The XML info space

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

Abstract

XML-related standards imply an architecture of distributed information which integrates all accessible XML resources into a coherent whole. Attempting to capture the key properties of this architecture, the concept of an info space is defined. The concept is used as a tool for deriving desirable extensions of the standards. The proposed extensions aim at a fuller realization of the potential offered by the architecture. Main aspects are better support for resource discovery and the integration of non-XML resources. If not adopted by standards, the extensions may also be emulated by application-level design patterns and product-specific features. Knowledge of them might therefore be of immediate interest to application developers and product designers.

Table of Contents

Introduction

The first version of a successful program is not the last one. But the more complex the program, the more care is required when making changes. When technologies and the underlying standards are concerned, any changes must certainly be based on a deep understanding of the status quo. But the nature of the required understanding is not obvious.

In 1938, Albert Einstein and Leopold Infeld published a book about the evolution of physics [10]. The focus was on the crucial role played by named *concepts*, guiding and at times limiting our thoughts and understanding. A striking example involves the concepts of "force" and "field". At a point in time, the scientific understanding of forces was a sort of interaction between two bodies. Then the field was discovered as a new reality, existing independently of the presence and absence of bodies. Without this image of a field, understanding simply could not have passed beyond a certain point.

In information technology, named concepts can play an important role in the development of industry standards. Fielding [11] gave a detailed account how the abstractions of a "resource" and "resource representations" enabled a unified vision of the emerging internet and led to the definition of an architectural style (REST - Representational State Transfer) which provided guidance in the evaluation of proposed changes to the standards.

How about XML – is it final or in need of a new version? Is there a direction to go? This paper proposes a novel abstraction to guide a better understanding of the technological reality already created, an abstraction intended to help in the evaluation of further steps. The reality implemented by XML technology is very versatile, and doubtless many conceptualizations are useful and important. Nevertheless, this paper suggests a new concept or image - the info space. The concept evokes a potential which is at the same time so large and so much limited by the current state of affairs that the perspective of a fuller realization may affect our sense of priorities. The remaining sections define the concept, explore the current limitations and suggest steps towards an increasing realization of its potential.

Experiments and observations

To get started, we examine a few simple programming tasks and possible solutions using XPath. The experience will invite some observations, which we then attempt to translate into a concept characterizing the new situation which XML technology has created.

Experiments

Assume we have an XML file (`foo.xml`) and want to retrieve all `countryCode` elements, no matter where they occur in the document. The following XPath expression does just that:

```
doc("foo.xml")//countryCode
```

Now we want to detect all `countryCode` instances that do not contain a valid two-letter code according to ISO 3166. As a code dictionary in XML format is available in the internet[1] we can solve the problem by adding a simple filter expression:

```
doc("foo.xml")//countryCode[not(. = doc("http://...")//code]
```

When the check should be applied to several files, we only have to replace the first step by an expression which yields several documents, rather than a single document:

```
(doc("foo.xml"), doc("foo2.xml"))
              //countryCode[not(. = doc("http://...")//code]
```

As there may be many documents to be checked, we introduce a helper resource, an XML document listing all documents concerned. The list represents each document by a `uri` element containing the document URI, which may be a file name or an HTTP address. In order to apply the check to all documents found in the document list, again we adapt the first step of our navigation:

```
doc("docs.xml")//uri/doc(.)
              //countryCode[not(. = doc("http://...")//code]
```

The documents may be found in the file system, an intranet or the internet, and their number could be large. If a specific document list has to be created for our checking task, this might become inconvenient, too. So we introduce a general purpose inventory of document URIs which describes a whole domain of documents, only some of which have to be checked in this particular case. We create a tree-structured document list in which only the leaf elements are `uri` elements containing a document URI, whereas all inner nodes have the purpose of adding structure, implicitly creating groups of documents. All elements - inner nodes and `uri` elements - may have attributes supplying metadata pertaining to the document(s) referenced by the element itself or its descendants. Schematically, such an inventory might look like this:

```
<doctree>
   <department name="d1">
```

[1] http://www.iso.org/iso/home/standards/country_codes/country_names_and_code_elements_xml.htm

```
      <project name="p1">
         <xsd>
            <uri app="a1">…</uri>
            <uri app="a2">…</uri>
         </xsd>
         <msg>…
            <uri srv="s1">…</uri>
            <uri srv="s2">…</uri>
         </msg>…
      </project>
   </department>
</doctree>
```

Now we can combine the selection of documents and their checking into a single expression:

```
doc("doctree.xml")//project[@name="p1"]/msg/uri/doc(.)/
              //countryCode[not(. = doc("http://...")//code]
```

A navigation across multiple resources - an inventory, all resources referenced by `uri` elements which match certain conditions, and finally the code dictionary – is achieved by a single expression, without taking actions like opening a file and without shifting information from data sources into intermediate variables.

The expression does not only yield the faulty codes; it resolves to the *nodes* containing them, which means information *and* its location. Assigning the nodes to an XQuery variable (`$faultyCodes`), we can later resume navigation, using those nodes as starting points. If the documents contain somewhere a `changeLog` element, we can collect the change logs of all faulty documents:

```
$faultyCodes/ancestor::document-node()//changeLog
```

Observations

This little programming experience may give a feeling of handling information in a different way than when using a general purpose language like Java or Perl. What was different?

First, we did not "load" resources - we just *addressed* them using an expression which was equivalent to the complete information content. Compare this to Java code extracting data from a CSV file. The equivalence enables us to to use the expression as input to navigation:

```
doc("…")//countryCode
```

Second, navigation can be downward (e.g. "…//..") or upward (e.g. "ancestor::...") in a *vertically unbounded* way, moving from any current location in a single bout down to the very leaves under it and up to the very root above it. Compare this to the navigation in an object tree: a downward move can only be achieved recursively, one level per invocation, and upward navigation is impossible.

Third, *resource boundaries* do not impose a resistence to navigation: the effort to enter a different document is not greater than the effort to move within the same document. The following expression moves down to `uri` elements, crosses over into the referenced documents and continues its path down into that document:

```
//uri/doc(.)//countryCode
```

Fourth, navigation is a *parallel operation*, multiply rooted and cascading: the starting point may be one or many locations, and subsequent steps resume the movement at every location reached by the preceding step. This is different from navigation of an object tree, which is a movement from single item to single item.

Fifth, navigation is a *generic operation*: the inventory, the resources to be checked and the code dictionary have different semantics, but they are navigated in the same way. Navigation of an object tree, on the other hand, must adapt each single step to object types.

Summarizing the observations, one perceives a continuous space of information: we can enter and leave resources at will, and within them we can move up, down, forward and backward in a uniform and effortless way. The space is a sum total of information which integrates every single accessible XML document. Within this space, every item of information is visible and addressable.

