element are obtained by (a) evaluating the content expession to an item sequence, (b) replacing in this sequence any document node by its document element, (c) replacing in the resulting sequence any element without a key by a copy which has a key equal to its local name. An error is raised if the result sequence contains atomic or text node items, or if it contains two elements with the same key. Otherwise, the expression value is guaranteed to be an element which can be serialized to JSON without information loss.

2. **Array constructors** are a shorthand for constructing element nodes which correspond to a JSON array. Syntax:

```
[ Expr ]

   is equivalent to the following code
   (where p:copy-without-key denotes a pseudo function creating an element co

<udl:array>{
  for $item in Expr return
    typeswitch($item)
    case document-node()  return $item/*/p:copy-without-key(.)
    case element()        return $item/p:copy-without-key(.)
    case text()           return <udl:value>{$item}</udl:value>
    case xs:anyAtomicType return <udl:value>{$item}</udl:value>
```

```
    default                return ()
}</udl:array>
```

The children of the newly constructed `udl:array` element are obtained by (a) evaluating the content expression to an item sequence, (b) replacing in this sequence any document nodes by their element children, (c) replacing in the resulting sequence any element with a key by a copy which does not have a key, (d) replacing in the resulting sequence any atomic values by a `udl:value` element containing the value as text. The expression value is guaranteed to be an element which can be serialized to a JSON array without information loss.

3. **Key-oriented constructors** are a shorthand for constructing element nodes with a non-empty [key]. They have the following syntax:

```
Expr ':' Expr

Examples:
"title" : "XML and JSON"
$ti     : $tnode
"title" : //title
"times" : [ "2012-01-01", "2012-03-31" ],
"time" : { "begin" : "2012-01-01", "end" : "2012-03-31" }
```

The value of this expression is determined as follows.

- Evaluate the expression to the left of the colon; the result must be a single item; determine its string value S.
- Evaluate the expression to the right of the colon; the result R must be either the empty sequence or a single item.
- If R is the empty sequence, the value of the constructor expression is an element node with name `udl:null`, a [key] property equal S and a [nilled] property equal true.
- If R is a node, the value of the constructor expression is a node obtained by making a copy of R and setting its [key] property to S.
- Otherwise (that is, if R is an atomic value) the value of the constructor expression is an element node with the name `udl:value`, a [key] property equal S and a single text node child whose string value is the string value of R. (Special case: empty content if the string value of R is a zero-length string.) The resulting element has a type annotation which depends on the type of R. If R has a number type, the type annotation is one of these: `xs:double`, `xs:decimal`, `xs:integer`, whatever is closest to the type of R. If R has a boolean type, the type annotation is `xs:boolean`. If R is a zero-length string, the type annotation is `xs:untypedAtomic`. Otherwise, the default type annotation is used ( `xs:untyped`).

# Checking use cases

The proposal made in this paper is motivated by several main use cases. In each of these, a significant simplification of the task should be achieved.

- JSON documents must be queried.
- JSON documents must be transformed into other JSON documents.
- JSON documents must be transformed into XML documents.
- JSON documents must be transformed into HTML documents.
- JSON documents must be transformed into other formats (e.g. CSV).

- JSON documents must be created from XML documents.
- JSON documents must be created from other formats (e.g. CSV).

The fact that XML standard technologies - XPath, XQuery, XSLT, XProc - now accept JSON documents as input suggests a great advantage. We should however take a closer look at how the processing of JSON data looks. Somewhat arbitrarily, these main aspects may be distinguished:

- Selecting JSON data
- The use of JSON data within XPath/XQuery expressions and XSLT instructions
- Creating JSON data

The **selection of data** is a crucial operation, underlying virtually all forms of data processing. This is the domain of XPath, so we shall take a look at how XPath deals with JSON data. The proposal avoids the creation of special item types - all JSON data reside in element nodes. Therefore it can be expected that the **use of JSON data in expressions** and XSLT instructions is indistinguishable from the use of any other element nodes. The **creation of JSON data** amounts to the creation of element nodes, so that again we may expect the same ease when creating JSON data as when creating any other element nodes.

Let us contemplate a few examples. As input we use the following JSON document:

```
[
   {
      "year" : 2011,
      "title" : "JSON",
      "author" : [
         {"last" : "Legoux", "first" : "C."}
      ],
      "price" : 35.95,
      "sigs" : ["LL1002"]
   },
   {
      "year" : 2012,
      "title" : "XML",
      "author" : [
         {"last" : "Legoux", "first" : "C."},
         {"last" : "Berlin", "first" : "D."}
      ],
      "price" : 29.95,
      "sigs" : []
   },
   {
      "year" : 2012,
      "title" : "UDL",
      "author" : [
         {"last" : "Legoux", "first" : "C."},
         {"last" : "Okuda", "first" : "J."},
         {"last" : "Berlin", "first" : "D."}
      ],
      "price" : 49.95,
      "sigs" : ["KL4005", "KL4011"]
   }
]
```

The following table shows a series of *data selections* with XPath/XQuery. The expressions typically use key tests (#foo) instead of name tests. Apart from that there is no difference compared to con-

ventional uses of XPath. Writing the expressions, one must keep in mind that object members (the name/value pairs) and array members are represented by child elements of the element representing the object or array, respectively.

**Table 1.**

***Selecting JSON data with XPath/XQuery.***

| task | expression | result |
|---|---|---|
| count books | `count(/*/*)` | 3 |
| maximum price | `max(//#price/xs:decimal(.))` | 49.95 |
| first book title | `/*/*[1]/#title/string()` | JSON |
| all publication years | `distinct-values(//#year/string())` | 2011 2012 |
| books about UDL | `//#title[contains(., 'UDL')]/string()` | UDL |
| books above 30$ | `//#title[../#price/xs:decimal(.) gt 30]/string()` | JSON UDL |
| books with a single author | `//#title[count(../#author/*) eq 1]` | JSON |
| books without signature | `//#title[empty(../#sigs/*)]` | XML |
| books written by Legoux | `/*/*[.//#last = 'Legoux']/#title/string()` | JSON XML UDL |
| coauthors of Legoux | `distinct-values(//#last[. eq 'Legoux']/../../*/#last[. ne 'Legoux'])` | Berlin Okuda |
| duplicate signatures | `for $s in distinct-values(//#sigs/*)`<br>`where count(//#sigs[* = $s]) gt 1`<br>`return $s` | LL1002 |

