
Managing XML references through the XRM vocabulary

Jean-Yves Vion-Dury

Abstract

This paper presents a general purpose method (called *XRM* for XML References Management) to express knowledge about links common to a family of XML documents (a.k.a. a document type) and to exploit this knowledge in order to operate verifications, transformations or derivations of the corresponding XML instances.

Table of Contents

Introduction	1
Problem overview	2
State of the Art	3
XCatalog	3
XLink	3
Approach Principle	3
Specification	4
Matching language	4
Execution model	4
The approach in more detail	5
Link Description	5
Link Validation	8
Link Transformation	9
Link Dependencies	11
Conclusion	11
A. The pattern matching language	12
B. Verification and execution of XRM specifications (principles)	13
References	14

Introduction

So far no specific method nor well suited technology exist to address XML link management related applications, although those are numerous and may require quite complex processing when using standard XML tools or programming languages.

We call link or reference any URL, URN, URI, IRI, XLink (see [1] and [3]) be it relative or absolute that can be found in a given XML document instance, either under the form of an attribute or as a text node (once parsed, an XML document is composed of element nodes, attribute nodes or text nodes: see [4] for a general description of XML standard).

The method and conceptual models we propose hereby allow concise and efficient XML descriptions of links that can be heavily reused, and enable adequate descriptions of main link-based operations required in XML processing environments, especially link relocation for packaging clusters of documents and associated resources, verification of link properties with respect to security, conformance to a predefined selection of HTTP servers, simplification and normalization of link representation inside a given XML instance, smooth redirection of database requests hidden inside the structure of links, to cite a few among the huge variety of relevant cases.

The knowledge about links is formalized into a specification language that

1. describes links location and typology inside a family of XML documents
2. tags these link descriptions in such a way that they can be further designated and reused either individually or collectively.

The operations on XML instances use the link descriptions above in order to

- verify the compliance of links according to the standards describing properties that these links must satisfy (e.g. lexical and syntactic structure),
- check the conformance to specific or general properties (e.g. URI must be relative, or must match a given pattern),
- generate a list of all links contained in the instance (dependencies), with related useful meta-information such as the path expression that uniquely locate them inside the hierarchical structure and the type of link (URI, IRI, XLink,...)
- rewrite some links into other links (reference relocation), depending on matching patterns, side conditions of source document as well as side conditions of referenced objects (links targets).

Problem overview

There are currently two different ways (inside XML standard) for identifying and designating items inside or outside a document. The first one is based on ID/IDREF mechanisms which only apply to intra-document references. The second one, more general, is based on URL (Uniform Resource Locator) that has been historically derived into several variants (e.g. URI, uniform resource identifiers; IRI, internationalized resource identifier; URN Uniform resource name), each having different intended use and slight lexical variations (see [1,8]).

This research work, whose results are described hereby, focused on the second kind of references. According to the related standards, references have a syntactic structure that enables describing the protocol used for accessing resources over networks, the address of the server providing the resource, the path which uniquely designates the object to be accessed, and in some cases the fragment inside the document (i.e. a unique element identifier) and/or parameters. For instance the URL *http://ds-1/example/dog.jpeg* designates an object located on the “ds-1” server and accessible through the “http” protocol. This object is called “dog.jpeg” and the server is supposed to find it through the path “/example” before delivering it back to the caller that invoked the protocol.

Although the referential objects are precisely defined through their syntactic and semantic structure, we have poor information about the context in which they are used and where they are located inside a given document. In the best case, an XML instance is compliant with an XML schema, e.g. XHTML, and thus we hopefully know where one can find such a reference, e.g. inside any *img* element, and more precisely, inside the value of its *href* attribute. Note that the semantics of the reference is implicitly defined by the informal description of the HTML standard (it points to an image; it must be fetched through the URL and incorporated into the visual representation of the containing document).

