Adding Intelligence to Media

XMP SPECIFICATION PART 1

DATA AND SERIALIZATION MODEL

Adobe

Extensible Metadata Platform (XMP) Specification: Part 1, Data and Serialization Model.

Contents

Preface **5**

About this document 5
- How this document is organized 5
- Conventions used in this document 6

Where to go for more information 6

1 Introduction **8**

What is metadata? 8

What is XMP? 8
- What XMP does not cover 9

2 XMP Data Model **10**

Metadata properties 10
- Naming 11

Namespaces and prefixes 11

Schemas and namespaces 11

Property values 12
- Simple types 12
- Structures 13
- Arrays 13
- Property qualifiers 14
- Language alternatives 15

3 XMP Serialization Model **16**

Serializing XMP 16
- x:xmpmeta element 17
- rdf:RDF element 17
- rdf:Description elements 17
- rdf:about attribute 17

XMP properties 18
- Simple types 18
- Structures 19
- Arrays 20
- Property qualifiers 20
- Language alternatives 21

RDF issues 22
- Unsupported features 22
- Validation 22
- rdf:about attribute 23
- Namespace URI termination 23

4 Formal Models for Implementers 24

Canonical representations as RDF 24
Basic property forms 24
Qualifiers 26

Supporting all equivalent input 30
Equivalent forms of XML 30
Equivalent forms of RDF 31
Alternative forms of structs 33

RDF parsing information 38
Collected RDF grammar 38
Top-down parsing of RDF 40

5 Implementation Guidance 48

Escaping XML markup in values 48

Using proper output encoding 49

Packet background 49

Byte-oriented packet scanning 49

Single packet rules 50

Namespace URI termination 50

Case-neutral xml:lang values 51

Distinguishing XMP from generic RDF 51

Client application policy 52
Avoiding the xml: and rdf: namespaces 52
Using local time values 53
Handling all newlines in user interfaces 53

Preface

This document set provides a complete specification for the Extensible Metadata Platform (XMP), which provides a standard format for the creation, processing, and interchange of metadata, for a wide variety of applications.

The specification has three parts:

- *Part 1, Data and Serialization Model* covers the basic metadata representation model that is the foundation of the XMP standard format. The Data Model prescribes how XMP metadata can be organized; it is independent of file format or specific usage. The Serialization Model prescribes how the Data Model is represented in XML, specifically RDF.

  This document also provides details needed to implement a metadata manipulation system such as the XMP Toolkit (which is available from Adobe).

- *Part 2, Standard Schemas*, provides detailed property lists and descriptions for standard XMP metadata schemas; these include general-purpose schemas such as Dublin Core, and special-purpose schemas for Adobe applications such as Photoshop. It also provides information on extending existing schemas and creating new schemas.

- *Part 3, Storage in Files*, provides information about how serialized XMP metadata is packaged into XMP Packets and embedded in different file formats. It includes information about how XMP relates to and incorporates other metadata formats, and how to reconcile values that are represented in multiple metadata formats.

About this document

This document, *XMP Specification Part 1, Data and Serialization Model*, provides a thorough understanding of the XMP Data Model. It is useful for anyone who wishes to use XMP metadata, including both developers and end-users of applications that handle metadata for documents of any kind.

The serialization information is vital for developers of applications (typically asset-management systems) that will generate, process, or manage files containing XMP metadata. Such developers may use either the XMP Toolkit provided by Adobe, or independent implementations. The serialization model will also interest application developers concerned about file size, or generally wishing to understand file content.

This document also provides guidelines and important information for programmers who will implement XMP metadata manipulation systems of their own.

How this document is organized

This document has the following sections:

- Chapter 1, "Introduction," explains what metadata is, and gives a brief overview of the XMP model.

- Chapter 2, "XMP Data Model," provides the formal definition of the XMP Data Model. It describes the abstract data structures that can be represented in XMP metadata.

- Chapter 3, "XMP Serialization Model," shows how the formal Data Model is written as XML, specifically RDF.

The remaining chapters are intended for programmers who are implementing XMP manipulation systems of their own, rather than using the Adobe XMP Toolkit.

- Chapter 4, "Formal Models for Implementers," provides the formal specification for the external representation of XMP using RDF, and discusses implementation issues of mapping in both directions between the XMP Data Model and RDF.
- Chapter 5, "Implementation Guidance," provides informal guidance for implementers, discussing specific issues of mapping between the XMP Data Model and RDF.

Conventions used in this document

The following type styles are used for specific types of text:

Typeface Style	Used for:
`Monospaced bold`	XMP property names. For example, **`xmp:CreationDate`**
`Monospaced Regular`	XML code and other literal values, such as value types and names in other languages or formats

Where to go for more information

See these sites for information on the Internet standards and recommendations on which XMP Metadata is based:

Dublin Core Metadata Initiative	http://dublincore.org/
Extensible Markup Language (XML)	http://www.w3.org/XML/
IETF RFC 3066, Tags for the Identification of Languages	http://www.ietf.org/rfc/rfc3066.txt
ISO 639, Standard for Language Codes	http://www.loc.gov/standards/iso639-2/
ISO 3166, Standard for Country Codes	http://www.iso.ch/iso/en/prods-services/iso3166ma/index.html
IETF RFC 3986, Uniform Resource Identifier (URI): Generic Syntax	http://www.ietf.org/rfc/rfc3986.txt
IETF RFC 2046, Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types	http://www.ietf.org/rfc/rfc2046.txt
Naming and Addressing: URIs, URLs, and so on	http://www.w3.org/Addressing/
Resource Description Framework (RDF):	http://www.w3.org/RDF/
RDF Model and Syntax Specification	http://www.w3.org/TR/rdf-syntax-grammar/

Unicode	http://www.unicode.org/
XML 1.0 Specification	http://www.w3.org/TR/2006/REC-xml-20060816/
Namespaces in XML 1.0	http://www.w3.org/TR/2006/REC-xml-names-20060816/

1 Introduction

What is metadata?