In order to get a feeling how selected JSON data can be *used in expressions* and how JSON data can be *constructed*, we build a report that transforms the input data into a new structure. The report shall list for each author all titles he or she has authored, along with the publication year. An XML version of the report might look like this:

```
<authors>
   <author name="Legoux, C.">
      <book title="JSON" year="2011"/>
      <book title="UDL" year="2012"/>
      <book title="XML" year="2012"/>
   </author>
   <author name="Okuda, J.">
      <book title="UDL" year="2012"/>
```

```
   </author>
   <author name="Berlin, D.">
      <book title="UDL" year="2012"/>
      <book title="XML" year="2012"/>
   </author>
</authors>
```

and a JSON version like this:

```
[
   {
      "author" : "Legoux, C.",
      "books" : [
         {"title" : "JSON", "year" : "2011"},
         {"title" : "UDL",  "year" : "2012"},
         {"title" : "XML", "year" : "2012"}
      ]
   },
   {
      "author" : "Okuda, J.",
      "books" : [
         {"title" : "UDL", "year" : "2012"}
      ]
   },
   {
      "author" : "Berlin, D.",
      "books" : [
         {"title" : "UDL", "year" : "2012"},
         {"title" : "XML", "year" : "2012"}
      ]
   }
]
```

The XML report can be produced with this query:

```
<authors>{
   for $author in distinct-values(//#author/*/concat(#last , ', ', #first))
   let $books := //#author[*/concat( #last , ', ', #first ) = $author]/..
   order by $author
   return
      <author name="{$author}">{
         for $book in $books
         order by $book/#title
         return
            <book title="{$book/#title}" year="{$book/#year}" />
      }</author>
}</authors>
```

and the JSON version can be produced with this query:

```
<udl:array>{
   for $author in distinct-values(//#author/*/concat(#last , ', ', #first))
   let $books := //#author[*/concat( #last , ', ', #first ) = $author]/..
   order by $author
   return
      <udl:map udl:model="map">{
```

```
            <udl:value udl:key="author">{$author}</udl:value>,
            <udl:array udl:key="books">{
               for $book in $books
               order by $book/#title
               return
               <udl:map udl:model="map">{
                  <udl:value udl:key="title">{$book/#title/string()}</udl:value>,
                  <udl:value udl:key="year">{$book/#year/string()}</udl:value>
               }</udl:map>
            }</udl:array>
         }</udl:map>
}</udl:array>
```

The element constructors required to create JSON nodes are somewhat verbose, and the code is not very readable as the distinctive information - the key - is embedded in stereotyped markup (e.g. `<udl:map udl:key="...">` The situation can be amended by resorting to the abbreviated constructors for maps and arrays along with the key-oriented constructors (see the section called " Extensions of the XQuery language "):

```
[
   for $author in distinct-values(//#author/*/concat(#last , ', ', #first))
   let $books := //#author[*/concat( #last , ', ', #first ) = $author]/..
   order by $author
   return
      {
         "author" : $author,
         "books" : [
            for $book in $books
            order by $book/#title
            return
               {
                  "title" : $book/#title/string(),
                  "year" : $book/#year/string()
               }
         ]
      }
]
```

The code examples demonstrated that the processing of JSON data with XPath and XQuery is comparable to the processing of XML data. For all use cases one may expect from XPath/XQuery/XSLT/XProc the same level of support which one is used to get when dealing with similar problems related to XML without JSON. This may be taken as encouragement to explore the proposal in greater detail.

# Various details

## UDL - pseudo-attributes and pseudo-tags

Pseudo-attributes are syntactical constructs which have the lexical form of attributes but can be distinguished from them by the use of a reserved QName. Pseudo-attributes do not represent an attribute node. Instead, they represent a node property (`udl:key`, `udl:model`) or a default value of a property (`udl:defaultModel`), or they identify the markup language used locally (`udl:markup`).

Pseudo-tags are syntactical constructs which have the lexical form of element tags but can be distinguished from them by the use of a reserved QName. One pseudo-tag is introduced (`udl:markupSection`) which delimits a section of non-XML markup.

Four further names from the UDL namespace are used as default element names, given to the nodes constructed from JSON values. It is important to note that these names have no specific semantics and can be used as node name without restrictions like any other QName. The only specific treatment of these names is when serializing to JSON using the `json.strict` mode. In this case a node name which is different from the default name expected (according to the node properties) is considered information that would be lost during serialization (see the section called “ Serialization: controlling the loss of information ”).

The following table summarizes the use of QNames from the UDL namespace.

**Table 2.**

***Names in the `udl` namespace and their usage.***

| Name | Usage category | Meaning |
|---|---|---|
| `udl:null` | element name | a standard name available for nilled elements with an unspecific name |
| `udl:value` | element name | a standard name available for a simple content element with an unspecific name |
| `udl:array` | element name | a standard name available for a complex element with [model] equal "sequence" |
| `udl:map` | element name | a standard name available for a complex element with [model] equal "map" |
| `udl:markupSection` | pseudo tag | delimits a markup section containing markup which may be non-XML; the section represents the nodes resulting from parsing the contained markup text |
| `udl:markup` | pseudo attribute | indicates the markup language used within element content, or within a markup section |
| `udl:model` | pseudo attribute | represents the [model] property value |
| `udl:defaultModel` | pseudo attribute | sets a default value for the [model] property |
| `udl:key` | pseudo attribute | represents the [key] property value |

# Mixing markup styles

The UDL defines a unified document model which can be represented by different markup languages. This unified content of heterogeneous outward shape invites not only a free choice of the markup language actually used. It also implies that markup languages might be mixed within a document, based on simple rules how to delimit the various chunks of markup. These rules are provided by the `udl:markup` pseudo-attribute and the `udl:markupSection` pseudo-tag (see the section called “ Supporting non-XML markup ”).

Occasionally there may be good reasons to use mixed styles. Consider the case that the document as a whole cannot be represented as JSON (e.g. because of attributes and namespaces), but subtrees represent JSON documents (perhaps imported from pure JSON sources, e.g. logged messages).

Without the mixing of markup styles, the resulting serialization would be difficult to read, due to the very verbose XML representation of JSON nodes. It should of course be remembered that this mixing of markup styles has no impact on the information content of the UDL document, which is exclusively defined in terms of nodes and their properties.

The following section describes in detail an additional markup style, which amounts to a "small-scale mixing" of XML and JSON, dubbed `telem` (text notation for simple elements).