However, many specific transformation operations can be envisioned which are quite focused on these referential objects, and no methods or tools are proposed today to simplify these operations and to make them more reliable and easier to specify. Among others, one can mention :

- *link relocation*, which consists in changing the external environment of a given instance (for instance, changing absolute reference to an external server into a pointer on a local cache where the target resources are stored)
- *document and resource packaging*, which consists for instance in building an archive containing all dependent resources under a suitable directory structure
- *selective link stabilization* ; this operation allows one to substitute some references by others pointing to the same resources, but via a storage system that guaranties the long term stability of the access

- *static xml:base attribute processing* ; this operation aims at interpreting the `xml:base` attribute according to the W3C standard [5], but as a standalone operation (usually, this process is done – or just ignored...- inside the applications)
- *static XInclude resolution* ; similar remark than above

Our contribution can be understood as a way to express link specific schemas, validations and transformations. It is orthogonal (and complementary) to general purpose schemas.

State of the Art

XCatalog

XCatalog [2,6] is an XML standard which allows describing link resolving mechanisms. More precisely, the links are categorized into references to XML entities, DTD and XML schema resolution (W3C schemas only) on the one hand, and general URI that are defined as strings that must match a given prefix on the other hand.

The first category is focused on link resolution, an operational concept that concerns only programmatic toolkits and software libraries that are in charge of retrieving the content of pointed objects (so called *resolvers*). It means that the only underlying semantics is predefined as “fetch the pointed resource when needed, the way I specify”, and this behavior must be implemented by the XCatalog aware processor (typically, XML parsers). A strange point is that the XML catalog specification defines “what” and “how”, but not “when”. In other words, the semantics of links is presupposed, and indeed strongly related to the XML validation that is accomplished after parsing.

The other link category is quite general, but only defined through the concept of “exact prefix matching”. Nothing is said about the location of links and a fortiori about their context.

Thus there is a deep conceptual difference between our proposal and XCatalog: the latter is focused on resolving links, where links are recognized through their content, whereas our proposal is based upon a methodology which makes explicit the description of links through their localization in the document structure. These descriptions can be used for specifying various link oriented validation and transformation operations.

XLink

XLink [3] is a standard that describes a vocabulary and syntax for specifying generic links inside XML documents. This standard relies in a rich model allowing among others the specification of hyper-graphs, that is, graphs based on a generalized notion of arcs possibly binding several sources to several targets. XLink is based upon URI mechanism and namespace modularity.

It is not comparable with our approach, as it is a way to express links whereas our method is a way to express properties of links and the related validation or transformation operations that can be derived from these properties. As a consequence, XLink objects are specific targets of the description mechanisms we propose, so as with XInclude, XPointer and other generic linking objects (URI, IRI,...) (see the section called “The link descriptors”)

Approach Principle

In order to express high level properties over links and their localization inside instances, one needs a specialized language and dedicated abstractions. Moreover, in order to consider the link normalization phenomenon, we also need an execution model. Once captured in an adapted format, the link descriptions we propose in this paper might be reusable for specifying almost any XML link-related operations.

Our method relies on a specification method, a specialized matching language and an execution model.

Specification

From the specification point of view, our vocabulary allows one to

1. express link features by means of three separate sections:
 - a. the link typology and localization (links description), thanks to an appropriate sublanguage, typically but not exclusively, XPath [7]
 - b. the link's expected properties (validation description)

This part expresses properties that (groups of) links have to satisfy inside a given XML instance in order to be considered as valid,
 - c. the link transformation rules (link translation description) :
 - i. transposition (selected links are eventually normalized, matched against some pattern and rewritten)
 - ii. dependency extraction rules (dependency description)
2. identify, group and designate link descriptions

This one allows the user to attach one or several tags to link descriptors, and offers a mechanism for factorizing the tag assignation. Tags are simple labels intended to abstract over the semantics of links and to memorize them easily.

The idea of points 1 and 2 above is to express bindings between the descriptive section and the other sections through a convenient designation mechanism. Hence there is little overhead, and the method enables reusing link descriptions in various applicative contexts.