Metadata is data that describes the characteristics or properties of a document. It can be distinguished from the main *contents* of a document. For example, for a word processing document, the contents include the actual text data and formatting information, while the metadata might include such properties as author, modification date, or copyright status.

There can be gray areas where the same information could be treated as content or metadata, depending on the workflow. In general, metadata should have value on its own without regard for the content. For example, a list of all fonts used in a document could be useful metadata, while information about the specific font used for a specific paragraph on a page would be logically treated as content.

Metadata allows users and applications to work more effectively with documents. Applications can do many useful things with metadata in files, even if they are not able to understand the native file format of the document. Metadata can greatly increase the utility of managed assets in collaborative production workflows. For example, an image file might contain metadata such as its working title, description, thumbnail image, and intellectual property rights data. Accessing the metadata makes it easier to perform such tasks as associating images with file names, locating image captions, or determining copyright clearance to use an image.

File systems have typically provided metadata such as file modification dates and sizes. Other metadata can be provided by other applications, or by users. Metadata might or might not be stored as part of the file it is associated with.

What is XMP?

In order for multiple applications to be able to work effectively with metadata, there must be a common standard that they understand. XMP—the Extensible Metadata Platform—is designed to provide such a standard.

XMP standardizes the definition, creation, and processing of metadata by providing the following:

- A *Data Model*: A useful and flexible way of describing metadata in documents; see Chapter 2, "XMP Data Model.
- A *Serialization Model*: The way in which XMP metadata is written out as a stream of XML; see Chapter 3, "XMP Serialization Model.
- *Schemas*: Predefined sets of metadata property definitions that are relevant for a wide range of applications, including all of Adobe's editing and publishing products, as well as for applications from a wide variety of vendors. *Part 2, Standard Schemas* provides details of standard schemas and guidelines for the extension and addition of schemas.
- An *XMP Packet*, which packages serialized XMP for storage, generally embedded in files. *Part 3, Storage in Files*, describes XMP packets and how to embed them in various file formats.

Additional XMP features are described in separate documents; see Adobe Developer Center. These include:

- The *Adobe XMP Toolkit*, which describes Adobe's open source toolkit API for metadata developers.
- The *Adobe JavaScript Tools Guide*, which provides a reference for XMPScript, a JavaScript API for manipulating XMP metadata.
- *XMP Custom Panels*, which describes how to create a Custom Panel Description file, which gives developers the ability to define, create, and manage custom metadata properties by customizing the standard **File Info** dialog in Adobe applications that support XMP.

XMP is designed to accommodate a wide variety of workflows and tool environments. It allows localization and supports Unicode.

XMP metadata is encoded as XML-formatted text, using the W3C standard Resource Description Framework (RDF), described in Chapter 3, "XMP Serialization Model.

> **NOTE:** The string "XAP" or "xap" appears in some namespaces, keywords, and related names in this document and in stored XMP data. It reflects an early internal code name for XMP; the names have been preserved for compatibility purposes.

What XMP does not cover

Applications can support XMP by providing the ability to preserve and generate XMP metadata, giving users access to the metadata, and supporting extension capabilities.

- A number of related areas are outside the scope of XMP itself, and should be under the control of the applications and tools that support XMP metadata, although this document may make some recommendations. These areas include the following:
- The specific metadata set by each application.
- The operation of media management systems.
- The user interface to metadata.
- The definition of schemas beyond those defined by XMP.
- Validity and consistency checking on metadata properties.
- The requirement that users set or edit metadata.

Following the XMP schemas and guidelines presented in this document cannot guarantee the integrity of metadata or metadata flow. That integrity must be accomplished and maintained by a specific set of applications and tools.

2 XMP Data Model

This chapter describes the kinds of data that XMP supports.

- "Metadata properties" describes how metadata items are associated with a document in the form of *properties*.
- "Namespaces and prefixes" on page 11 discusses how properties are named and organized into groups called *schemas*.
- "Property values" on page 12 describes the data types that can be used for XMP properties.

Metadata properties

In XMP, metadata consists of a set of properties. Properties are always associated with a particular entity referred to as a *resource*; that is, the properties are "about" the resource. A resource may be:

- A file. This includes simple files such as JPEG images, or more complex files such as entire PDF documents.
- A meaningful portion of a file, as determined by the file structure and the applications that process it. For example, an image imported into a PDF file is a meaningful entity that could have associated metadata. However, a range of pages is not meaningful for a PDF file, because there is no specific PDF structure that corresponds to a range of pages. In general, XMP is not designed to be used with very fine-grained subcomponents, such as words or characters.

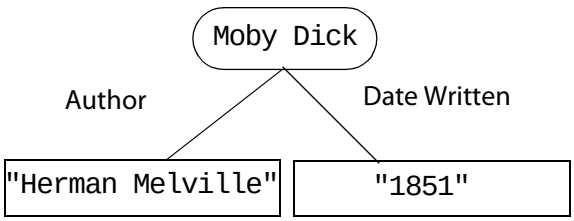
Any given property has a *name* and a *value*. Conceptually, each property makes a statement about a resource of the form

"The *property_name* of *resource* is *property_value*."

For example:

The author of *Moby Dick* is Herman Melville.

This statement is represented by metadata in which the resource is the book "Moby Dick," the property name is "author," and the property value is "Herman Melville," as in the following figure.



In the diagrams that illustrate the Data Model in this chapter, the top or root of the metadata tree is the resource—that is, the document or component to which the metadata applies.

Naming

All property, structure field, and qualifier names in XMP must be legal XML qualified names. That is, they must be well formed XML names and in an XML namespace.

The top level nodes in these illustrations have names simply for the sake of clarity for the reader. Within the XMP Data Model the resource need not have a formal name. The example namespaces used here, such as `http://ns.example.com/xyz/`, are all artificial. They do not exist and should not be used as a model for real namespaces.