The XML info space

This section proposes the notion of an **info space** as an architecture of distributed information which comprises two models, describing content and navigation. The definition of the space is guided by the reality introduced by XML technology but, being an abstraction, does in no way depend on XML as its implementation. Let us summarize some important properties:

The definition

- Space content is decomposable into information items
  - The space is an unstructured collection of resources
  - A resource is a structured collection of items
- Inter-item relationships are well defined
  - Items within a resource are related by axes
  - Items in different resources are related by URI+axes
- Navigation is item-oriented
  - Input, output, intermediaries are sets of items
  - Input, output, intermediaries can reside in multiple resources
- Navigation is generic
  - The specification of a movement can be separated from the choice of start items
  - Any specification of a movement can be applied to any choice of start items
- Information is well defined
  - Information is a sequence of items
  - An item is a component with properties
  - Semantics and value space of item properties are well defined
- Information is expressible
  - Information is expressible as navigation leading to it

It is easy to see how all properties are implemented by current XML technology. The resources in question are XML documents [6], their itemized structure is defined by the node model part of the XDM [2] (which is more suitable as a substrate for navigation than the info set model [8]), intra-resource relationships are defined by XPath axes [1] and inter-resource relationships by the URI standard [3]. The navigation model is provided by XPath [1] and XQuery [4] and the information model by the XDM in its entirety [2].

The limitations

The URIs of all accessible documents are an unstructured collection of strings. There is no built-in way how to navigate this collection, iterate over its members or in some other way discover URIs. Document access simply requires beforehand knowledge of a document URI. If relying on standard XPath functions only, even file directories cannot be inspected in order to discover the files it contains. To throw light on this limitation, think of a stock of information distributed over thousands of XML documents which are accessible either as local files or via HTTP address. While on the one hand it is a fascinating thought that anything within that pool of documents can be immediately *addressed*, it can become a serious problem how to *discover* the resources: to find out which URI to use. If we compared XML documents to relational table rows, the info space would resemble a relational database in which `select` commands are constrained to specify a primary key.

The boundaries

Given a computing site - which part of the resources of the world are in the info space, and which are not? Any resource not accessible via network or file system is obviously not within the info space: this is the *physical boundary*. But also when physical access is possible, the protocol requirements may not be supported by XPath functions providing document access (namely the `fn:doc` function), for example due to the need of supplying authorization information or an HTTP body: this is the *API boundary*. Those resources which are accessible both physically and API-wise can be grouped into two categories: XML documents and other resources. The former are in the XML info space, the latter are not: the *media type boundary* excludes any non-XML resources.

Addressing the limitations of the info space

XPath navigation has a strong part and a weak part. Within-resource movement is governed by a powerful model – the application of semantic filters (name test and predicates) to structural relationships (navigational axes). Between-resource movement, on the other hand, is mainly enabled by the `fn:doc` function, which resolves a URI to the corresponding document. This is a lookup requiring beforehand knowledge of the key – neither structural nor semantic conditions can be specified. Regarded as a means of navigation, the `fn:doc` function produces “manifold motion making little speed” (S. T. Coleridge, “The butterfly”), as it delivers only one result item per item of knowledge. The `fn:collection` function offers multiple items for the price of a single URI, but the collection is a pre-defined unit, which often will not match the actual requirements.

These limitations reflect the nature of the info space, which is an unstructured set of unrelated, indistinguishable trees. Or more precisely: trees that cannot be distinguished or related until an in-memory representation is constructed, as URIs are devoid of semantics. The info space does not relate resource URIs to a structure (like directories), does not define inter-resource relationships (like XLink links) and does not provide any way how to search for resources (as a database does).

Space structure, resource relationships and searchability may, however, be introduced on the grounds of *data* which create a second layer of information, built upon the first layer consisting of the resources themselves. Think of a linkbase defining relationships between documents, or a data set associating URIs with meta information. Data-based navigation aids can be introduced on the application level. But they might also be supported by standardization. The basic principle would be to *extend the dynamic context* of XPath evaluation by novel components which expose those data to the navigational engine of XPath - enabling novel XPath functions, perhaps even novel navigational axes.

This section explores some possibilities how standards may be extended in order to enhance the possibilities of between-resource navigation.

XLink per XPath

XLink [9] is a specification that defines XML syntax for describing links between resources. Although these resources are not necessarily nodes, we can for the sake of our discussion treat XLink as a language for defining node relationships. The basic unit of resource linkage is an *arc* which maps one set of resources (the starting resources) to another set of resources (the ending resources). The arc enables “traversal” from any one of the starting resources to any one of the ending resources. This concept of XLink traversal is similar, though not identical, to XPath navigation. Both rely on a unidirectional mapping from a start or context node to a set of result nodes. Consequently, the definition of XLink arcs might be viewed as the definition of potential navigation steps available to XPath. Let us investigate possibilities to translate XLink-defined arcs into XPath steps.

XLink defines containers, called extended links, which hold together a set of resource locators and associated arcs, which are mappings between subsets of the complete set. (For the sake of clarity, we ignore the possibility of using local resources instead of or together with resource locators.) The extended link can assign to each of its resources an optional label (NCName). The label is like a property attached to the resources which allows them to be identified and grouped into subsets of the complete set contained by the extended link. The start and the end of an arc is defined in terms of an optional label value: if a label value is set, the corresponding subset of resources comprises all resources which have this label; if no label value is set, the subset is equal to the complete set. Consider the following extended link containing four arcs:

```
<foo xlink:type='extended'>
   <a1 xlink:type='arc' xlink:from='a' xlink:to='b'/>
   <a2 xlink:type='arc' xlink:to='b'/>
   <a3 xlink:type='arc' xlink:from='a'/>
   <a4 xlink:type='arc' />

   <res xlink:type='locator' xlink:href='…1' xlink:label='a'/>
   <res xlink:type='locator' xlink:href='…2' xlink:label='b'/>
   <res xlink:type='locator' xlink:href='…3' xlink:label='b'/>
   <res xlink:type='locator' xlink:href='…4' xlink:label='b'/>
   <res xlink:type='locator' xlink:href='…5' xlink:label='c'/>
</foo>
```

The arc <a1> leads from the single one resource associated with a label “a” to all three resources with a label “b”; <a2> leads from any one of the five resources to the three resources with a label “b”; <a3> leads from the resource with a label “a” to all five resources, including itself; and <a4> leads from any one of the five resources to all five resources, including itself.

An arc is an option for navigation. Given a node (N), a navigation step may be specified by *selecting a particular arc* (A). The destination is the collection of nodes which

- is empty if N is not among the starting resources of A
- comprises the ending resources of A, otherwise

Such navigation might be integrated into XPath if two changes are applied to the current standards:

- extend the static context of XPath by a component **Available extended links**, comprising the extended links which are “in scope”. (These extended links may be imported as linkbase documents, analogously to the import of in-scope schemas as schema documents.)
- define a new kind of navigation step which is based on the selection of a particular arc

The new kind of navigation step may be a combination of an `xlink` axis with an “arc test” selecting arcs. The expression result could then be collected from the ending resources of the selected arcs.

How to specify the selection of arcs? The XLink standard defines extended links to have an optional `role` property and arcs to have an optional `arcrole` property, where both properties have as value a URI. Therefore a selection of arcs might be specified by two URIs, a `role` selecting extended links and an `arcrole` selecting arcs within those links. If we now deviate from the XLink standard and assume that arcs have an additional property, an "`arclabel`" with an NCName as value, then arc selection might be specified by a single QName, as a QName consists of a URI plus an NCName. The "arc test" selecting an arc might now be specified by a QName, in analogy to the name test which is also specified by a QName, but selects elements or attributes . Consider the following expression:

```
xlink::x:y
```

Its value would be determined as follows.