Matching language

The specialized matching language is designed in order to optimize the ratio expressive power versus complexity; in other words, it simplifies the task of expressing the structural properties of links, the (pre/post) processing and transformation of links; by offering the right abstractions, and by relying on the inherent lexical/syntactical structure of links, it avoids the burden of mastering general regular expression languages, tricky and error prone for a non-specialist. Details on this aspect of our contribution can be found in Appendix A, *The pattern matching language*

Execution model

From the execution model point of view, our approach allows one to

1. use the link validation description either via an interpreter or via a compiler to operate the verification on any instance expected to comply with the description; the verification may output an error report including the faulty links, their location in the document and an indicative error message or any other relevant information ;
2. use the link translation descriptions either via a direct interpretation or via a compilation/execution scheme to operate the modification of links and possibly generate a new document instance in which relevant links have been modified according to the transcription rules (but without any other structural changes); this operation may output a log report indicating which links have been processed and any other relevant information ;
3. use the dependency extraction rules either via an interpreter or via a compiler to produce a list of all dependencies, i.e. all resources the given instance is sensitive to, as estimated by the designer who specified the dependency rules (Order may be significant, if specified so).

Details of significant steps behind applying XRM to some target XML instances can be found in appendix Appendix B, *Verification and execution of XRM specifications (principles)*

The approach in more detail

Link Description

Overview

Links are described in a dedicated XRM element called “links” associated with information

- indicating a unique logical name for this section, which will be used for designing it without ambiguity
- specifying the namespace of the target document, if any (see [13] for a description of namespaces)
- providing the URL of one or several schemas to which the target document is expected to comply with (optional)
- listing all tags used to annotate the link descriptions; this list is optional, but if provided, it defines exactly and exhaustively the authorized tags. Tags are names with any relevant lexical structure, as commonly found in the art.

Inside the section, the designer of the description can input as many descriptors, possibly embedded in grouping subsections. These subsections are decorated with a tag list; the meaning of this grouping subsection is that all embedded descriptions will be automatically assigned the associated tags. It is thus a way to simplify the specification of descriptors (see example Figure 1, “a link description for XHTML”).

The link descriptors

The descriptors themselves are specified through one of the following keywords :

1. *URL* stands for Uniform Resource Locator (see [1]) and is commonly used to give information on where a resource is located, understanding that the implicit action is to fetch this resource in order to incorporate it inside the document (e.g. an image, a sub-part) or to interpret it with respect to the current document (e.g. a script)
2. *URN* stands for Uniform Resource Name and aims at naming resources in a worldwide unique and temporally stable way. Thus no specific action or usage is associated with them, they are just used to designate things (e.g. in PUBLIC field of DTDs); however, they often have a specific lexical structure, mainly a “urn” scheme and ‘:’ separated sequence of characters (e.g. urn:example:animal:ferret:nose)
3. *URI* stands for Uniform Resource Identifier ([1]) and commonly used to identify a resource in a broader way. The RFC 3986 from IETF explicitly says:

“ [...] A Uniform Resource Identifier (URI) is a compact sequence of characters that identifies an abstract or physical resource. [...] ”
[RFC 3986 from IETF]

This excerpt insists on the potential abstract nature of the pointed resource. In the sequel, the abstraction hierarchy and relationship between URL, URN and URI is clearly described:

“ [...] URI can be further classified as a locator, a name, or both. The term "Uniform Resource Locator" (URL) refers to the subset of URIs that, in addition to identifying a resource, provide a means of locating the resource by describing its primary access mechanism (e.g., its network "location"). The term "Uniform Resource Name" (URN) has been used historically to refer to both URIs under the "urn" scheme [RFC2141], which are required to remain globally unique and persistent even when the resource ceases to exist or becomes unavailable, and to any other URI with the properties of a name. [...] ”

[idem]

From the lexical point of view, a URI must only use UCS (Universal Character Set) code points; these code points must be converted to bytes through the UTF-8 encoding, but when the character doesn't belong to the unreserved subset, it must be escaped using a "%HH" pattern before encoding (full details in [1]).