> **IMPORTANT:** To work properly with RDF, all XML namespace URIs used in XMP must be terminated with "/" or "#". See "Namespace URI termination" on page 23 for details.

Namespaces and prefixes

XMP uses XML namespaces for top-level properties, struct fields, and qualifiers. This is a requirement inherited from RDF. The current specification for XML namespaces is "Namespaces in XML 1.0:"

> http://www.w3.org/TR/2006/REC-xml-names-20060816/

XML defines the fundamental notion of an *expanded name*, consisting of a namespace name and local name pair. The namespace name is a URI. The local name is an XML `NCName`, a relatively simple name that does not contain a colon or other special characters. Names are compared by textual matching of the (URI, local) pairs; two names are the same if their URIs match and their local names match.

A general XML expanded name can be in no namespace, having an empty URI part of the expanded name; RDF, however, requires that all names must be in a namespace. A name in a namespace is often informally called a qualified name, although formally that term covers all names subject to namespace interpretation.

To make the stored XML text smaller, namespace URIs are associated with a *prefix* and names are then written as *`prefix:local`*. The prefix is also an XML `NCName`. The URI and prefix are associated in stored XML using an `xmlns` attribute:

```
xmlns:dc="http://purl.org/dc/elements/1.1/"
```

It is very important to understand that prefixes are local and scoped in the XML. The prefix is only a means to look up the URI; the prefix itself is not considered in name comparison. Software must never depend on the use of a specific prefix in stored XML.

Schemas and namespaces

An XMP Schema is a set of top level property names in a common XML namespace, along with data type and descriptive information. Typically, an XMP schema contains properties that are relevant for particular types of documents or for certain stages of a workflow.

XMP Specification Part 2, Standard Schemas defines a set of standard metadata schemas and explains how to define new schemas.

> **NOTE:** The term "XMP Schema" used here to clearly distinguish this concept from other uses of the term "schema", and notably from the W3C XML Schema language. An XMP Schema is typically less formal, defined by documentation instead of a machine readable schema file.

An XMP Schema is identified by its XML namespace URI. The use of namespaces avoids conflict between properties in different schemas that have the same name but different meanings. For example, two

independently designed schemas might have a Creator property: in one, it might mean the person who created a resource; in another, the application used to create the resource.

The namespace URI for an XMP Schema must obey the rules for XML 1.0 namespaces. In addition, to operate well with RDF it must end with a '/' or '#' character. (See "Namespace URI termination" on page 23) The URI might or might not actually locate a resource such as a web page. XMP places no significance on the scheme or components of the namespace URI.

An XMP Schema has a preferred namespace prefix. Property names are often written in documentation with the prefix in the manner of XML qualified names, such as **`dc:creator`**. Following the rules of XML namespaces, use of this prefix is only a suggestion not a requirement. The actual prefix used when saving XMP might differ, and is local to the **`xmlns`** attribute that declares it.

Use of standard URI schemes is encouraged when creating namespace URIs. In order to avoid collisions, the URI should contain a component owned by the namespace creator such as an Internet domain name. Do not create namespaces in domains owned by others.

The term "top level" distinguishes the root properties in an XMP Schema from the named fields of a structure within a property value. By convention an XMP Schema defines its top level properties, the names of structure fields are part of the data type information.

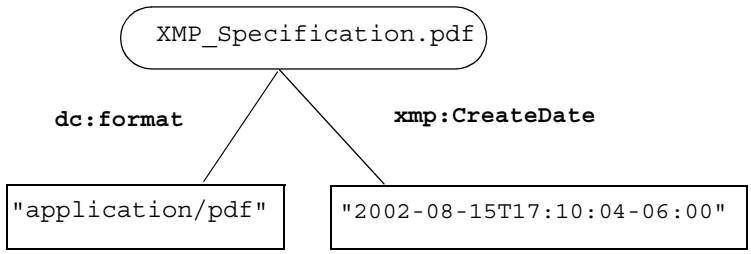
Property values

The data types that can represent the values of XMP properties are in three basic categories, described here: *simple types*, *structures*, and *arrays*. Since XMP metadata is stored as XML, values of all types are written as Unicode strings.

This section shows conceptual examples of XMP data types. "Serializing XMP" on page 16 shows how these examples are represented in XML. Definitions of all predefined properties and value types can be found in *XMP Specification Part 2, Standard Schemas*.

Simple types

A simple type has a single literal value. Simple types include familiar ones such as strings, Booleans, integers and real numbers, as well as others such as `Choice`.

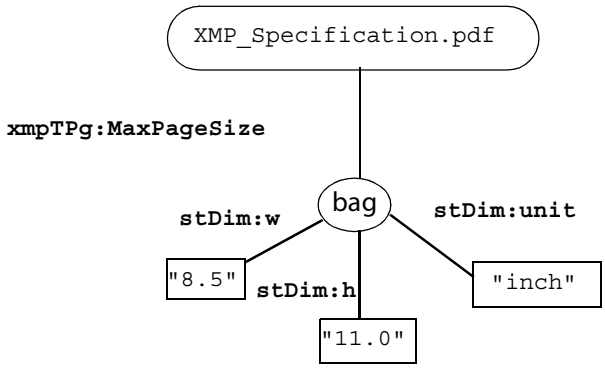


In this figure, the document `XMP_Specification.pdf` is shown with 2 simple properties:

- The value of the property **`dc:format`** is the `MIMEType` value `"application/pdf"`.
- The value of the property **`xmp:CreateDate`** is the `Date` value `"2002-08-15T17:10:04-06:00"`.

Structures

A structured property consists of one or more named fields.