# XML syntax variant: `telem`

XML markup representing JSON data is ugly. Typically it contains many elements which correspond to simple values and are tiresome to read. The *distinctive* property of the elements is shifted from the eye-catching node name to a pseudo-attribute, and the markup is often inflated by explicit type annotations:

```
<udl:value udl:key=”foo”>someContent</udl:value>
<udl:value udl:key=”bar” xsi:type="xs:integer">99</udl:value>
<udl:value udl:key=”foobar” xsi:type="xs:boolean">true</udl:value
```

whereas the JSON representation could not be more succinct:

```
"foo" : "someContent",
”bar”" : 99,
"foobar” : true
```

Fragments containing *only* JSON nodes can best be represented by switching to JSON. But sometimes such JSON values occur interspersed with conventional XML elements which have specific names, have attributes, etc. In such cases it is attractive to apply the JSON style to the simple values and retain XML style for the fragment as a whole. This option is provided by the `telem` markup style.

This style is XML markup augmented by a shorthand representation of simple elements meeting several constraints:

- element name is the standard name `udl:value`
- simple content or nilled
- no attributes
- [schema-type] is one of these: `xs:integer`, `xs:decimal`, `xs:double`, `xs:boolean`, `xs:untypedAtomic`, `xs:untyped`

The syntax corresponds to the JSON representation of a simple or null value, or of a name/value pair with simple or null value, depending on whether the element has a [key]. If consecutive element children are represented in `telem` style, these representations are separated by a comma. If the value is not put in quotes, it must be a number or one of the constants `true`, `false` or `null`, which will be interpreted as implicit type information, following the JSON rules. Example: the following fragment

```
<e udl:model="map">
   <udl:value udl:key="mode">repeated</udl:value>
   <udl:value udl:key="nrep" xsi:type=”xs:integer”>52076</udl:value>
   <udl:value udl:key="eval" xs:type=”xs:boolean”>true</udl:value>
   <locInfo udl:key="cities">
     <udl:value>Paris</udl:value>
```

```
      <udl:value>London</udl:value>
      <udl:value>Manchester</udl:value>
    </locInfo>
</e>
```

may be alternatively represented this way:

```
<e udl:model="map" udl:markup=”telem”>
   "mode" : “repeated”,
   "nrep" : 52076,
   “eval” : true
   <locInfo udl:key="cities">
     "Paris",
     "London",
     "Mancester"
   </locInfo>
</e>
```

Both representations have the same information content.

# Deserializing from / serializing to JSON

The exact rules for translating JSON into UDL (deserialization) and for translating UDL into JSON (serialization) are listed in the appendix (the section called “ Deserialization ” and the section called “ Serialization ”). In this section, the principles are summarized.

## Deserialization

During deserialization every JSON “item” (object, array, simple value, null) is translated into a UDL element node whose name and content are determined by the kind of the JSON item (see Table 3, “

***Deserialization - translating JSON items into UDL nodes.***

”). If the JSON item is associated with a name, the name is copied into the [key] property of the element node; otherwise the element node has no [key].

Numbers and Boolean constants are translated into simple elements with a [schema-type] property reflecting the source item (one of: `xs:integer`, `xs:decimal`, `xs:double`, `xs:boolean`). A string which has non-zero length is translated into a simple element with [schema-type] `xs:untyped`. A zero-length string is translated into an empty element node with [schema-type] `xs:untypedAtomic`, so as to make it distinguishable from a node constructed from an empty array or object.

### Table 3.

### *Deserialization - translating JSON items into UDL nodes.*

| **JSON item** | **UDL node properties** | | | **remarks** |
|---|---|---|---|---|
| | **node-name** | **model** | **children** | |
| name/value pair | see below | see below | see below | the JSON value can be any item kind (null, simple value, object, array); |

| JSON item | UDL node properties | | | remarks |
|---|---|---|---|---|
| | node-name | model | children | |
| | | | | all node properties – except for the [key] – depend on the item kind;<br>the [key] is set to the JSON name |
| null | `udl:null` | sequence | none | element is nilled |
| object | `udl:map` | map | elements, one for each name/value | all child elements have a [key] |
| array | `udl:array` | sequence | elements, one for each member | all child elements without a [key] |
| string (non-empty) | `udl:value` | sequence | text node | [schema-type] is xs:untyped |
| zero-length string | `udl:value` | sequence | none | [schema-type] is xs:untypedAtomic |
| number | `udl:value` | sequence | text node | [schema-type] is one of: xs:integer, xs:decimal, xs:double |
| true\|false | `udl:value` | sequence | text node | [schema-type] is xs:boolean |

## Serialization

The translation of UDL nodes into JSON items does not depend on node names; rather, it is wholly determined by the element content (empty / element children / text child) and several properties ([key], [model], [nilled], [schema-type]). The node name is however checked if the serialization parameter `info-loss` is `json.strict`. In this case, the actual node name is compared with the default node name associated with the given element content and properties, and an unrecoverable error is raised if actual node name and expected node name are not the same.

See the section called “ Serialization: controlling the loss of information ” for details about how serialization may accept or reject loss of information, dependent on serialization parameter `info-loss`.

**Table 4.**

***Serialization - translating UDL nodes into JSON items. CT = complex type with complex content; ST = simple type.***

| node properties | | | | JSON item |
|---|---|---|---|---|
| children | model | nilled | schema-type | |
| empty | sequence | false | `xs:untyped` or CT | array (empty) |
| empty | sequence | false | `xs:untypedAtomic` or ST | string (zero-length) |
| empty | map | false | any | object (empty) |
| empty | sequence | true | any | null |
| element children | sequence | false | any | array |

| node properties | | | | JSON item |
|---|---|---|---|---|
| **children** | **model** | **nilled** | **schema-type** | |
| element children | map | false | any | object |
| text node | sequence | false | `xs:double` | number |
| text node | sequence | false | `xs:decimal` | number |
| text node | sequence | false | `xs:integer` | number |
| text node | sequence | false | `xs:boolean` | `true`\|`false` |
| text node | sequence | false | `xs:untypedAT` | or string |

# Serialization: controlling the loss of information

Serialization of a document to a markup language should preserve all information so that the serialization is a complete representation from which the document may be reconstructed. Such a lossless serialization of a UDL document is always possible for XML markup; it is only in special cases possible for JSON markup. For example, any attributes or non-default element names are lost when serializing to JSON.