1. select from the **Available extended links** those with a `role` equal to the URI bound to prefix "x"
2. within the selected extended links select all arcs with an `arclabel` equal to "y"
3. filter the selected arcs, retaining only those whose set of starting resources "matches" the context node (see below for details)
4. obtain as the expression value the union of the ending resources of all arcs retained by the previous step

A remaining issue is the definition of when a set of starting resources "matches" the context node. The proposal is to define matching as containment: the starting resources include a node which is either identical to or contains the context node. Consider this expression:

```
doc('foo.xml')//a/xlink::x:y
```

The selected arcs may include as starting resource either the document "foo.xml", or the "a" element to which the `xlink` step is applied, or any ancestor of that "a" element.

As an alternative to using a new `xlink` axis, XLink navigation might be supported by a new standard function, `fn:xlink`. In this case, the set of extended links to be considered might alternatively be taken from the static context or from linkbase documents supplied as a parameter:

```
fn:xlink($arcTest as xs:QName) as node()*
fn:xlink($arcTest as xs:QName, $linkbases as node()*) as node()*
```

XLink versus data-driven navigation

XLink defines resource relationships in terms of URI relationships - static mappings between single URIs and/or URI collections. The XLink model does not cover well a frequent requirement – data-driven navigation, which is guided by data and unawares of structural relationships between a current context and the destination. For example, assume a document D containing a country code, and further assume a collection of documents describing countries, one country per document: how can the evaluation of D take advantage of the country description matching the country code?

Currently, the navigation model of XPath does not offer standardized support for such tasks. Working within the bounds of a single document, this is felt less acutely as expressions like `//country[@code eq 'FR']` have a sweeping nature which removes any structural assumptions apart from which document the information is located in. But when the target might be located in other documents, the lack of support may become an issue. Navigation between documents requires knowledge of a URI. In the general case this will be the target document URI, as support for named collections (the `fn:collection` function) is implementation-defined and hence not portable.

Without such support we simply cannot “look into” other documents without first providing a document URI.

If the processor does support the `fn:collection` function, we can access documents without knowledge of individual document URIs. But collections are a problematic answer to the challenge of data-driven navigation. The collection is not yet the destination of our navigation, but a set of candidates from which to select the destination. In order to navigate into one country description, two hundred country descriptions have to be parsed (unless they are located in an XML database) and queried (unless we can resort to a database index). The greater the collection, the greater the overhead incurred by processing the collection in order to identify the one or few matching items. When there are a large number of candidates – e.g. accumulated log messages – the approach of constructing all candidate nodes and then filtering them can quickly become unfeasible.

The problem is deep-rooted. As navigational axes are bounded by the document border, conventional navigation cannot lead to the discovery of information in other documents, unless the URIs of those documents are either known beforehand or supplied by a document visited at an earlier navigation step. The next section develops a concept of data-driven navigation which is based on a filtering of collections which does not require the prior in-memory construction of its nodes.

Property-faced collections

External node properties

Every XML node has a set of properties defined by the XDM data model. An element node, for example, has twelve properties, which include the name, relationships to other nodes ([parent], [children], [attributes]), a type, a typed value and a string value. Evaluation of these properties is what drives XPath navigation, guiding it from a set of starting points to a set of destination nodes.

In certain situations, however, the locating of nodes might be greatly simplified if we manage to associate nodes with *additional* properties which help us to identify the nodes of interest. Such properties could represent metadata, or they could reflect the contained data, for example echoing a value found at a particular data path within the node, or providing a value derived from contained data, like a minimum or maximum. As these additional properties are not XDM properties, their values are not part of the node data. Hence we call them **external node properties**. Their nature implies an interesting possibility: they can provide information about a node which is *accessible without constructing the node*. Access to such information might enhance the navigational model of XPath in general, and add support for data-driven navigation in particular. The following sections describe an approach how to achieve this.

An external property has a name and a value. A set of external node properties can be modelled as a map with which the node is associated. The map entries have a key which represents the property name, and a value which represents the property value. Such a map we call a *property face* of the node, or **p-face** for short. Striving to keep things simple, we constrain external node properties to have an NCName and a value which is a sequence of atomic items of type `xs:untypedAtomic`. The value space of p-faces is accordingly constrained.

Property-faced collections

A p-face is not part of the node data – it is merely *associated* with them. This relationship can be compared to the general relationship between the keys and values of a map: the keys are not part of the values, but only associated with them in the context of the map. Such an association is not global, but scoped to the map.

The association between a node and its p-face is provided by a node collection which contains the node. We introduce the concept of a **p-faced collection** (short for property-faced collection), which is a conventional node collection plus a mapping associating each node with a p-face. We do not constrain the individual p-faces to use the same property names: the number and names of external properties can vary from node to node.

The following table shows a simple example of a p-faced collection. In this case, all collection members are associated with the same set of external properties. The collection may therefore be represented by a table in which one column identifies the node and the remaining columns represent external properties.

Table 1.

***Example of a p-faced collection.* The collection associates each node with three external properties. Property value items are separated by comma.**

node URI	countryCode	neighbourCountries	rivers
`file://coun-tries/fr.xml`	fr	at, be, ch, de, es, it, lu, mc	Rhein, Loire, Maas, Rhone, Seine, Garonne, Mosel, Marne, …
`file://coun-tries/de.xml`	de	at, be, ch, cz, dk, fr, lu, nl, pl	Donau, Elbe, Rhein, Oder, Weser, Main, Havel, Mosel, …
`file://coun-tries/it.xml`	it	at, ch, fr, sm, si, va	Drava, Po, Adige, Te-vere, Adda, Oglio, Ta-naro, Ticino, ...

A p-faced collection can be filtered by supplying a set of conditions to be tested against the p-faces of its nodes. Such a set of conditions we call a **p-filter**. The result of applying a p-filter to a p-faced collection is a sub collection of the original collection, which we call a **p-filtered collection**.

A subsequent section will define the syntax and semantics of property filters in detail. For the moment, assume that such a filter may be represented by a string, and further assume that a p-faced collection is identified by a URI. Concatenating the URI and the query string, we can then identify sub collections by a single string, for example:

```
http://example.com/pcollections/countries?neighbourCountries=be
```

A new XPath function may provide the functionality of resolving such strings to the node sequences they identify. If the model were adopted by the XPath standard, such a function might be `fn:nodes`, and one of its signatures might be:

```
fn:nodes($uriQuery as xs:string) as node()*
```

The following snippet shows a simple XQuery program using this function:

```
declare variable $country external;
declare variable $countries := "http://example.com/pcollections/countries";

<neighbours cy="{$country}">{
   fn:nodes(concat($countries, '?', $country))
}</neighbours>
```

A crucial aspect of p-faced collections is the possibility to separate the evaluation of the p-filter from the construction of the selected nodes. Filter evaluation does not require looking “into” the candidate nodes, in fact it does not require the construction of any nodes. Filter evaluation is an operation on a collection of logical maps, the p-faces. The **p-filter engine** may treat the nodes as opaque strings, to be delivered as such. Resolving these strings to the nodes they represent is the task of an independent component, the **node builder**. The operations of p-filter evaluation and node construction are distinct and independent. It follows that the p-filter engine need not be based on XML technology. It is possible, for example, to store the p-faced collection in a relational database table and use as p-filter engine the relational database system. In this case, the p-filter supplied to the `fn:nodes` function would be translated into a SQL query. It is also possible to store the p-faced collection in a NOSQL object collection. The p-filter engine would then be the NOSQL data base system, and the p-filter would be translated into a NOSQL query. Any p-filter engine can be combined with the same node builder, as the mapping of a string to a node is in no way influenced by the way how the string was obtained.