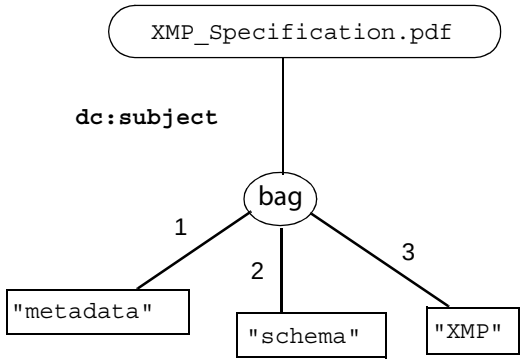


This example shows a single structured property whose type is `Dimensions`. There are three fields: **`stDim:w`** (width), **`stDim:h`** (height) and **`stDim:unit`** (units), whose values are `"8.5"`, `"11.0"` and `"inch"`.

A field in a structure can itself be a structure or an array.

Arrays

An array consists of a set of values. You can think of an array as a structure whose field names are ordinal numbers, as shown in this figure.



In addition to simple types, array elements may be structures or arrays.

XMP supports three types of arrays: *unordered*, *ordered*, and *alternative*.

Unordered arrays

An *unordered* array is a list of values whose order does not have significance. For example, the order of keywords associated with a document does not generally matter, so the **`dc:subject`** property is defined as an unordered array.

In the schema definitions, an unordered array is referred to as a *bag*. For example, **`dc:subject`** is defined as "`bag Text`", meaning that there may be multiple text-valued subjects whose order does not matter.

Ordered arrays

An *ordered* array is a list whose order is significant. For example, the order of authors of a document might matter (such as in academic journals), so the **dc:creator** property is defined as an ordered array.

In the schema definitions, an ordered array is referred to as a *seq*. For example, **dc:creator** is defined as "`seq ProperName`", meaning the order of the creators matters and each creator value is a proper name (defined elsewhere).

Alternative arrays

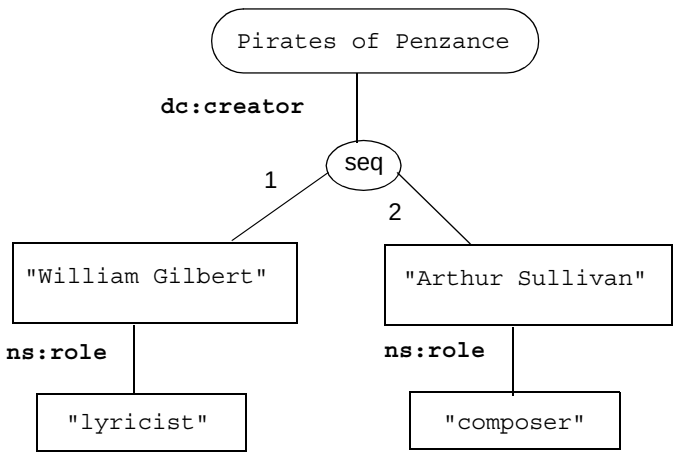
An *alternative* array is a set of one or more values, one of which should be chosen. In the schema definitions, an alternative array is referred to as an *alt*. For example, **xmp:Thumbnails** is defined as "`alt Thumbnail`". There are no specific rules for selection of alternatives: in some situations, an application may make a choice; in others, a user may make a choice. The first item in the array is considered by RDF to be the default value.

A common example is an array that contains the same logical text (such as a title or copyright) in multiple languages. This is known as a *language alternative*; it is described further in "Language alternatives" on page 15.

Property qualifiers

Any individual property value may have other properties attached to it; these attached properties are called *property qualifiers*. They are in effect "properties of properties"; they can provide additional information about the property value. For example, a digital resource representing a musical production might have one or more authors, specified using the **dc:creator** property, which is an array (see the figure below). Each array value might have a property qualifier called **ns:role**, which could take a value of `"composer"` or `"lyricist"` or possibly other values.

> **NOTE:** At this time, only simple properties may have qualifiers, and the qualifiers themselves must be simple values (not structures or arrays). This is because of limitations in early versions of the Adobe XMP Toolkit.



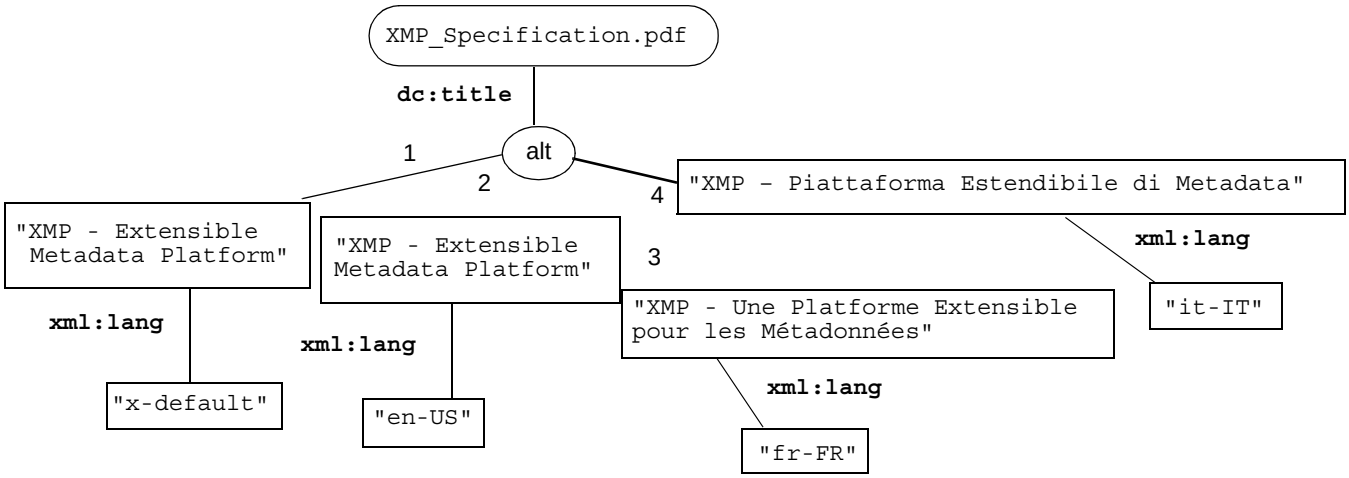
Property qualifiers allow values to be extended without breaking existing usage. For example, the ns:role qualifier in the diagram does not interfere with readers who simply want the **dc:creator** names. An alternative would be to change **dc:creator** values to structures with name and role fields, but that would confuse old software that expected to find a simple value.