However, it depends on circumstances whether such loss of information renders the serialization result worthless. If, for example, the loss consists of element names only and these names were only introduced in order to facilitate document creation or processing, with an understanding that they will get lost during later processing steps – then a serialization which loses element names might be as valuable as a lossless serialization. Such considerations suggest a refinement of the serialization model: a new serialization parameter might control what losses are acceptable and what losses are not acceptable.

The proposal of a unified document language includes such a new serialization parameter: `info-loss`. Presently the parameter is only relevant when serializing to JSON. Three values are defined:

- `json.strict` – any information loss causes an unrecoverable error
- `json.ignore-names` – element names are ignored, but any other information loss causes an unrecoverable error
- `json.projection` – any information that JSON cannot represent is simply ignored

In particular, `info-loss` equal `json.projection` means:

- element names are ignored
- attributes are ignored
- text node siblings of element nodes are ignored (that is, mixed content is projected onto the element children)

# UDL and XSD

The proposed extensions of the XML node model amount to the introduction of two new node properties. Obviously, they require also an extension of the XSD language. In particular, constraints concerning the [key] property should be supported. However, such changes are out of scope of this paper.

# Limitations and future research

UDL defines the information content of JSON text in terms of a node tree and provides the rules for translating between text and tree, that is, parsing and serialization. This makes JSON data accessible

to / producible by XML processing technologies, but there are also important use cases which are not addressed:

1. given a JSON document, a well-readable XML representation is required
2. given an arbitrary XML document, a JSON representation is required
3. round-tripping XML - JSON - XML

As will be shown below, these are operations which require some change of information content in the formal sense (in terms of nodes and properties), necessary to create a “semantic” equivalence. Such a change of information content cannot be achieved based on parsing/serialization alone. This section explores the basic limitation and makes a suggestion how an extension of the current UDL proposal might look.

# Issue: mapping arbitrary XML to JSON

A serialization of arbitrary XML documents to JSON is usually not possible without a loss of information, as JSON cannot natively express element names, the distinction between attributes and elements, mixed content and the occurrence of siblings with the same name. This does not mean that JSON could not be used to represent the complete information content of an arbitrary XML document. This representation would however not be a serialization of the XML document tree to JSON, but the (serialized) result of a *transformation*, a different node tree, adhering to a specific format which is designed to capture the content of arbitrary XML documents (e.g. [JsonML] ). The equivalence between the resulting JSON (or the node tree it represents) and the original XML document is not based on the data model, but established on the level of a specific mapping application (as a set of rules).

# Issue: mapping JSON to readable XML

A similar problem concerns the translation of JSON documents into readable XML documents: the documents created by parsing JSON as defined by UDL are well-suited for processing (e.g. per XPath, XQuery, XSLT), but when serialized into XML text look hardly readable. The practical need to obtain a well readable XML representation of a given JSON document, however, cannot be denied. (Think, for example, of a web service which may at user option deliver XML or JSON results). Again, it is a transformation from one tree to a different tree what is required, as opposed to serializations into alternative formats.

# Issue: Round-tripping XML - JSON - XML

The impossibility of serializing arbitrary XML to JSON of course implies that round-tripping XML-JSON-XML is not generally possible solely based on serialization and parsing.

# Conceivable extension of UDL: integration of standardized mappings

Given the scope of UDL’s main goal – a unified document language supporting multiple markup languages – it may be questioned if the UDL concept is complete if not addressing fundamental mapping tasks, too. A conceivable extension of UDL might include two parts: (a) the definition of *mappings*, which are standardized transformations (XML to JSON, lossless; JSON to readable XML; (b) the integration of these mappings with parsing / serialization into new “first-class” operations, “*mparse*” (parse & map) and “*mserialize*” (map & serialize). The appendix contains a first step in this direction [Appendix B, *Additional support for "NCName-only JSON"*], which is, however, limited to the use case of JSON documents in which all names are NCNames. Another limitation is that it does not yet support attributes in the mapping result, which probably cannot be the last say.

The mapping between XML and JSON is a question to which already many answers have been given (e.g. [Lee], [Pemberton], [Couthures], [JsonML], [BaseX]). The diversity is mainly due to differ-

ences between the exact goals which the solutions pursue (concerning mapping direction, losslessness, readability, configurability, …). It is unclear if the suggested extension of UDL, which involves standardized mappings between XML and JSON, is a realistic task. But it is not hopeless, neither. First, the unified node model provides a conceptual base which other mapping approaches did not have. Second, the goals can be defined precisely, which greatly removes competition between existent solutions and a new standard. And finally, the use of other, non-standard mappings would always remain possible.

# Alternatives to UDL

How to process XML and JSON data in a unified way? The approach taken by UDL should be compared with published alternatives. These fall into two categories.

- mapping approach – map JSON data to an XML representation and process the latter
- XDM extension – extend the XDM by new item types which can represent JSON data

The second category contains two variants:

- extend the XDM by JSON-specific item types (JSONiq)
- extend the XDM by generic item types (W3C XSL Working Group proposal)

The next three sections attempt to clarify the relationships between these approaches and UDL.

# The mapping approach

The mapping approach (e.g. BaseX, Pemberton) is based on an XML representation of JSON data. It uses a simple processing model:

- preprocessing: JSON => XML
- processing: applied to XML data
- optional postprocessing: serialization (possibly to JSON)

This is a clean solution, provided the XML representation preserves all information contained in the JSON data, and the mapping rule can be applied bidirectionally without loss of information. To define such a mapping is not very difficult, as one can use reserved element names and introduce helper attributes in order to exclude any information loss (see Table 5, “

***The use of reserved element names and helper attributes to assist in the mapping of JSON to/from XML (examples).***
” for examples).

**Table 5.**

***The use of reserved element names and helper attributes to assist in the mapping of JSON to/from XML (examples).***

| source | elements | attributes |
|---|---|---|
| BaseX | `json, value` | `arrays, booleans, nulls, numbers, objects, type` |
| Couthures | `exml:anonymous` | `exml:fullname, exml:maxOccurs` |
| Hunter | `json, item` | `boolean, type` |

| source | elements | attributes |
|---|---|---|
| Pemberton | `json` | `name, starts, type` |

Outwardly, UDL looks similar to such a mapping-for-the-sake-of-processing; it is tempting to classify it as yet another mapping variant. But that would be a mistake. Mapping approaches treat the problem as an XML *application*: introduce a specific XML dialect designed to achieve a particular goal. Like any application, these approaches are free to require the use of application-specific element names and the addition of attributes with application-specific semantics, to be evaluated by application code. The `type` attribute, for example, used in [BaseX] is a helper attribute which clearly duplicates `xsi:type` for certain values, yet nevertheless had to be introduced as additional attribute, because the value range includes values `array` and `object` with ad hoc semantics dictated by the mapping task. Such attributes reveal the fact that the current XML node model does not support a bidirectional mapping into JSON markup. To enable such a mapping, the node tree must contain special items with serialization semantics. This is at odds with the basic principle of serialization being a process solely controlled by serialization parameters, without a need to interfere with the information content of the node tree.