In summary, p-faced collections promise two distinct benefits. First, they augment the navigational model of XPath with data-based access to information, overcoming the current limitation that document access requires knowledge of the document URI. Second, they offer the possibility to integrate non-XML technologies into the implementation of XML data storage and XPath navigation, which may enhance the scalability of XML based systems significantly. The concept of p-faced collections must now provide

- a model of p-filters in terms of data model, syntax and semantics
- a proposal how to integrate p-faced collections into the XPath language
- an outline how XPath processors may support the implementation of p-faced collections using non-XML technologies

Filter model

A p-filter defines a condition which can be applied to a p-face and yields a Boolean result. As every node in a p-faced collection is associated with a p-face, a p-filter can be used to specify sub collections. This section proposes a generic model of a p-filter, which includes the data model, syntax and semantics.

Concept

Attempting to strike a balance between simplicity and expressiveness, we constrain a p-filter to consist only of atomic conditions and Boolean conditions. An atomic condition submits the value of one particular property to a particular test, and a Boolean condition is a Boolean operation (`and, or, not`) applied to the results of atomic and/or Boolean conditions. A p-filter can therefore be viewed as a logical tree with leaves representing atomic conditions and inner nodes representing Boolean conditions.

Atomic conditions are modelled as comparisons between a property value and a test value. Like the property value, also the test value is a sequence of atomic items. These items are specified as literal strings or string patterns – they cannot be references to other property values. The comparison is defined in terms of an operator (=, <, <=, …) and a quantifier (`some` or `every`) specifying if all property value items or only at least one item must pass the test.

The information content of a p-filter is defined in terms of the XDM data model, namely as an element information item constrained in particular ways. The data model must be distinguished from the syntax model which defines string representations of this information content. The semantics of a p-filter consist of the rules how to evaluate a p-filter in the context of a p-face to a Boolean result.

The generic p-filter model allows the use of a p-filter engine (for example a particular relational database system) which does not support all possibilities implied by the model. To this end, the model includes the definition of *features* with implementation-defined values. This approach combines the precise specification of a fairly rich functionality with conformance criteria based on a subset of the functionality, and it provides for a standard way how to express implementation-specific limitations.

Data model

A p-filter is an element information item with the local name `filter` and a namespace URI yet to be decided. (For the time being we assume the URI `http://www.w3.org/2013/xpath-structures`). In the following text we use the prefix p to denote this URI. The filter element is either empty or has element-only content. It can have any number of child elements. Each child element represents either a property condition or a Boolean condition.

A **property condition** specifies a condition referring to a particular property. It is represented by an element information item in no namespace and with a local name equal to the property name. The condition can be structured into four logical components, described by the following table.

Table 2.

The components of a property condition and their XML representation.

Condition component	**XML representationn**	**Explanation**
Property name	The local name of the element.	Note that property names are constrained to be NCNames. If this is changed, the property name will be represented by the value of a @name attribute.
Test value	Either the single text child of the element, or the `p:term` child elements.	If there is only one test value item, it is represented by the text content of the element. If there are several items, use for each item a `p:term` child element with a single text child. Note that test value items are literal – they cannot be references to other property values. Also note that test value items cannot be accompanied by type information.
Operator	Optional @op attribute; default value: =; valid values: =, !=, <, <=, >, >=, ~, %, #=, #!=, #<, #<=, #>, #>=	A leading # indicates a numeric comparison; ~ is a regex match governed by the XPath regex rules; % is a regex or pattern match with implementation-defined semantics.
Quantifier	Option @qua attribute; default value: `some`; valid values: `some, every`	If the value is `every`, the condition requires every item of the property value to meet the condition; otherwise, only at least one item must meet the condition.

An additional detail concerns the use of regex matching (@op equal ~). In this case test value items containing a # are interpreted as follows: the substring preceding the # is the pattern, the substring following it is a set of match option flags (s, m, x, m, q) as they can be used in the third parameter of standard function `fn:matches`.

Some examples of property conditions:

```
<time>2013-01-01T00:00:00</time>
<time op="&lt;">2013-01-01T00:00:00</time>
<time op="&lt;" qua="every">2013-01-01T00:00:00</time>
<keywords op="~">^semantic.*</keywords>
<keywords op="~">^semantic.*#i</keywords>
<keywords op="~">semantic%</keywords>
<codes>
```

```
      <p:item>DUS</p:item>
      <p:item>CGN</p:item>
   </codes
```

A **Boolean condition** specifies a Boolean operation applied to the results of other conditions which are property conditions and/or Boolean conditions. A Boolean condition is an element information item with a local name equal to `and,` `or` or `not` and the same namespace URI as the `p:filter` element. Examples:

```
   <p:and>
      <time op="&gt;">2012-12-31T22:30:00</time>
      <time op="&lt;">2013-01-01T00:00:00</time>
      <p:or>
         <travelType op="~">^accommodation.*</travelType>
         <supplier>TUI</supplier>
      </p:or>
      <p:not>
         <eventDate op="&lt;" qua="every">2013-08-01</eventDate>
      </p:not>
   </p:and>
```

Syntax

Being defined as an element information item, a p-filter can be serialized to a string like any other XML data. However, in certain situations a more compact and intuitive syntax is desirable - for example when passing a p-filter to a REST service as part of a URI. Therefore we define a “query syntax” which represents a p-filter by a non-XML string. The syntax represents a *property condition* by these patterns:

```
   property-name operator testvalue
   property-name operator (testvalueItem-1, testvalueItem-2, ...)
```

and a *Boolean condition* by these patterns:

```
   condition and condition
   condition or condition
   not condition
   not (condition1, condition2, ...)
```

Parentheses can be used to change the binding of operands to operator (by default: and > or > not). A non-default quantifier (that is, `every`) is encoded by a prefix character preceding the operator string (e.g. $=). Example of a p-filter rendered in query syntax:

```
   time >= 2012…
   and time <=2013…
   and (travelType ~ ^accommodation.* or supplier=TUI)
   and not(eventData $> 2013-01-01)
```

Note that this is only a sketch of the intended syntax – a detailed specification is left to future investigation. In particular it must be decided if test values are quoted.