The most common specific use of property qualifiers is for language alternative arrays (see next section).

Language alternatives

Language alternatives allow the text value of a property to be chosen based on a desired language. Each item in a language alternative array is a simple text value, which must have a *language qualifier* associated with it. The language qualifier is a property qualifier, as described in the previous section. The qualifier name is **`xml:lang`**, and its value is a locale string that conforms to RFC 3066 notation.

XMP uses the `"x-default"` language code to denote the default value for a language alternative. Each language alternative array must have at least the `x-default` **`xml:lang`** qualifier, and there can be only one `x-default`. It must be the first item in the array, so that generic RDF processors can use it as the default (according to the RDF default-is-first rule). The figure below shows an example:



The array should not have more than one specific value for a specific language. The order beyond the first default member is not important.

3 XMP Serialization Model

This chapter describes how XMP metadata that conforms to the XMP Data Model discussed in the previous chapter is written (*serialized*) as XML text.

- XMP properties are serialized as XML, specifically RDF. See Serializing XMP below.
- The serialized data is wrapped in packets for embedding in files. The *XMP Specification Part 3, Storage in Files*, describes the structure and capabilities of these packets.

A single XMP packet contains information about only one resource; XMP about distinct resources must be in separate packets. All of the XMP about a resource should be stored in a single XMP packet. The XMP Specification does not describe any notion of merging multiple packets about the same resource.

A packet is typically embedded in the file that the metadata describes. The manner of embedding varies according to each file format; specific file formats are discussed in *XMP Specification Part 3, Storage in Files*.

The connection between the XMP and the resource that it is about is typically physical, given by the embedding of the XMP within a file. XMP data can also be stored in a separate file from the document with which it is associated. The XMP specification does not define a general mechanism for making the connection where the XMP is not embedded. See *XMP Specification Part 3, Storage in Files* for additional details.

Serializing XMP

In order to represent the metadata properties associated with a document (that is, to serialize it in a file), XMP makes use of the Resource Description Framework (RDF) standard, which is based on XML. By adopting the RDF standard, XMP benefits from the documentation, tools, and shared implementation experience that come with an open W3C standard. RDF is described in the W3C document *Resource Description Framework (RDF) Model and Syntax Specification*.

The sections below describe the high-level structure of the XML content in an XMP Packet:

- The outermost text is the XML processing instructions and whitespace comprising the XMP Packet wrapper.
- The outermost XML element is optionally an x:xmpmeta element, which contains a single rdf:RDF element (or the `rdf:RDF` element can be outermost).
- The `rdf:RDF` element contains one or more rdf:Description elements
- Each Description element contains one or more XMP properties.

The examples in this document are shown in RDF syntax. RDF has multiple ways to serialize the same Data Model: a "typical" or verbose way, and several forms of shorthand. The examples shown here use the typical way plus a few forms of shorthand used by the Adobe XMP Toolkit; they are designed to assist human readers of stored XMP. Any valid RDF shorthand may be used, as may any equivalent XML.

XMP supports a subset of RDF; see "RDF issues" on page 22 for further information.

XMP must be serialized as Unicode. XMP supports the full Unicode character set, and is stored in files using one of the five Unicode encodings. The entire XMP packet must use a single encoding. Individual file

formats can, and generally do, specify a particular encoding, often UTF-8. For details, see the descriptions of file formats in *XMP Specification Part 3, Storage in Files*.

x:xmpmeta element

It is recommended that an `x:xmpmeta` element be the outermost XML element in the serialized XMP data, to simplify locating XMP metadata in general XML streams. The format is:

```
<x:xmpmeta xmlns:x='adobe:ns:meta/'>
    ...the serialized XMP metadata
</x:xmpmeta>
```

The `xmpmeta` element can have any number of attributes. All unrecognized attributes are ignored, and there are no required attributes. The only defined attribute at present is `x:xmptk`, written by the Adobe XMP Toolkit; its value is the version of the toolkit.

> **NOTE:** Earlier versions of XMP suggested use of the `x:xapmeta` element. Applications filtering input should recognize both.

rdf:RDF element

Immediately within the `x:xmpmeta` element should be a single `rdf:RDF` element.

```
<x:xmpmeta xmlns:x='adobe:ns:meta/'>
    <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
        ...
    </rdf:RDF>
</x:xmpmeta>
```