UDL does not *map* XML nodes to JSON structures. It redefines JSON to *be* a representation of nodes. As a consequence, it need not "inject" any ad hoc items into the data tree for the sake of controlling a serialization to JSON. None of the reserved element and attribute names in Table 2, “

***Names in the `udl` namespace and their usage.***

” have anything to do with JSON or serialization. Rather, they represent standard names without semantics, reflect node properties or signal the markup language currently used within a well-defined scope. The extended node model is expressive enough to represent JSON structures natively.

The main difference between UDL and mapping approaches concerns the handling of JSON names. Mapping approaches represent JSON names as element names if possible, and if not, resort to one of two possible solutions: either place the JSON name in an additional attribute (e.g. Pemberton), or define a bidirectional name mapping (e.g. BaseX). But there are three differences between the concepts of JSON names and XML names:

- a JSON name has no namespace component
- a JSON name can use arbitrary characters
- a JSON name must be unique amongst the JSON names of all siblings

Note that the last point (the uniqueness constraint) means that a JSON name resembles an `xml:id` attribute more than an element name. It can be compared to a locally scoped `xml:id` attribute (uniqueness among all element children of an element). For these reasons UDL distinguishes the concepts of names and keys. It thus enables native relationships between nodes and XML markup on the one hand and JSON markup on the other hand. As a result it becomes possible to regard JSON markup and XML markup as alternative representations of an information content which is defined in terms of nodes and their properties. Remembering Plato, one kind of “thing” is inferred from - or may cast - two different "shadows".

Should we not keep things simple - do we *need* to extend the document model and introduce new node properties? Imagine this alternative:

- the W3C publishes a small specification defining a standardized bi-directional mapping between arbitrary strings and QNames
- the XPath language syntax is slightly extended, introducing a second notation of a name test (e.g. `#foo`) which is interpreted as a string which is *automatically mapped* to a QName according to the standard name mapping:

```
a/b/#c       equivalent to:   a/b/c
```

```
a/b/#c_d      equivalent to:   a/b/c__D
a/b/#1        equivalent to:   a/b/_1
a/b/#"1 2"    equivalent to:   a/b/_1_00322
```

This is an attractive scenario: one can formulate the XPath expressions without a mental translation, just using the names one *sees* in the source data. The net result is an elegant approach to the processing of JSON data with XML tools.

The approach would be a good – and perhaps a better – alternative to UDL if the processing of JSON documents with XML tools were the only goal. However, UDL's design aims at a unified document model which expresses the entities represented by dominant markup languages in a native way. Only this way can information content and representation (markup) be decoupled and can the latter be switched easily at various scales (whole documents, sections or single elements) and in various contexts (data and program code).

# JSONiq

JSONiq [JSONiq], [JSONiq Specification] is an extension of the XQuery language designed to add support for JSON data. Like UDL, JSONiq extends the XDM in order to accommodate JSON structures. However, JSONiq does not change the node sub model of the XDM. Rather, two new item types are introduced, designed to represent JSON data:

- object
- array

It is interesting to note the parallel: both, JSONiq and UDL extend the XDM in order to accommodate JSON data; but the changes UDL proposes are *within* the node sub model, whereas JSONiq adds a second sub model for structured data, in parallel to the node model. Using the terms proposed in the section called “Distinction between markup and document language”: JSONiq keeps the XML document language, but extends the XML information language, whereas UDL shifts the changes into the very document language, refraining from changes outside of the node model.

An advantage UDL offers is to represent JSON data as node trees and thus expose them to XPath navigation. JSONiq, on the other hand, might be easier to accept exactly because it does not introduce any change to the document language and therefore restricts itself to the extension of a query language, rather than an extension of XML.

# Map items

The W3C XSL Working Group has made a proposal for extending the XDM by a new item type: map items [W3C XSLT 3.0]. They represent generic containers, but nevertheless can represent JSON data. One should note the relationship between JSONiq and the map proposal: both approaches mandate new item types which are not nodes and yet can represent structured data; but one (JSONiq) resorts to JSON-specific items, whereas the other advocates generic containers.

Not being nodes, map items are lightweight containers which can collect items without requiring or imposing a structural relationship. Therefore node relationships between container and members are not possible. This contrasts sharply with the UDL approach which models JSON containers and their members as nodes and their child nodes. Only this way can JSON data be seamlessly integrated into the navigational system based on axes and node tests.

This is not to say that such lightweight containers would not be very useful extensions of XDM. Lightweight containers and nodes cannot replace each other. A node model is required for the navigational power of XPath. Lightweight containers are required to model node relationships independently from their structural relationship. And among other benefits they enable a mapping of keys to node references, rather than the nodes themselves, which is a highly desirable feature.

It is interesting to note a conceptual relationship between UDL and the "map proposal": the content of an element with [model] equal "map" can be described as a map item constrained in the following way: (a) every map value is a child element; (b) every map key is the [key] of the associated value.

# Discussion

There is a growing awareness in the XML community that other markup languages do, will and should coexist with XML. So integration is a crucial task. Being determined to integrate, one may look at XML - as well as other markup languages - as both: markup, and information content represented by markup.

We happen to be in the possession of a rigorous, formal definition of the information content of XML data. If this model is only approximately, but not quite capable of expressing what new markup languages have to say (compare the section called “ The mapping approach ”), this may reflect the circumstances when those formal definitions were set down: a point in time when XML structure was the only thing that had to be expressed. But *now* it seems a natural course to consider extending the model cautiously, turning it into a unified document language. When infoset and XDM became recommendations, there was nothing to unify, now there is.

UDL might change our perception of markup languages: they are freely exchangeable in various contexts - both in data resources and in program code (within XQuery and XSLT constructors) - and at various scales - whole document, document section, single element. This becomes possible when different markup is seen as alternative representation of unified content.