4. *IRI* stands for Internationalized Resource Identifier (see [8]) and has the same meaning and syntactic structure than URI, but a more abstract lexical structure. An IRI uses hence an extended character set supporting foreign languages (foreign should be understood here as non-English), including right-to-left writing languages such as Arabic. The specification describes the translation algorithm that transforms an IRI into an URI (thus allowing physical access if required) through a character normalization phase followed by an escaping mechanism based on %HH patterns (H stands for any hexadecimal letter taken from the 0-9A-F alphabet).
5. *HREF* refers to "Hyper-references" defined in the HTML vocabulary among others. Those links have a specific encoding policy, using a similar escaping mechanism than URI, but with stricter character set (namely, ASCII)
6. *XInclude* refers not only to the link associated with it, but to the whole node. This element is meant to express document inclusion, a not so simple mechanism whose semantics is precisely specified in [9] and makes use of a predefined attribute "href" containing a specifically encoded URI according to section 4.2.2 of the XML 1.1 specification [10]:

“ [...] System identifiers (and other XML strings meant to be used as URI references) MAY contain characters that, according to [IETF RFC 2396] and [IETF RFC 2732], must be escaped before a URI can be used to retrieve the referenced resource. The characters to be escaped are the control characters #x0 to #x1F and #x7F (most of which cannot appear in XML), space #x20, the delimiters '<' #x3C, '>' #x3E and '"' #x22, the unwise characters '{' #x7B, '}' #x7D, '|' #x7C, '\' #x5C, '^' #x5E and '~' #x60, as well as all characters above #x7F. Since escaping is not always a fully reversible process, it MUST be performed only when absolutely necessary and as late as possible in a processing chain. In particular, neither the process of converting a relative URI to an absolute one nor the process of passing a URI reference to a process or software component responsible for dereferencing it SHOULD trigger escaping. When escaping does occur, it MUST be performed as follows: 1. Each character to be escaped is represented in UTF-8 [Unicode] as one or more bytes. 2. The resulting bytes are escaped with the URI escaping mechanism (that is, converted to %HH, where HH is the hexadecimal notation of the byte value). 3. The original character is replaced by the resulting character sequence. [...] ”

[World Wide Web Consortium]

7. *XLink* as for XInclude, refers to a node supposed to contain XLink related attributes (see [3]); the specific href attribute from the XLink namespace is an URI. The general semantics constraints of XLink are captured by this descriptor.
8. *XPointer* describes a very rich mechanism (see [11, 12]), based on URI and possibly using various selection languages (so-called *schemes*), one of them, most notably, extending XPath in order to designate one or several fragments of an XML document tree including segments in text nodes.

Each such descriptor is associated with a locator, that is, an expression of a node selection language that defines where the link should be located in the document instances under consideration. Note that these XPath may use various namespaces, provided they are consistently declared thanks to a special element called *ns* (the same mechanism is used inside Schematron specifications [18])

The Figure below illustrates how our method can be used to describe links in any XHTML compliant document ¹.

¹All XPath expressions are here interpreted inside the default namespace specified in top-level element "links" through the "ns" attribute.

Figure 1. a link description for XHTML

```
<links id="xhtml-1.0" ns="http://www.w3.org/1999/xhtml">
<!-- XHTML 1.0 -->

<tags>image-locator source-locator code-locator
      header links descriptor citation doc-base</tags>

<group tag="header" locator="/html/head">
  <iri locator="/@profile"/>
  <iri tag="doc-base" locator="/base/@href" />
  <iri tag="links" locator="/link/@href"/>
  <uri tag="source-locator code-locator" locator="/script/@src"/>
</group>

<iri tag="descriptor" locator="//iframe/@longdesc"/>
<iri tag="source-locator" locator="//iframe/@src"/>
<iri tag="image-locator" locator="/body/@background"/>

<group tag="citation">
  <iri locator="//blockquote/@cite"/>
  <iri locator="//ins/@cite"/>
  <iri locator="//del/@cite"/>
  <iri locator="//q/@cite"/>
</group>

<group tag="references">
  <iri locator="//a/@href"/>
  <group locator="//object">
    <iri locator="/@classid"/>
    <iri tag="code-locator" locator="/@codebase"/>
    <iri locator="/@data"/>
    <iri locator="/@archive" list="yes"/>
    <iri locator="/@usemap"/>
  </group>
  <iri tag="code-locator" locator="//applet/@codebase"/>
</group>

<group locator="//img">
  <iri tag="image-locator" locator="/@src"/>
  <iri tag="descriptor" locator="/@longdesc"/>
  <iri locator="/@usemap"/>
</group>

<iri locator="//area/@href"/>
<iri locator="//form/@action"/>
<iri locator="//input/@src"/>
<iri locator="//input/@usemap"/>

</links>
```