rdf:Description elements

The `rdf:RDF` element can contain zero or more `rdf:Description` elements. The following example shows a single `rdf:Description` element:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <rdf:Description rdf:about=""
        xmlns:dc="http://purl.org/dc/elements/1.1/">
           ... Dublin Core properties go here
    </rdf:Description>
</rdf:RDF>
```

By convention, all properties from a given schema, and only that schema, are listed within a single `rdf:Description` element. (This is not a requirement, just a means to improve readability.) In this example, properties from the Dublin Core schema are specified within the `rdf:Description` element. The `xmlns:dc` attribute defines the namespace prefix (`dc:`) to be used. Properties from other schemas would be specified inside additional `rdf:Description` elements.

The `rdf:Description` element is also used when specifying structured properties; see "Structures" on page 19.

rdf:about attribute

The `rdf:about` attribute on the `rdf:Description` element is a required attribute that may be used to identify the resource whose metadata this XMP describes. The value of this attribute should generally be

empty. Otherwise it may be a URI that names the resource in some manner that is meaningful to the application writing the XMP. The XMP Specification does not mandate or recommend any particular interpretation for this URI.

Because the `rdf:about` attribute is the only identification of the resource from the formal RDF point of view, it is useful to format a non-empty value in a standard manner. This lets generic RDF processors know what kind of URI is used. There is no formal standard for URIs that are based on an abstract UUID. The following protocol may be relevant:

http://www.faqs.org/rfcs/rfc4122.html

All elements within an `rdf:RDF` element must have the same value for their `rdf:about` attributes.

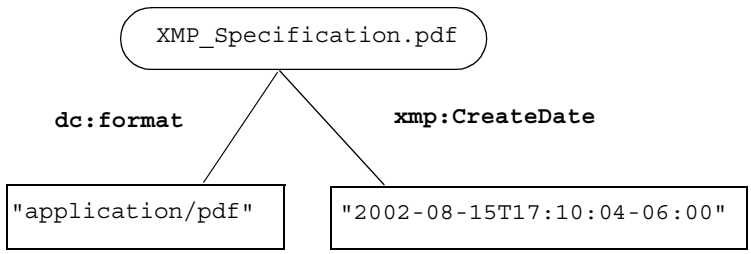
NOTE: The XMP storage model does not use the `rdf:about` attribute to identify the resource. The value is preserved, but is not meaningful to XMP. Previous versions of the XMP Specification suggested placing an instance ID here. Instead, an instance ID should now be placed in the **`xmpMM:InstanceID`** property.

XMP properties

This section shows how the properties diagrammed in "Property values" on page 12 would be serialized in XMP. The data diagrams are repeated for convenience. In these examples, the `rdf:RDF` element has been elided for brevity. The `rdf:Description` elements are kept as a convenient place for `xmlns` attributes.

Simple types

All property names must be legal XML qualified names. This example is from "Simple types" on page 12.



In XMP, these properties would specified as follows:

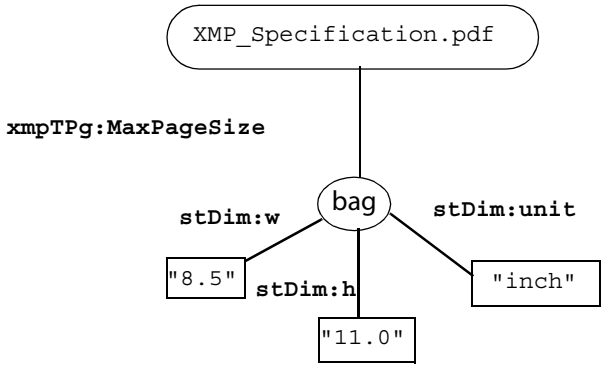
```
<rdf:Description rdf:about="" xmlns:dc="http://purl.org/dc/elements/1.1/">
   <dc:format>application/pdf</dc:format>
</rdf:Description>

<rdf:Description rdf:about="" xmlns:xmp="http://ns.adobe.com/xap/1.0/">
   <xmp:CreateDate>2002-08-15T17:10:04Z</xmp:CreateDate>
</rdf:Description>
```

Alternatively, there is a common form of RDF shorthand that writes simple unqualified properties as attributes of the `rdf:Description` element. The second `rdf:Description` element in the example would be specified as follows:

```
<rdf:Description rdf:about="" xmlns:xmp="http://ns.adobe.com/xap/1.0/"
   xmp:CreateDate="2002-08-15T17:10:04Z"/>
```

Structures

This example from "Structures" on page 13 shows a property that is a structure containing three fields.



This example would be serialized in RDF as:

```
<rdf:Description rdf:about=""
          xmlns:xmpTPg="http://ns.adobe.com/xap/1.0/t/pg/">
    <xmpTPg:MaxPageSize>
        <rdf:Description
              xmlns:stDim="http:ns.adobe.com/xap/1.0/sType/Dimensions#">
            <stDim:w>4</stDim:w>
            <stDim:h>3</stDim:h>
            <stDim:unit>inch</stDim:unit>
        </rdf:Description>
    </xmpTPg:MaxPageSize>
</rdf:Description>
```

The element hierarchy consists of:

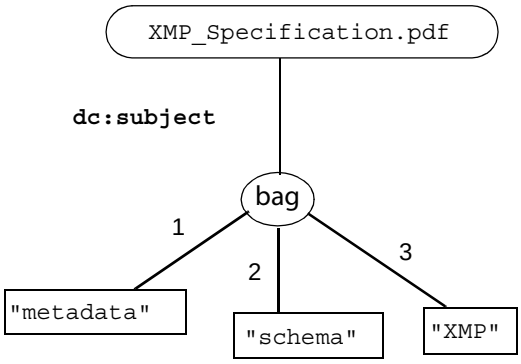
- The `rdf:Description` element, described above, which specifies the namespace for the property.
- The `xmpTPg:MaxPageSize` element, which is a property of type `Dimensions`
- An inner `rdf:Description` element, which is necessary to declare the presence of a structure. It also defines the namespace that is used by the structure fields. Inner `rdf:Description` elements do not have an `rdf:about` attribute.
- The fields of the `Dimensions` structure. All structure field names must be legal XML qualified names.

A common shorthand form of writing structures avoids the inner `rdf:Description` element:

```
<rdf:Description rdf:about=""
          xmlns:xmpTPg="http://ns.adobe.com/xap/1.0/t/pg/">