An evaluation of the UDL proposal may profit from making two distinctions. The first distinction is between UDL's central idea and its translation into technical details. The idea is to relate multiple markup languages to a single, unified node model, which turns XML processing technologies into general information processing technologies. For this purpose, the node model was extended in a particular way (e.g. adding a new node property, [key]). Doubtless, other approaches how to extend the node model are conceivable, too. An evaluation of the UDL proposal might modify or even replace the model extensions by alternatives, preserving the central idea as such.

The second distinction is between what UDL does achieve and what is deliberately left to a future extension (or to complementary components). UDL does not yet offer support for certain transformations ("mappings") which are acknowledged to be important in the context of markup integration. In other words: the existence of a unified document language does not yet mean comprehensive support for all use cases in the context of integrating multiple markup languages. An evaluation of the UDL proposal should regard the unified document language as a *foundation* for mapping support - not as a substitute.

# A. Deserializing from / serializing to JSON

This appendix contains the precise rules how to deserialize a JSON document to a tree of nodes and how to serialize a tree of nodes to a JSON document.

## Deserialization

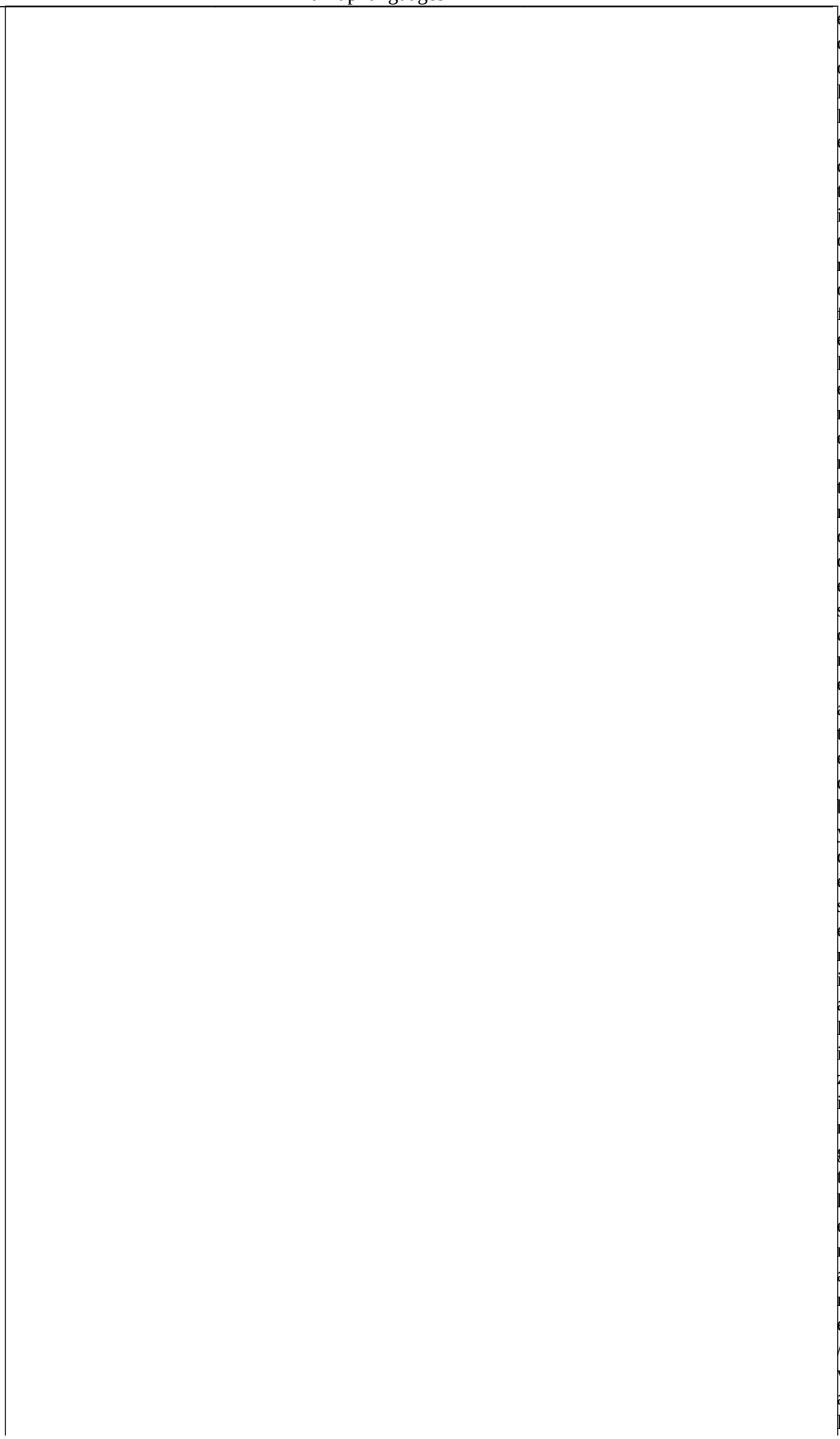
| **JSON null** => node | |
|---|---|
| property | value |
| name | ud |

| | | |
|---|---|---|
| | key | l:nulli ftheenullistheval ueofaname/valuepair_thename;emptyl |

| | |
|---|---|
| model | otherwise sequence |
| nilled | true |
| schema type | xs:untyped |

| **JSON object** => node | |
|---|---|
| property | value |
| name | udl:map |
| key | if the obj |

| | | |
|---|---|---|
| | | this value of a name/value pair – the name; empty, otherwise |
| | model | map |
| | content | th |


ecollectionofelementnodescreatedbydeserializingthenamee/val

| | |
|---|---|
| schema type | upairs xs:untyped |

| | |
|---|---|
| **JSON array** => node | |
| property | value udl:array |
| name | |
| key | if the array is the value of a |

| | | |
|---|---|---|
| | | name/value pair – the name; empty, otherwise sequence the collecti |
| | model | |
| | content | |

| | | |
|---|---|---|
| | schema type | onoffelementnodescreatedbydeserializingthearraymembersxs:u |

| | |
|---|---|
| | ntyped |
| **JSON simple value** => node | |
| property | value |
| name | udl:value |
| key | if the value is part of a name/value pair– |

| | | |
|---|---|---|
| | | the name; empty, otherwise sequence if the value is not a zero-length |
| | model | |
| | content | |

string: a text node containing the text representation of the value

| | | |
|---|---|---|
| | | ; empty content, otherwise |
| | schema type | • if the JSON value is a number: xs:integer |

/xs:decimal/xs:double_dependingonthelexicalformifthe JSON

value is one of the constants true or false: xs:boolean

- if the JSONv

alue is a zero-length string: xs:untypedAtomic
otherwise: xs:unt

| |
|---|
| |

y
p
e
d

Note: The XML representation of a zero-length JSON string is an empty element with type annotation `xs:untypedAtomic`. The type annotation makes the node distinguishable from empty elements corresponding to empty arrays or objects.