Semantics

A p-filter can be applied to a p-face in order to obtain a Boolean result. The atomic operation of this evaluation is the checking of a single property value item against a single test value item. Such a test is called a **comparison constraint**. Consequently, the semantics of p-filter evaluation can be divided into general rules and rules governing the evaluation of a comparison constraint. These are the general rules:

- an empty p-filter evaluates to `true`
- a non-empty p-filter evaluates to `false` if at least one child element evaluates to `false`
- an `and` condition evaluations to `false` if at least one child element evaluates to `false`
- an `or` condition evaluates to `true` if at least one child element evaluates to `true`
- a `not` condition evaluates to `false` if at least one child element evaluates to `true`
- a property condition referencing a missing property evaluates to `false`
- a property condition with quantifier `some` evaluates to `true` if at least one property value item and at least one test value item meet the comparison constraint
- a property condition with quantifier `every` evaluates to `false` if for at least one property value item there is no test value item with which it meets the comparison constraint

A comparison constraint is specified in terms of an operator. These are the evaluation rules:

- the operators =, !=, <, <= >, >= are evaluated by applying to the operands the corresponding XPath operator; operands are typed as `xs:untypedAtomic`
- a comparison operator with a leading # is evaluated by applying to the operands the corresponding XPath operator; operands are converted to `xs:double`
- the operand ~ is evaluated by evaluating the `fn:matches` function, using the property value item as first argument and the regex and options parts of the test value item as second and third arguments. If the test value item contains a # character, the regex and options parts are the substrings before and after the character; otherwise, the regex part is the complete item and the options part is the empty string.
- the operator % has implementation-defined matching semantics

New XPath functions

Given a p-faced collection, p-filters can be used to select a sub collection, retaining only those nodes whose p-face passes the p-filter. This amounts to a logical operation, mapping a collection URI and a p-filter to a sub collection. New XPath functions can make this operation available. Assuming as a tentative function name `fn:nodes`, three variants may be useful:

```
fn:nodes($collectionUri as xs:string, p:filter as element(p:filter)) as node
fn:nodes($collectionUri as xs:string, p:filter as xs:string) as node()*
fn:nodes($filteredCollectionUri as xs:string) as node()*
```

The second signature expects the p-filter in query syntax (see *Syntax*). The third signature expects a string composed of a collection URI and a p-filter in query syntax, separated by a ? character, for example:

```
http://www.traveltainment.de/pfc/shoppingCart?booking-date=2013-08-01
```

Also when concatenated with the collection URI, the p-filter in query syntax is not URI-encoded.

The introduction of these functions should be accompanied by an extension of the dynamic context of XPath evaluation: an additional component **available property-faced collections** provides a mapping of URIs to property-faced collections. A property-faced collection is formally defined as a collection which is associated with a mapping of p-filter instances to sub collections

Exposure by REST-ful web services

A p-faced collection can be exposed as a REST-full web service, using as URI the concatenation of collection URI and (URI-encoded) p-filter described in the previous section. As a web service is expected to return a single document, rather than a sequence of nodes, the selected sub collection may be wrapped by an element and thus integrated into a single document. The proposed convention is to use a `<p:collection>` element with two attributes providing the collection URI and the (not URI-encoded) p-filter. Example:

```
<p:collection uri="http://www.traveltainment.de/pfc/offers" p-filter="destin
      xmlns="http://www.traveltainment.de/pfc/offers"
      xmlns:p="http://www.w3.org/2013/xpath-structures">
   <offer>...</offer>
   <offer>...</offer>
   <offer>...</offer>
</p:collection>
```

Deployed and external collections

An XPath processor may provide access to two different groups of p-faced collections:

- deployed collections
- external collections

A deployed collection is accessed according to configuration data. This might, for example, involve the translation of the p-filter into a SQL query, its execution and the construction of nodes from result set data. See *Implementation and configuration* for details about the deployment of p-faced collection.

An external collection is accessed by applying the `fn:doc` function to a URI which is a concatenation of collection URI and (URI-encoded) p-filter in query syntax. The URI is expected to yield a document which represents the selected nodes as child elements of the document element (see *Exposure as REST-ful web services*)). The `fn:nodes` function might operate as follows:

- if the collection URI identifies a deployed collection – evaluate the selection in accordance with the configuration data
- otherwise – construct a URI concatenating the collection URI and the p-filter, resolve this URI to a document and deliver the child nodes of the document element

Implementation and configuration

A crucial aspect of p-faced collections is that they may be implemented using non-XML technologies, namely relational database tables and NOSQL object collections. This follows from the fact that the node itself is represented by a string and that it is not inspected during the filtering operation. Consequently, the implementation of a p-faced collection is not dealing with any tree structure. The implementation must provide for each node the representation of a logical map in which one entry represents the node itself and the remaining entries represent the names and values of properties. Configuration data include the name of the entry representing the node and specify how the node (if selected) is constructed from the string representing it. The string may be a serialization of the node, or it may consist of locator information, either a regular URI or a locator with implementation-defined semantics.

If the p-faced collection is “homogeneous” in the sense of every node having the same external properties, a relational table is a straightforward choice. If the nodes have different sets of properties, a NOSQL collection may be more appropriate. An example for the latter case are log data, where each log entry has some generic properties including an event type and further properties which are event type-specific. An event of type “file-not-found” may have a property “file-name”, whereas an event “container-delivered” may have properties specifying a container-number and a location.

Configuration data are required in order to enable an XPath processor to access the implementation of a p-faced collection:

- the collection URI
- the implementation technology, implying the backend query language to use
- connection data (e.g. database name and credentials and a database table name, or the name of an object collection)
- the type of node representation (serialized node, or URI, or an implementation-defined kind of locator)

Collection features

The model of a p-filter includes details which may not be well supported by a particular technology which on the whole seems well-suited to implement p-faced collections. For example, the used technology may not support …

- the regular expression dialect used by XPath
- case-sensitive property names
- case-sensitive property values
- property values with multiple items
- searches requiring all property value items to meet a condition

Two perspectives must be distinguished:

- the collection provider – using a particular implementation technology, his choices of property names and values may be limited
- the collection user – certain aspects of selection may not be supported

Defining the integration of p-faced collections into XPath, only the second perspective is relevant, as it impacts the semantics of XPath expressions. The first perspective may affect the names and values of properties, but not the syntax and semantics of XPath expressions. Such limitations are hidden from the collection user by the actual use of property names and values. For example, if the technology excludes multiple property values, the collection simply does not use multiple-valued properties; or if the technology does not support case-sensitive property names or values, the collection arbitrarily defines all property names to be lower case (or upper case).

The concept of **collection features** is introduced in order to enable a standardized representation of limitations, and to enable a definition of conformance which can be achieved in spite of limited support of the p-filter model. The following table presents a tentative set of collection features.

Table 3.

Collection features.

Collection feature	Description	Remark
{operators}	A list of supported operators. The list must contain the “=”	A p-filter must not use an operator not contained in {operators}.

Collection feature	Description	Remark
	operator, all other operators are optional.	
{every-supported}	If `false`, a p-filter must not contain the `every` quantifier.	Every implementation must support the `some` quantifier.
{multiple-test-supported}	If `false`, a p-filter must not specify test values with multiple items.	The use of multiple test values can be emulated using `p:or`.