An example of a generic link description for XHTML. The descriptors can be further reused for other operations through a tag based designation mechanism

Link Validation

The link verification is specified in a dedicated section called *validate* which contains at least the reference on a link description section, as detailed above (this reference is an URL), which can be located inside or outside the document containing the *validate* section. If no other information is specified, all links should be checked with respect to the specified semantics. This means that when the verification is executed on a given target XML instance, the links are extracted thanks to the localization information and are examined in accordance with their type as detailed in the previous section.

Additional constraints can be provided through one or many “properties” subsections.

Each properties subsection applies to one or several link subsets designated through a list of one or several tags. Each tag may designate one or several links, depending on the link description section, as explained above. Each properties subsection is optionally identified through a unique identifier.

The properties are specified through one or several descriptors as listed hereafter:

1. *scheme* defines the expected scheme, e.g. “http”, “ftp” or “mailto”
2. *absolute* expresses that an absolute link is expected (the scheme and server location are provided)
3. *relative* expresses that a relative link is expected (the path, resource name and optionally the fragments are provided; the scheme and server location are those of the base URI of the target instance, as specified in [1])
4. *matches(p)* expresses that the link content must match the provided pattern p. This pattern is expressed according to the method described later.
5. *path(p)* expresses that the “path” part of the link (see URI syntactic structure in [1]) must match the given pattern p.
6. *fragment(p)* expresses that the “fragment” part of the link (see [1]) must match the given pattern p.
7. *query(p)* expresses that the “query” part of the link (see [1]) must match the given pattern p.
8. *target()* expresses that the target reference is available at the time of the verification; one of several sub-descriptor can be specified, in order to make-it more precise:
 - a. *mime-type* This is a standardized notation for indicating the type of internet resources (see [15])
 - b. *namespace(ns)* (makes sense only if the mime-type is text/xml or derived).
 - c. *condition(p)* ; as for previous item, this condition needs a parsable XML content ; requires checking if conditions p holds (p is a XPath qualifier expression)

Note that points b and c above require solving the reference at verification time, and also possibly XML decoding and/or parsing.

If no descriptor is specified, only standard verifications related to the nature of links are conducted.

An additional error message can be specified within each property descriptor, that will be used to report any property violation (e.g. `matches(http://{*}:{*}/{*},”an explicit port number is expected”)` will display the error message for non-matching link such as `http://barnum/circus.jpg`)

The following example illustrates the method when applied to an XHTML document

Figure 2. A link validation specification

```
<validate link-description="../../schemas/html.xrm.xml#xhtml-1.0">

  <property of="code-locator" xml:id="code1">
    <relative>references to code-related objects
      are expected to be relative</relative>
    <fragment> references on code location
      cannot point to document fragments </fragment>

    <matches normalize="yes"
      pattern="http://bonobo:{*}/code/{*}" />
  </property>

  <property of="image-locator">
    <relative/>
    <query>
      references to images cannot contain query
    </query>
    <matches pattern="http://bonobo:{*}/image/{*}" />
  </property>

</validate>
```

This specification reuses the generic description of XHTML links as shown Figure 1, “a link description for XHTML”

Link Transformation

Link transformations are specified in a dedicated section called “rewrite” which comprises a header having the following attributes:

1. *link-description*: the name of a link description section, against which link tags will be interpreted (mandatory)
2. *normalize*: take the value yes or no (defaults to yes if omitted); if set to yes, the relevant normalization process will be performed on all links before applying matching operation (the exact nature of normalization operation depends upon the nature of link); if set to no, the pattern matching operation will be applied on the original link ²;
3. *resolving-base*: optionally specifies an URI that will be considered as the reference URI for solving relative link. It supersedes the xml:base information, if present, or the static-base-uri of the original document.

2

Beside header attributes, this section is composed of zero or many rewriting descriptors possibly embedded inside a base descriptor. Each base descriptor has

4. an optional “location” attribute which expresses where an xml:base attribute must be inserted inside the transformed document. When omitted, the xml:base attribute is inserted into the root node (of course, in any case, it is an inconsistency error if several base descriptors are allocated to the same node).