# Serialization

***Handling attributes***

If attributes are encountered, the behaviour depends on the serialization parameter `info-loss`: if the value is `json.projection`, the attributes are ignored; otherwise, a non-recoverable error is raised.

***Handling of processing instructions and comments***

Processing instructions and comments are ignored.

***Handling elements***

The handling of elements depends on various properties:

- the [nilled] property
- the [model] property
- the presence of element children
- the presence of text node children
- the [schema-type] property

The following table shows all details.

### Table A.1.

### *Serialization - translating UDL nodes into JSON items.*

| UDL element node | JSON item |
|---|---|
| [nilled] is `true` | If serialization parameter `info-loss` is `json.strict` and the element name is not `udl:null`, a non-recoverable error is raised. Otherwise the element is serialized as a JSON null value. |
| [model] = "map" | If serialization parameter `info-loss` is `json.strict` and the element name is not `udl:map`, a non-re- |

| UDL element node | JSON item |
| --- | --- |
| | coverable error is raised. Otherwise the element is serialized as a JSON object. The contained name/value pairs are obtained by serializing the element children. An error is raised if the element has a text node child with non-whitespace content. |
| [model] = "sequence";<br><br>at least one element child | If serialization parameter `info-loss` is `json.strict` and the element name is not `udl:array`, a non-recoverable error is raised. Otherwise the element is serialized as a JSON array. The array members are obtained by serializing the element children. An error is raised if the element has a text node child with non-whitespace content. |
| [model] = "sequence";<br><br>no element children;<br><br>at least one text node child | If serialization parameter `info-loss` is `json.strict` and the element name is not `udl:value`, a non-recoverable error is raised. Otherwise, the element is serialized as a JSON simple value. The string values of the text nodes are concatenated and the result is used to construct a simple JSON value whose type depends on the node's [schema-type]: number (if [schema-type] is equal to or derived from xs:double or xs:decimal), Boolean (if [schema-type] is equal to or derived from xs:boolean) or a string (otherwise). |

| UDL element node | JSON item |
|---|---|
| [model] = "sequence";<br>no element children;<br>no text node child;<br>[schema-type] is an atomic type | If serialization parameter `info-loss` is `json.strict` and the element name is not `udl:value`, a non-recoverable error is raised. Otherwise, the element is serialized as a JSON string value of zero length. |
| [model] = "sequence";<br>no element children;<br>no text node child;<br>[schema-type] is not atomic | If serialization parameter `info-loss` is `json.strict` and the element name is not `udl:array`, a non-recoverable error is raised. Otherwise, the element is serialized as an empty JSON array. |

# B. Additional support for "NCName-only JSON"

This appendix describes an extension of the UDL proposal which provides additional support for processing JSON documents in which every name is an NCName. This extension is presented as an appendix as it has a more tentative character than the core parts of the proposal and is more likely to be modified, removed or replaced by alternatives, should the UDL proposal be evaluated as a whole.

## Introduction

UDL does not map JSON documents to XML documents, but defines the information content of a JSON document as an UDL node tree. Such a node tree can be serialized as both, JSON or XML. The XML representation of a JSON document is ugly and not meant for human consumption. The UDL proposal regards the readability of this XML representation as a non-goal and concentrates on the node tree which is designed to enable a JSON processing as powerful and elegant as XML processing. The developer is expected to design his JSON processing code while regarding the JSON serialization, not the XML serialization.

The poor readability of XML-encoded JSON documents is caused by the fact than JSON names are captured as node keys, rather than node names. The [key] property was introduced because the alternative approach of regarding JSON names as node names would introduce a dependence of the node model's name representation on whether the JSON name happens to be an NCName: NCNames are preserved, and non-NCNames are changed into the result of a name mapping which is either generic and non-semantic (e.g. "2012" to "_2012") or semantic and application specific (e.g. "2012" to "year-2012"). A semantic mapping is certainly an interesting solution in many situations (compare for example [Lee]), but it gives up the advantages of a unified document language. The purely technical mapping which replaces inacceptable characters, on the other hand, is unnatural, as it amounts to the perspective that "_2012" is the essential information content, whereas "2012" is some deviating representation.

# Definition of UDL document styles: nJSON, nnJSON

Nevertheless it cannot be denied that in some situations one would like to have available both, a readable JSON representation and a readable XML representation. For example, an increasing number of web services is expected to deliver both, XML and JSON, at user option. In this scenario, UDL's XML representation of a JSON document is downright inappropriate. What is needed is an XML representation whose element names mirror as good as possible the JSON names. If the JSON document contains non-NCName names, the mapping problem arises - but what if all JSON names *are* NCNames? Let us define an **nJSON** document as a JSON document in which all names are NCNames. An XML representation of an nJSON document may then reuse the JSON names as element names (the local part of it), and yet represent a JSON document without any ambiguity: JSON-serialization using the serialization parameter `info-loss` with a value `json.ignore-names` will yield the same document as the strict JSON-serialization of the counterpart which sticks to unspecific names. Let us further introduce the notion of **nnJSON** documents defined as follows:

1. every name is an NCName (indicated by the first "n")

2. every element with a key has a local name equal to the key (the second "n")

3. the document is JSON-serializable using `json.ignore-names` (the "JSON")

Note that (3) implies further constraints: no attributes and no mixed content. At the same time this definition leaves considerable freedom: namespaces and the names of key-less elements can be chosen arbitrarily. nnJSON documents can be regarded as augmented nJSON documents – the additional information consisting of element names which can later be used or discarded, dependent on purpose.

nnJSON documents have a remarkable property: they represent an unambiguously determined nJSON document, to whose JSON text they can be serialized, using `json.ignore-names`; and they can also be serialized to a well-readable XML representation of that JSON document. When dealing with nJSON documents, nnJSON can be used as a normalization of information which enables unified processing code: code that is used no matter if the input is JSON or XML and whether the output is JSON or XML. This unified code consumes an nnJSON tree and it produces an nnJSON tree. The UDL extensions discussed so far ensure that the nnJSON output can be alternatively serialized as readable XML or nJSON. The extensions do however not enable the parsing of both, nJSON text (JSON) and nnJSON text (XML) into an nnJSON tree. After all, the information content of nJSON and nnJSON is different and parsing by definition does not change the information content: parsing alone will always produce one kind of tree or the other. The processing pattern just sketched – "read and write nnJSON" - therefore has to rely on a translation of an nJSON text or node tree into an nnJSON node tree.