Pushing the boundaries of the info space

The boundaries of the info space distinguish resources that belong to the space from resources that do not belong to the space. A resource is excluded from the info space for one of these reasons:

- Physical inaccessibility
- API-based inaccessibility
- Media type-based inaccessibility

Physical access is out of scope. API-based inaccessibility is a target for extensions of the XPath function library. Media type-based inaccessibility presently amounts to the simple rule: XML resources are within, non-XML resources are not within the space. This is the basic weakness of the info space seen as an information architecture, and it is arguably the greatest challenge for a well-guided evolution of XML. Accordingly, this paper concentrates on the media type boundary and limits itself to a couple of notes concerning the API-based boundary.

The API-based boundary can be shifted by the introduction of new standard functions for accessing XML documents which are presently not accessible using `fn:doc`. These functions might address:

- Documents requiring HTTP authorization
- Documents requiring special HTTP header settings
- Documents requiring the HTTP POST method
- Documents within archives (e.g. zip)

Conceivable signatures would be (informally, for the sake of brevity):

```
httpDoc($uri, $username, $password, $headerFields, $body) as document-node()
zipDocs($uri, $directories, $namePatterns) as document-node()*
```

Model oriented resource representation

The most critical limitation of the info space is its commitment to the XML media type. The most important extension of the space can be achieved by overcoming this limitation.

An integration of non-XML resources into the info space is enabled by a consistent distinction between information and representation: between model and format. The basic principle of XML technology might be applied to non-XML resources: information *is* an instance of the data model. The info space can be extended by redefining non-XML formats to be *representations* of XDM node trees.

A format defined to represent node trees might be called a **model oriented resource representation**: a resource is viewed as a unit of information which is captured by a model instance, and this instance is represented by the format. The format instance is an expression, the model instance its value.

Introductory example

Scenario: we want to check some country codes, we have access to a dictionary of country codes and names, but this dictionary is represented by a CSV dataset.

Now let us assume there is a standardized XML vocabulary for table data, comprising "table", "row" and "cell" elements. And we further assume the existence of a new `fn:doc` variant which delivers CSV data as an XML node tree based on the table vocabulary. It enables us to select all invalid country codes by a simple expression:

```
doc("foo.xml")//countryCode
   [not(. = doc("countries.csv", "csv")//row/cell[1])]
```

This example demonstrates a seamless integration of data from XML and non-XML resources: both participate in the info space.

A generic pattern of integration

Let us assume that for a given data format F three sets of rules have been defined:

R1	Rules how to map an instance of F to a node tree
R2	Rules how to map a node tree to an instance of F
R3	Rules how to decide whether a given node tree can or cannot be mapped to an instance of F

And let us further assume that these rules satisfy two conditions:

C1	For any format instance G: R1(G) satisfies R3
C2	For any R3-conformant node tree N: R1(R2(N)) is deep-equal to N

where `R1(G)` denotes the result of applying R1 to a format instance G and `R2(N)` denotes the result of applying R2 to a node tree N.

C1 and C2 imply a constraint concerning round-tripping which starts with a format instance G: the result is a format instance G' which is not necessarily equal G but which is "FORMAT-deep-equal": R1(G) is deep-equal R1(G'). This amounts to defining the information content of a format instance G: it is the node tree provided by R1. Relying on this definition of information content, instances of F can be safely evaluated or modified by evaluating or modifying the corresponding node tree. This is the foundation of an integration of format F into the info space.

A normative definition of rule sets R1, R2 and R3 enables the standardized integration of a non-XML format into the info space. This requires two extensions of the XML standards. The first extension is an XPath function which parses an instance of the format into a node tree, reflecting R1. The second change extends the XML serialization model [5] by the definition of a serialization to format F, reflecting R2 and R3. The interface of serialization is accordingly extended by an additional value of the serialization parameter `model`.

A concrete proposal for the required XPath function is the introduction of a second variant of `fn:doc` which has an additional parameter specifying the source data format:

```
fn:doc($uri as xs:string, $mediaType as xs:string) as document-node()
```

Assuming that rule sets have been standardized for the formats HTML, JSON, CSV, SQL and CSS, access to non-XML resources might look like this:

```
fn:doc("http://example.com/foo.htm", "htm")
fn:doc("http://example.com/foo.jsn", "jsn")
fn:doc("http://example.com/foo.csv", "csv")
```

```
fn:doc("http://example.com/foo.sql", "sql")
fn:doc("http://example.com/foo.css", "css")
```

The following sections explore how the rule sets govering parsing and serialization may be defined.

Vocabulary based integration

Domain-specific languages (like SQL) and non-recursive generic formats (like CSV) may be represented by a particular XML vocabulary. The formal definition of such a vocabulary might comprise an XSD and a (possibly empty) set of additional constraints expressible as XPath assertions. As a simple example, the vocabulary for representing CSV data might be defined as a schema containing element declarations with names like "table", "row" and "cell" as well as type definitions constraining a `table` element to have only `row` children and `row` elements to have only `cell` children. An additional constraint might prescribe that every `row` element has the same number of cell children:

```
count(distinct-values(//row/count(cell))) eq 1
```

As a second example, an SQL vocabulary might contain a specific element for each SQL statement, with a content model designed to capture all information which the SQL syntax may convey.

Wiki dialects are non-XML formats representing a node vocabulary (XHTML). The motivation is of course not integration into the info space, but access to a rendering processor. And here a format was added to an existing node vocabulary, whereas integration adds a node vocabulary to an existing format. Nevertheless, it is interesting to notice the popularity which a non-XML/HTML representation of node trees can attain. Standardization work might take this phenomenon into consideration.

Vocabulary-assisted integration

Vocabulary-based integration is natural for formats which do not themselves generate vocabularies as JSON does. Such a generative format might in principle be mapped to a fixed XML vocabulary, by mapping the names supplied by the format instance to XML *data*, rather than to XML names. However, if the format has a recursive structure – like JSON – the resulting XML would look unnatural and be inconvenient to evaluate. Such a mapping would have very limited practical value, and it would violate the concept of integration as explained in *a generic pattern of integration*. The concept is based on regarding non-XML formats as alternative representations of node trees, as if the node trees were the original information and the format were a means to capture this information. If the node tree looks unnatural, the image presupposes the desire to convey information which is unnaturally structured in the first place, which is absurd.

If JSON is redefined to be the representation of node trees, these node trees should probably capture JSON names by node names. Several approaches of mapping JSON to XML have done that as far as possible (e.g. [7], [12], [13], [14], [16]) but they could not achieve two goals:

- Represent any JSON name by an equal node name
- Renounce the use of XML nodes containing JSON-related control information

The first problem arises from the fact that XML names are QNames. The second problem reflects a mismatch of data models: whereas JSON uses two distinct content models, one a map and the other a sequence of anonymous items, XML has a single content model which is sequence-and-name based. The XML model can be used to emulate the JSON models, but it cannot express them natively. Those mapping approaches might be classified as vocabulary-assisted: the bulk of items is named after their JSON counterparts, but a small additional vocabulary is used in order to specify relationships between the nodes and their JSON counterparts. These additional items prevent an evaluation of the node tree from ignoring representations and only seeing the node tree itself.

Native integration

JSON is, like XML, recursively defined and generates vocabularies. The ideal way to integrate it into the info space would be native integration: every JSON instance should be the representation of a node tree which natively expresses the JSON content, so that it can be serialized to JSON based on the node properties only, without requiring the presence of additional nodes guiding the serialization. Native integration of JSON is not possible due to differences of the XML and JSON data models: the XML model is *not* a superset of the JSON model.