²Some normalization operation may nevertheless occur due to standard XML processing, such as interpretation of escaping sequences and expansion of reference entities.

5. a “value” attribute which defines the content of the xml:base attribute. This must be an absolute URL in accordance with the standard [5]; if omitted, the static-base-uri is used.

Each rewriting descriptor may have

- a *tags* attribute, which is a list of tag name corresponding to the links to be selected as candidates (all link descriptors are considered if the tags attribute is omitted)
- a *condition* attribute, which optionally specifies an additional condition to be checked before trying to apply the rewriting (typically, an XPath expression)
- a *from* attribute, which optionally specifies a pattern matching expression that must be successfully applied in order to rewrite the link ; such pattern may define matching variables (see the subsection 3.4 “Specification of Patterns” for the whole description of the link pattern language).
- an *into* attribute, which optionally specifies a new value for the link. This value may partially or totally reuse the pattern variables defined inside the from pattern (see the subsection 3.4 “Specification of Patterns”) if any.

In the case where a rewriting descriptor has no “from” and no “into” attribute, it may have one or more rewrite sub-descriptor, each of it having a pair of “from/into” attribute. The meaning of this list is that each rewriting is tried in order, until a matching “from” is found.

Below is an example of link rewriting based on a two-rule sequence to be applied on any link tagged as “images” or “scripts”

```
<rewrite
  link-description=" ../schemas/html.xrm.xml#xhtml-1.0"
  tags="images scripts" >
  <rewriting from="{{*}}/{name}.jpg" into=" ./images/JPEG/{name}.jpg" />
  <rewriting from="{{*}}/{name}.js" into=" ./javascripts/{name}.js" />
</rewrite>
```

Note that after computing the rewritten link, and if the rewriting descriptor is embedded inside a base descriptor, the result is checked against the value of the base descriptor, and made relative if required.

```
<base location="/html/body">
  <rewrite
    link-description=" ../schemas/html.xrm.xml#xhtml-1.0"
    tags="images scripts" >
    <rewriting from="{{A}}/{name}.jpg" into="{{A}}/JPEG/{name}.jpg" />
    <rewriting from="{{A}}/{name}.js" into="{{A}}/javascripts/{name}.js" />
  </rewrite>
</base>
```

The example above will, for instance, change the document below

```
<html >
  <body>
```

```
<img href="http://catworld:8080/friends/garfield.jpg" />
</body>
</html>
```

into

```
<html >
  <body xml:base="http://catworld:8080" >
    <img href="JPEG/garfield.jpg" />
  </body>
</html>
```

where the *xml:base* attribute attached to the body element has been extrapolated from the static-base-uri of the input document (because no more precise information was provided)

Link Dependencies

They are described using a similar mechanism than for link transformation, through a dedicated section “dependencies” having the following attributes:

1. *link-description*: the name of a link description section, against which link tags will be interpreted (mandatory)
2. *normalize-input*: take the value yes or no (defaults to yes if omitted); if set to yes, the relevant normalization process will be performed on all links before testing operation (the exact nature of normalization operation depends upon the nature of link); if set to no, all tests will be applied on the original link²;
3. *normalize-output*: take the value yes or no (defaults to yes if omitted); if set to yes, the relevant normalization process will be performed on all links before dumping the dependency (the exact nature of normalization operation depends upon the nature of link); when set to no, minimal transformation may nevertheless occur².
4. *resolving-base*: optionally specifies an URI that will be considered as the reference URI for solving relative link. It supersedes the *xml:base* information, if present, or the static-base-uri of the original document otherwise.
5. *sorting*: takes one of the following values {“document-order”, “content-order”, “tag-order”}, and expresses the method used to order the link dependencies dumped into the dependency report. With document-order, links are organized in the same order than found inside the original input document. Using content-order, links are alphabetically classified according to the lexical structure of the URL. The flag mode use an alphabetical classification based on the tag name of the link, as defined by the link description section. If omitted, the sorting attribute defaults to “document-order”.