# Special support for the processing of nJSON documents – a further extension of XPath

After parsing an nJSON document (`doc("foo.json")`) it can easily be transformed into an nnJSON document, e.g. with a simple stylesheet. nJSON documents are however so important that they warrant a built-in support supplied by the UDL extensions. Therefore the present proposal adds a special-purpose-function which combines the JSON parsing and its transformation to an equivalent nnJSON document:

```
nnjson($uri as xs:anyURI) as document-node()
```

Further signatures allow for control of several aspects of the result document which are not constrained by the definition of nnJSON documents. These are:

1. the element namespaces

2. the node name of the root element

3. the node names of other elements without a key

Consider an example. Let the following nJSON document be a response to a “getWeather” service request:

```
  {
     "date" : "2012-08-06",
     "place" : " London",
     "temperatures" : ["12", "21"]
  }
```

Here comes a matching nnJSON document:

```
  <getWeatherRS xmlns="http://example.com" udl:model="map">
   <date>2012-08-06</date>
   <place>London</place>
   <temperatures>
      <t>12</t>
      <t>21</t>
   </temperatures>
</getWeatherRS>
```

This document looks as a fairly natural representation of the original JSON document, and it can be serialized to the original JSON document using `json.ignore-names`. Note the use of an arbitrary namespace and the choice of intuitive element names for key-less elements. A second signature of the `nnjson` function enables control of these customizations:

```
  nnjson($uri as xs:anyURI,
         $namespace as xs:anyURI?,
         $rootName as xs:string,
         $patternsAndNames as item()*)
```

The `patternsAndNames` parameter expects an alternating sequence of XSLT pattern values and an element name; when renaming a key-less element, the first matching pattern is located and the name is taken from the item following the pattern item. Our example could be produced by the following call:

```
   nnjson("rsp.json",
          "http://example.com",
          "getWeatherRS",
          ("#temperatures/*", "t")
         )
```

Using the simple signature without control paramters, on the other hand:

```
nnjson("rsp.json")
```

produces a document without namespace and with some unspecific element names:

```
<udl:map udl:model="map">
   <date>2012-08-06</date>
   <place>London</place>
   <temperatures>
```

```
            <udl:value>12</udl:value>
            <udl:value>21</udl:value>
         </udl:temperatures>
      </udl:map>
```

The `nnjson` function is a convenience function which combines the parsing of an nJSON document with a transformation of particular interest. The transformation is defined in such a way that the changes of information content do not interfere with a subsequent JSON-serialization (using `json.ignore-names`). This curious mixture of parsing and transformation is regarded as a first-class operation deserving a built-in XPath function because of a well-defined relationship between the resulting XML document and the original JSON document.

# Bibliography

[BaseX] Gruen, Christian, et al. BaseX Documentation Version 7.2, section "JSON Module", p. 125-127. http://docs.basex.org/wiki/Main_Page.

[Couthures] Couthures, Alain. JSON for XForms - adding JSON support in XForms data instances. XML Prague 2011, Conference Proceedings, p. 13-24. http://www.xmlprague.cz/2011/files/xmlprague-2011-proceedings.pdf.

[JSON] Web resource without source information: Introducing JSON. http://json.org.

[JSONiq] Robie, Jonathan, Mathias Brantner, Daniela Florescu, Ghislain Fourny and Till Westmann. JSONiq - XQuery for JSON, JSON for XQuery. XML Prague 2012, Conference Proceedings, p. 63-72. http://www.xmlprague.cz/2012/files/xmlprague-2012-proceedings.pdf.

[JSONiq Specification] Robie, Jonathan, Mathias Brantner, Daniela Florescu, Ghislain Fourny and Till Westmann. JSONiq: Language Specification. http://jsoniq.com/docs/spec/en-US/html/index.html.

[Hunter] Hunter, Jason. A JSON facade on MarkLogic Server. XML Prague 2011, Conference Proceedings, p. 25-34. http://www.xmlprague.cz/2011/files/xmlprague-2011-proceedings.pdf.

[JsonML] Web resource without source information: JSON Markup Language (JsonML). http://www.jsonml.org/.

[Lee] Lee, David A. JXON: an Architecture for Schema and Annotation Driven JSON/XML Bidirectional Transformations. Presented at Balisage: The Markup Conference 2011, Montréal, Canada, August 2 - 5, 2011. In Proceedings of Balisage: The Markup Conference 2011. Balisage Series on Markup Technologies, vol. 7 (2011). doi:10.4242/BalisageVol7.Lee01. http://www.balisage.net/Proceedings/vol7/html/Lee01/BalisageVol7-Lee01.html.

[Pemberton] Pemberton, Steven. Treating JSON as a subset of XML. XML Prague 2012, Conference Proceedings, p. 81-90. http://www.xmlprague.cz/2012/files/xmlprague-2012-proceedings.pdf.

[Robie] Robie, Jonathan. A universal markup language and a universal query language. A contribution to the discussion of Google group JSONiq, 18 October 2011. http://groups.google.com/group/jsoniq.

[Tennison] Tennison, Jeni. Opening keynote - collisions, chimera and consonance in web content. A presentation at xmlprague 2012. http://www.slideshare.net/JeniT/collisions-chimera-and-consonance-in-web-content.

[W3C Information Set] John Cowan and Richard Tobin, eds. XML Information Set. W3C Recommendation 4 February 2004. http://www.w3.org/TR/xml-infoset/.

[W3C XDM] Mary Fernandez et al, eds. XQuery 1.0 and XPath 2.0 Data Model (XDM). W3C Recommendation 23 January 2007. http://www.w3.org/TR/xpath-datamodel/.

[W3C XDM 3.0] Norman Walsh et al, eds. XQuery and XPath Data Model 3.0. W3C Working Draft 14 June 2011. http://www.w3.org/TR/xpath-datamodel-30/.

[W3C XML] Tim Bray et al, eds. Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C Recommendation 26 November 2008. http://www.w3.org/TR/REC-xml/.

[W3C XSLT 3.0] Michael Kay, ed. XSL Transformations (XSLT) Version 3.0. W3C Working Draft 10 July 2012. http://www.w3.org/TR/xslt-30/.