The extension of XML

JSON is a very popular generic markup language. If the XML data model is not a superset of the JSON model, it must be questioned if the XML data model is qualified to define the info space - an information architecture not tied to a particular markup language. The XML data model is very expressive and allows representing virtually any tree-structured data as long as each node cannot have more than one parent. But in three cases (at least) a given structure must be emulated, rather than be expressed natively. Calling the data to be represented a source tree composed of nodes, these are:

1. If a source node has a name which is not a valid QName, the XML node has a different name.
2. If a source node is anonymous, the XML node must nevertheless have a name.
3. If a source node has unordered content, the XML node cannot avoid adding the information of a particular order – the data model does not allow “switching off” the meaningfulness of order.

The emulation may deal with arbitrary names either by mapping the original name to an NCName or by shifting the source name into data (e.g. an attribute). Anonymous nodes can be represented by nodes with an arbitrary name. The insignificance of order can be conveyed by out-of-band information. As long as one can start and end with XML, these workarounds are not perceived as serious issues. Likewise, as long as an initial JSON representation can be permanently replaced by XML, there is also no problem as the emulation provides semantic equivalence. But if there is a need to *maintain* a relationship between node tree and JSON representation, one discovers that there is no way to natively express arbitrary JSON data by an XML node tree.

(2) and (3) amount to a mismatch between XML and the dominant content model used in programming languages: which is a choice between sequence-based (an "array" of anonymous items) and key-based content ("map", "hash", etc.: an unordered collection of keyed items). XML has one single content model which is a curious hybrid: sequence+name based, which means the content is always ordered (array-like), but each sequence item has a name, which is key-like but not a key, as it is not required to be unique among the siblings.

The fact that virtually any tree-structured information can be emulated in XML has encouraged the view that the content model is sufficient. The fact that the content model is not as generic as it seems on first sight – that it is incapable of expressing natively arrays, maps and string-based names – has not yet caught much attention. This might change if the concept of an info space gains momentum. The concept implies a certain shift of priorities, attaching importance to a native representation of non-XML resources by the XML data model. Such a representation is required for seamless navigation: navigation which sees the information content (the model instance) and is unaware of future or past representations.

Extension of the XML data model

Rennau [17] proposed an extension of the XML data model which claims to enable a native representation of JSON data. The proposal introduces the same boolean choice of content organization as is used in common programming language like Java: a choice between the sequence based model (which is the only model in current XML) and an alternative model which is truly map-based: the child elements are unordered, and every child element has a string-typed key – which is a new node property, [key]. The choice between the two content models is made by each complex element, and therefore it is modeled as a second new node property, [model]. It has two values, "sequence" and "map", and the relationship between the two new properties is constrained as follows: (a) an element with a non-null [key] must have a parent element with [model] equal "map"; (b) an element with a null [key] must not have a parent element with [model] equal "map". In other words: "map" elements have unordered children which must have keys, and "sequence" elements have ordered children which must not have keys.

Concerning the handling of anonymous items, Rennau proposed the introduction of "unspecific standard names", but this is unsatisfactory, as it renounces the goal of a genuinely native representation: the only natural representation of an anonymous item is an anonymous item. Therefore we should modify the XML data model by allowing anonymous elements. A JSON array would thus be an elegant representation of an element node with [model] equal "sequence" and anonymous child elements.

In summary, the following extensions of the XML data model should be considered, as they promise to overcome the limitations which disqualify the model as foundation of the info space: (a) addition of two new node properties, [model] and [key]; (b) modification of the property [node-name], introducing the possibility of a null value.

Extension of the XML syntax

Introducing new node properties implies the necessity to define the representation of these properties in XML syntax. Rennau proposed the use of "pseudo attributes" which look like attributes but do not represent attribute nodes, as they represent node properties:

- udl:model - indicates the value of the [model] property
- udl:key - indicates the value of the [key] property

where the prefix "udl" stands for "unified document language". As a consequence, however, the XML representation of a JSON instance tends to be cluttered with pseudo-attributes, which is perhaps a less important disadvantage as it may seem, as the processing of JSON-supplied data is as elegant as the processing of XML-supplied data (see below, *Extension of the XPath/XQuery language*) and the very info space architecture encourages the use of JSON notation whereever it seems more attractive than XML notation (see below, *Markup integration*) .

However, a refusal to express the new node properties with the same immediateness as the other properties is unsatisfactory. Pseudo-attributes are an adequate representation of a node property when they can be expected to be rare, as for example `xml:type` attributes. However, knowing in advance that the XML representation of JSON-supplied data will produce XML documents which are studded with pseudo-attributes - rendering them hardly readable - we are motivated to go a step further and truly extend the XML syntax, introducing new syntax constructs. The goal is to ensure readable XML representations also of such node trees as are constructed from JSON instances. The following syntax extensions should therefore be considered:

- A special character preceding an element name indicates that its [model] is "map", rather than "sequence". Example: <~foo>
- The key can be expressed within the element tag by the quoted key value. Example: <foo "2012">
- Anonymous elements which do not have a key use instead of an element name a special character. Example: <*>
- Anonymous elements which do have a key are represented by shifting the quoted key value to the beginning of the tag. Example: <"2012">

These rules would allow an acceptable representation of JSON data in XML syntax. A special advantage is that a given information content – e.g. a web service response – might then be served *without changes* alternatively in JSON or XML. Example:

```
<~getWeatherResponse>
      <"temperature min">28</"temperature min">
      <"temperature max">32</"temperature max">
</~getWeatherResponse>

{
   "temperature min" : 28,
   "temperature max" : 32
}
```

To round these considerations off, a further detail of Rennau's proposal which concerns the XML syntax should be retained: for elements which have a parent element with [model] equal "map", the key defaults to the local name. Therefore, a slight modification of the above example would look like this:

```
<~getWeatherResponse>
   <temperatureMin>28</temperatureMin>
   <temperatureMax>32</temperatureMax>
</~getWeatherResponse>
```

In this variant the child elements are not any more anonymous but have a local name equal to the key.

Extension of the XPath/XQuery language

Integration of non-XML resources into the info space is driven by the desire to make them accessible to navigation and evaluation. The extension of the data model by a new [key] property must therefore be mirrored by an extension of the XPath semantics. Rennau proposed the introduction of a third node test, the "key test" which functions in analogy to the name test: selecting elements with a given key, rather than elements with a given name. Being a node test, the key test is combined with navigational axes in the same way as a name test. The syntax of a key test is a special character (proposed: "#") followed by the key value within quotes. Examples:

```
$response/#"temperature min"
$response//#"2012"
$item/ancestor-or-self::#"AC#A-2917"
```

Thanks to the key test, JSON-supplied data can be navigated with the same ease as XML-supplied data, using key tests to select among object members and conventional numerical predicates to select among array items:

```
$response//#"weather"/#"temperatures"/*[3]
```

Whereas the key test provides for navigation, Rennau also proposed an extension of XQuery which enables the construction of nodes using JSON syntax. The extension is purely syntactical: an additional syntax which is a JSON-style shorthand notation for the construction of nodes which, due to their properties, correspond to JSON objects and arrays. For example, the following XQuery constructor:

```
<*>
   <"temperature min">{$min}</"temperature min">
   <"temperature max">{$max}</"temperature max">
   <"temperatures">{for $v in $values return <*>{$v}</*>}</"temperatures">
</*>
```

can equivalently be written using JSON syntax:

```
{
   "temperature min" : $min,
   "temperature max" : $max,
   "temperatures" : [ $values ]
}
```

The XQuery construction of JSON data using JSON syntax is as convenient as the construction of XML data using XML syntax. Taken together, the proposed extensions of XPath and XQuery ensure a complete integration of JSON into the info space:

- JSON instances can be navigated as easily as any other node trees.
- JSON instances can be constructed using JSON syntax.