Note that if no *extract* sub-descriptor is provided, all links found in the input document are dumped into the dependency report.

Conclusion

We have implemented most of the features described in this proposal through an XML syntax from which the examples above are extracted, which comes with a RelaxNG schema. An XSLT

2.0 stylesheet (interpreter/compiler front-end) analyzes the specifications and generates another XSLT 2.0 stylesheet for each of the three operations (link verification, link transformation and link dependencies) ; the link description section is only interpreted during the compilation phase in order to produce the adequate code. A dedicated, home-made XSLT 2.0 library defines common operations (such as pattern matching functions), and is reused by all stylesheets including the front-end analyzer. The compiled stylesheet can be dumped for later use, or directly executed through the on-the-fly invocation mechanism offered by the Open Source Saxonica Engine [17].

Our experimental results demonstrate that the approach is realistic, useful and leads to realistic performance levels (no particular implementation issue raised).

Evaluation of the qualitative aspect of such a proposal is always a difficult issue, because strongly related to usability and far from being objective matter.

From this point of view, we were happy to observe that the verbosity of specifications turned out to be nicely under control, mainly thanks to the clear conceptual separation between link descriptors and operations, and also because we designed well-targeted default parameters and behaviors. An other fruitful principle we tried to follow was trying to capture as much as possible common and simple operations into simple abstractions, and to scale up most complex operations toward adding attributes or embedding additional information inside the element content (e.g. a simple rewrite operation can use the "from" and "into" attributes whereas a more complex rewrite operation can be decomposed into a sublist of ordered rewriting rules to try sequentially)

Regarding the expressive power, it turned out to be adequate for the cases we had to analyze. Of course, the difficult point is to extrapolate to cases we did not forecast. What we can say is that the methodology we have adopted allowed us to abstract over applications and to focus as much as possible on the functions associated with referential objects

We now consider opening the technology and related tools to a larger technical community as a service accessible through a corporate web portal, and thus to understand if it triggers interest, and hopefully to understand in a deeper way the potential enhancements and evolutions we could envision.

A. The pattern matching language

The pattern matching language we propose hereafter is based on the "{" and "}" characters to serve as delimiters of pattern variables. Those characters have no precise meaning (see the URI specification [1]) and do not belong to the standard alphabet or separator sets. Variables are named through using any identifier built from any alphabet excluding the braces and the star "*". A label can only be used once in a given pattern. If a star is used instead of a name (e.g. "{*}"), it just means that the matching substring is not stored. Double braces mean that the longest matching substring is expected, whereas the shortest match is returned for single braces.

The table below illustrates the various pattern matching mechanisms:

Table A.1.

Pattern	Value	Result
http://{server}:{*}/{*}.jpg	http://barnum:80/circus/jumper.jpg	Matches=yes ; server="barnum"
http://barnum:80/circus/acrobats/juggler.jpg	Matches=yes ; server="barnum"	
https://barnum:80/circus/jumper.jpg	Matches=no	
http://{server}/{ {path} }/{object}	http://barnum:80/circus/jumper.gif	Matches=yes server="barnum:80" path="circus" object="jumper.gif"
http://barnum:80/circus/acrobats/juggler.jpg	Matches=yes server="barnum:80" path="circus/acrobat" object="juggler.gif"	
http://{server}/{path}/{object}	http://barnum:80/circus/jumper.gif	Matches=yes server="barnum:80" path="circus" object ="jumper.gif"
http://barnum:80/circus/acrobats/juggler.jpg	Matches=yes server="barnum:80" path="circus" object ="acrobats/juggler.jpg"	

B. Verification and execution of XRM specifications (principles)

Our descriptions can be expressed through XML or any appropriate language. If the language is not based on XML, a bidirectional, lossless, translation to XML could be provided (this technique is used by the RelaxNG [14] schema language, which provides both an XML based syntax and a so-called "compact syntax", strictly equivalent).