Markup integration

Every syntax which is defined to be the representation of a node tree might be integrated into XML markup. Provided that any section using non-XML syntax

- identifies the syntax it uses
- is clearly delimited
- appears in a place where a sequence of elements might appear

the document text is guaranteed to be parsable into a node tree in an unambiguous way. Rennau proposed a restricted use of this new possibility which allows to provide *element contents* in non-XML syntax. The proposal is to introduce a pseudo-attribute (`udl:markup`) which identifies the syntax used to encode the element content. As a consequence, the following text would be a valid XML document:

```
<~getWeatherResponse udl:markup="json">
   "temperature min" : 28,
   "temperature max" : 32
   "temperatures" : [28, 28, 30, 32, 31]
</~getWeatherResponse>
```

Note that the `getWeatherResponse` element has three child elements, as the content of an element with `udl:markup='...'` is defined to be the nodes resulting from parsing the text found between start tag and end tag according to the parsing rules for format '...'.

Wrapping up

The info space is an information architecture consisting of distributed resources whose information content is defined to be a node tree, understood as an instance of the (slightly extended) XDM data model. The resources are identified by URIs and accessed by XPath functions (`fn:doc`, `fn:httpDoc`, ...) which hide the actual data formats and expose the node structure. Taken together, the node structure of all accessible resources provides a uniform substrate for navigation and discovery.

It should be noted that the concept of a "resource" as used in the info space model is different from the use in Fielding's REST model [11]. The latter defines a resource as a conceptual unit whose mapping to concrete representations is a function of time. The resources of the info space are units of information which are *values*, precisely defined in terms of a node tree. As such, they are more closely related to Fielding's concept of a resource representation than to his concept of a resource. On the other hand, the insistence on a well-defined *information content* is in sharp contrast with Fielding's model, in which there is no place for such a thing. Resources of the info space are the accessible resource representations (in Fielding's sense) which can be resolved to a node tree. As such, the info space is a *snapshot* of all accessible resource representations (in Fielding's sense) which have a media type defined to represent a node tree, according to the standardizations endorsed by the W3C.

Fielding's REST model is aimed at an architecture for the internet, understood as a distributed hypermedia system. His view of resource navigation assumes a model of application behaviour in which the rendering of resource representations powers subsequent navigation by supplying embedded links. The info space model, on the other hand, is unconcerned with application behaviour and paints a picture in which nothing appears but information, aggregated into distinct resources and nevertheless coalesced into a single, homogeneous substrate of information. Embedded links are predefined point-to-point connections. The connectivity of the info space, however, relies on XPath navigation, which is like an ubiquitous field of possibility, potentially leading from anywhere to anywhere. Therefore the info space is not a net but more like a space: it integrates anything (any resource) appearing within it.

The info space integrates information on a global scale, relying on a uniform data model. The same can be said about RDF [15]. What is the relationship between the XML info space and RDF? RDF is about *semantic integration*, establishing the identities of real world resources described by the data. The RDF data model is designed to enable the assembling of distributed items into synthetic descriptions, which are organized by the identities of the described resources.

The XML info space, on the other hand, is about *navigational integration*, offering it as a possible foundation for semantic integration. The semantic integration is left to applications which are supposed to understand the trees they are dealing with as a coherent, structured unit - rather than regarding them as a "mine" from which to undig self-contained items of information (triples). Such a dealing with *well-understood resources* is the typical task of conventional applications. They know what to do with, say, /a/b/c taken from one resource and with /x/y/z taken from another resource, and how they are related. For such applications the benefit of simple and efficient navigation is probably greater than the benefit of further aids how to find out those meanings and relationships. Therefore conventional applications may profit from the navigational info space immensely and probably far less from the semantic space created by RDF. Nevertheless, there may be a wonderful synergy between the XML space and the RDF space yet to be explored.

Bibliography

[1] Berglund, Anders et al, eds. XML Path Language (XPath) 2.0 (Second Edition). W3C Recommendation 14 November 2010. http://www.w3.org/TR/xpath20/

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

[3] Berners-Lee, Tim et al. Uniform Resource Identifier (URI): Generic Syntax. Network Working Group, Request for Comments: 3986. January 2005. http://tools.ietf.org/html/rfc3986

[4] Boag, Scott et al, eds. XQuery 1.0: An XML Query Language (Second Edition). W3C Recommendation 14 December 2010. http://www.w3.org/TR/xquery/

[5] Boag, Scott et al, eds. XSLT 2.0 and XQuery 1.0 Serialization (Second Edition). W3C Recommendation 14 December 2010. http://www.w3.org/TR/xslt-xquery-serialization/

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

[7] 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 .

[8] Cowan, John & Tobin R., eds. XML Information Set (Second Edition). W3C Recommendation 4 February 2004. http://www.w3.org/TR/xml-infoset/

[9] DeRose, Steve et al, eds. XML Linking Language (XLink) Version 1.1. W3C Recommendation 6 May 2010. http://www.w3.org/TR/xlink11/

[10] Einstein, Albert & Infield, L. The evolution of physics. Edited by C.P. Snow. Cambridge University Press, ASIN: B000S52QZ4, 1938 http://archive.org/details/evolutionofphysi033254mbp .

[11] Fielding, Roy T. Architectural Styles and the Design of Network-based Software Architectures. Ph.D. Dissertation, University of California, Irvine, 2000. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm .

[12] Gruen, Christian, et al. BaseX Documentation Version 7.3, section "JSON Module", p. 161-166. http://docs.basex.org/wiki/Main_Page.

[13] 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.

[14] 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.

[15] Manola, Frank & Miller E., eds. RDF Primer. W3C Recommendation 10 February 2004. http://www.w3.org/TR/rdf-primer/

[16] 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.

[17] Rennau, Hans-Jürgen. "From XML to UDL: a unified document language, supporting multiple markup languages." Presented at Balisage: The Markup Conference 2012, Montréal, Canada, August 7 - 10, 2012. In Proceedings of Balisage: The Markup Conference 2012. Balisage Series on Markup Technologies, vol. 8 (2012). doi:10.4242/BalisageVol8.Rennau01. http://www.balisage.net/Proceedings/vol8/html/Rennau01/BalisageVol8-Rennau01.html.

[18] Walsh, Norman, ed. XML Catalogs. OASIS Standard V1.1, 7 October 2005. https://www.oasis-open.org/committees/download.php/14809/xml-catalogs.html.

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