In order to be consistent and usable, our link descriptions must comply with specific properties that can be checked in order to assess the correctness of the specifications:

1. Wellformedness of the logical structure (correct occurrence of sections, subsections and attributes)
2. Correct use of tags (no dangling tag references, coherence of tag declarations if any)
3. Correct structure of URI (reference on link descriptions)

The execution model of any processing component functionally encompasses 3 stages (points 4, 5, 6 below all cover the third stage, depending on the active operation):

4. Performs the XML parsing
5. Extracts of the so-called *base-uri* (the URL that describes the localization of the instance to be processed)
6. For each link specified into the link validation description,
 - a. Extracts the link value, using the localization information described in point 1.a above, and accessed through the tag designation mechanism

- b. Perform a partial normalization of the link, according to information provided (deals only with escaping issues, depending on the kind of reference, as specified)
- c. Verifies if the lexical structure of link meets the validation requirement, depending on those:
 - i. The link structure is compliant with the declared link type
 - ii. The link is verifying the condition (if provided)
 - iii. The link is matching the pattern (if provided)
 - iv. The link target is available (if this constraint is specified)
 - v. The link target verifies the expected properties, if any such is specified (namespace, node selection condition)
7. For each link specified into the link transformation description,
 - a. Extracts the link value, using the localization information described in point 1.a above, and accessed through the tag designation mechanism
 - b. Normalizes the link, according to the information provided by the *normalize* attribute of the link transformation section (if *normalize* is set to true, solves the relative references into absolute references, in accordance with the XML Base standard [5] ; deal with escaping issues, depending on the kind of reference, as specified)
 - c. Applies the rule logic as described above for rewriting descriptors
 - d. Normalizes the resulting link, with respect to *xml:base* mechanism, if required
 - e. Handle forbidden characters inside link content, as required by its type (use escaping mechanisms defined in [1], e.g. a space “ ” is escaped into “%20”)
 - f. Inserts the resulting link into the output document in replacement of the original link
8. For each link specified in the dependencies section,
 - a. extracts all relevant link values satisfying the filtering conditions (prior normalization if required)
 - b. normalize the link (if required by the extract sub-descriptor) and orders the links according to the specified ordering policy
 - c. creates an output report with the relevant meta-information: for instance the date and time of the dependency extraction operation ; the URL of the input document, the URL of the link dependencies specification interpreted by the operation
 - d. dumps the links in the right order inside the report with the relevant meta-information as specified by *show-tag* and *show-location* attributes

References

[1] *Uniform Resource Identifier: Generic syntax (URI)*, IETF - RFC 3986 T. Berners-Lee, R. Fielding, L. Masinter, January 2005 rfc content

[2] *XML Catalogs*, OASIS Committee specification, August 2001 specification

[3] *XML Linking Language* W3C Recommendation, June 2003, recommendation

- [4] *Extensible Markup Language (XML) 1.0 (Second Edition)* World Wide Web Consortium, 2000, specification
- [5] *XML Base* W3C Recommendation, June 2001, recommendation
- [6] *How to Write an XML Catalog File* Bob Stayton, In “DocBook XSL: The Complete Guide”, Part 1, Chapter 5 article
- [7] *XML Path Language (XPath), version 1.0* W3C recommendation, 16 November 1999, recommendation
- [8] *Internationalized Resource Identifiers (IRIs)* IETF – RFC 3987, Duerest and Suignard, January 2005, rfc content
- [9] *XML Inclusion 1.0 (XInclude - Second Edition)* W3C recommendation, 15 November 2006, recommendation
- [10] *Extensible Markup Language (XML) 1.1* W3C recommendation, 4 February 2004 recommendation (ext. entity)
- [11] *XPointer xpointer() Scheme* W3C Working Draft 19 December 2002 working draft
- [12] *XPointer Framework* W3C Recommendation, 25 March 2005 recommendation
- [13] *XML Namespaces* Wikipedia, the free Encyclopedia article
- [14] *RelaxNG* Wikipedia, the free Encyclopedia article
- [15] *Mime Media Types* IANA (Internet Assigned Numbers Authority) specification
- [16] *Mime Types File References* non normative list of mime media types and usual associated file name extensions <http://www.mimetype.org/>
- [17] *Saxonica, XSLT and XQuery processing* Michael Kay, <http://www.saxonica.com/>
- [18] *ISO Schematron, a language for making assertions about patterns found in XML documents*, Topologi , web site