    <xmpTPg:MaxPageSize rdf:parseType="Resource"
          xmlns:stDim="http:ns.adobe.com/xap/1.0/sType/Dimensions#">
        <stDim:w>4</stDim:w>
        <stDim:h>3</stDim:h>
        <stDim:unit>inches</stDim:unit>
    </xmpTPg:MaxPageSize>
</rdf:Description>
```

Arrays

This example is from "Arrays" on page 13.



This example is serialized as follows:

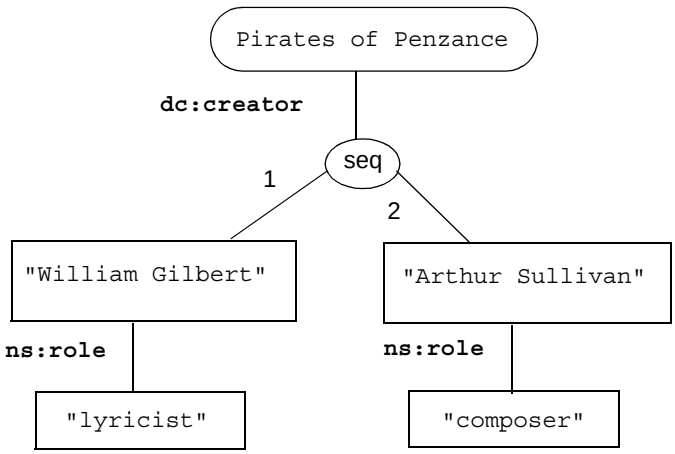
```
<rdf:Description rdf:about="" xmlns:dc="http://purl.org/dc/elements/1.1/">
    <dc:subject>
        <rdf:Bag>
            <rdf:li>metadata</rdf:li>
            <rdf:li>schema</rdf:li>
            <rdf:li>XMP</rdf:li>
        </rdf:Bag>
    </dc:subject>
</rdf:Description>
```

The **dc:subject** property is an unordered array, represented by the type `rdf:Bag`. It contains one `rdf:li` element for each item in the array. Ordered and alternative arrays are similar, except that they use the types `rdf:Seq` and `rdf:Alt`, respectively. An example of an alternative array is shown in "Language alternatives" on page 21.

Property qualifiers

Property qualifiers can generally be serialized as shown in the following figure. There is a special representation for **xml:lang** qualifiers; see "Language alternatives" on page 21.

This general example is repeated from "Property qualifiers" on page 14.



This figure shows an array with two elements, each of which has a property qualifier called **ns:role** (defined in the fictitious namespace "ns:myNamespace/"). It would be serialized as follows:

```
<rdf:Description rdf:about=""
          xmlns:dc="http://purl.org/dc/elements/1.1/"
          xmlns:ns="ns:myNamespace/">
    <dc:creator>
        <rdf:Seq>
            <rdf:li>
                <rdf:Description>
                    <rdf:value>William Gilbert</rdf:value>
                    <ns:role>lyricist</ns:role>
                </rdf:Description>
            </rdf:li>
            <rdf:li>
                <rdf:Description >
                    <rdf:value>Arthur Sullivan</rdf:value>
                    <ns:role>composer</ns:role>
                </rdf:Description>
            </rdf:li>
        </rdf:Seq>
    </dc:creator>
</rdf:Description>
```

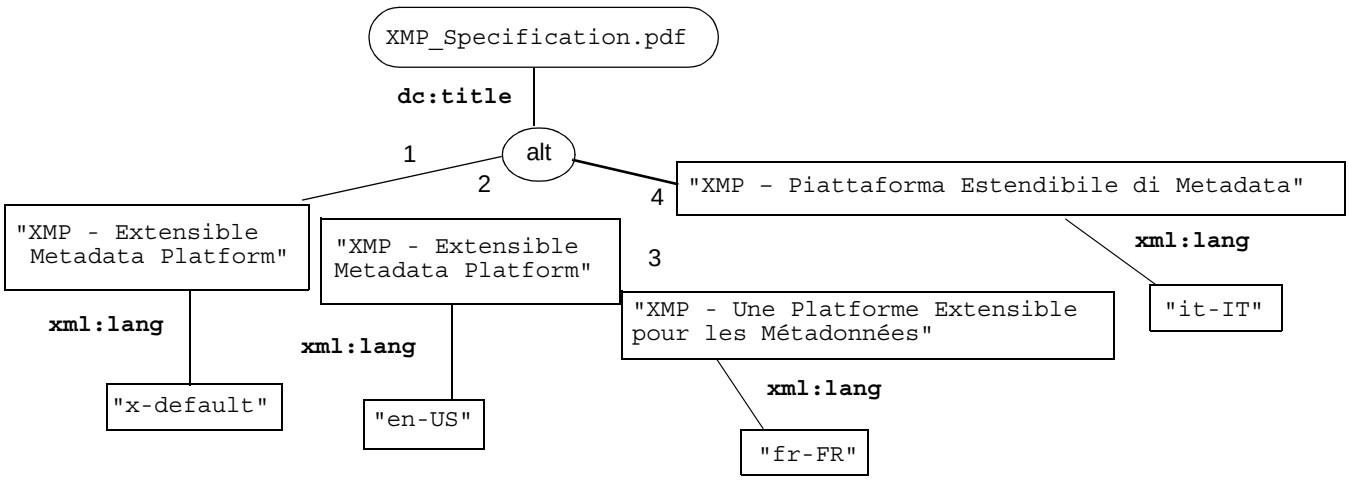
The presence of property qualifiers is indicated by a special use of the `rdf:Description` element. Each `rdf:li array` item in the example contains an `rdf:Description` element, which itself contains the following:

- a special element called `rdf:value` that represents the value of the property
- zero or more other elements that represent qualifiers of the value. In this case, there is one property qualifier called **ns:role**.

All qualifier names must be legal XML qualified names.

Language alternatives

Text properties may have an **xml:lang** property qualifier that specifies the language of the text. A common use is with language alternative arrays.



Language alternatives are a form of `rdf:Alt` array, referred to as the `Lang Alt` type. In this example, each array item is a simple text value; the value has a property qualifier, specified as the property **`xml:lang`**, giving the language of that value.

The XMP for this array looks like this:

```
<xmp:Title>
    <rdf:Alt>
        <rdf:li xml:lang="x-default">XMP - Extensible Metadata Platform</rdf:li>
        <rdf:li xml:lang="en-us">XMP - Extensible Metadata Platform</rdf:li>
        <rdf:li xml:lang="fr-fr">XMP - Une Platforme Extensible pour les
            Métadonnées</rdf:li>
        <rdf:li xml:lang="it-it">XMP - Piattaforma Estendibile di Metadata</rdf:li>
    </rdf:Alt>
</xmp:Title>
```

The **`xml:lang`** qualifier is written as an attribute of the XML element whose character data is the value (in this case, the `rdf:li` elements). Note also the special language value `"x-default"`, which specifies the default title to be used.

RDF issues

Unsupported features

XMP uses a subset of RDF. Valid XMP is limited to the RDF described in the previous sections, along with all equivalent alternate forms of that RDF. (RDF has a variety of alternative ways to represent the same information.) All XMP is valid RDF, but a number of RDF features are not valid XMP, in particular:

- The `rdf:RDF` element is required by XMP (it is optional in RDF).
- The elements immediately within `rdf:RDF` must be `rdf:Description` elements.
- The `rdf:ID` and `rdf:nodeID` attributes are ignored.
- The `rdf:aboutEach` or `rdf:aboutEachPrefix` attributes are not supported, the entire `rdf:Description` element is ignored.
- The `rdf:parseType='Literal'` attribute is not supported.
- Top-level RDF typed nodes are not supported.

Validation

If DTD or XML Schema validation is required, be aware that RDF provides many equivalent ways to express the same model. Also, the open nature of XMP means that it is in general not possible to predict or desirable to constrain the allowable set of XML elements and attributes. There appears to be no way to write a DTD that allows arbitrary elements and attributes. Even use of ANY requires declared child elements (see validity constraint #4 in section 3 of the XML specification).

The recommended approach to placing XMP in XML using DTD validation is to wrap the XMP Packet in a `CDATA` section. This requires escaping any use of "`]]>`" in the packet.

rdf:about attribute

All `rdf:Description` elements within an `rdf:RDF` element must have the same value for their `rdf:about` attributes.

Namespace URI termination

The formal definition of RDF transforms the XML representation into "triples" in a manner that catenates XML namespace URI strings with the local part of XML element and attribute names. This can lead to ambiguities if the URI does not end in separator such as '/' or '#'. This is not a problem for Adobe software, which does not utilize the triple representation. But it could be a problem in other implementations of XMP, or if the RDF form of XMP were fed to a traditional RDF processor.

Here is an artificial example of RDF that produces ambiguities in the triples:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <rdf:Description rdf:about="myName:example"
        xmlns:ns1="myName:namespace" xmlns:ns2="myName:name">
           <ns1:ship>value of ns1:ship</ns1:ship>
           <ns2:spaceship>value of ns2:spaceship</ns2:spaceship>
    </rdf:Description>
</rdf:RDF>
```

Here are the ambiguous RDF triples, notice that the two predicates are the same:

```
Subject: myName:example
Predicate: myName:namespaceship
Object: "value of ns1:ship"

Subject: myName:example
Predicate: myName:namespaceship
Object: "value of ns2:spaceship"
```

4 Formal Models for Implementers

This chapter provides formal guidance for the external representation of XMP using RDF, and the implementation issues involved with mapping in both directions between the XMP Data Model and RDF.

The chapter presents canonical examples of XMP using RDF, then discusses equivalent input issues and RDF parsing.

Canonical representations as RDF

This section provides a more concise repetition of basic structure information, along with a textual display of the underlying XMP Data Models. This Data Model display is part of the output from the DumpXMP sample provided in the Adobe XMP SDK.

The following examples provide informal canonical representations of aspects of the XMP Data Model in RDF. They are informal in that they simply illustrate one possible form of RDF. There is no requirement for any XMP processor to output precisely these forms. These examples are canonical in that they show the proper mapping between the XMP and RDF Data Models. There are multiple ways to express any single RDF Data Model in XML (see "Equivalent forms of RDF" on page 31), plus alternatives in the basic XML (see "Equivalent forms of XML" on page 30).

> **NOTE:** These examples are artificial. They are intended to illustrate the capabilities of the XMP Data Model, not specific real-world usage. The examples are not exhaustive, but should be sufficiently complete when obvious generalizations are applied, such as use of nested structs and arrays.

The first example of basic property forms shows a full XMP Packet, including the packet wrapper processing instructions and **`x:xmpmeta element`**. This is done to provide one complete example of an XMP Packet. Other examples begin with the **`rdf:RDF`** element, omitting the boilerplate packet wrapper processing instructions and **`x:xmpmeta`** element.

Basic property forms

This is an overall example showing top level simple, struct, and array properties; simple, struct, and array fields of a struct; simple, struct, and array items of an array. The XMP Data Model is fully recursive, structs and arrays can be nested to arbitrary depths. There are no inherent limits to the number of fields in a struct or the number of items in an array.

The separation of the top level properties into the ns1: and ns2: namespaces is strictly for illustration of multiple XMP Schema. The fields of a struct may be in any namespace.

XMP Data Model

```
ns:myName/1/  ns1:  (0x80000000 : schema)
    ns1:Simple = "value of ns1:Simple"
    ns1:Struct  (0x100 : isStruct)
        ns1:SimpleField = "value of ns1:Struct/ns1:SimpleField"
        ns2:StructField  (0x100 : isStruct)
        ns3:ArrayField  (0x200 : isArray)
            [1] = "value of ns1:Struct/ns3:ArrayField[1]"
            [2] = "value of ns1:Struct/ns3:ArrayField[2]"
```

```
ns:myName/2/  ns2:  (0x80000000 : schema)
    ns2:UnorderedArray-of-Simple  (0x200 : isArray)
        [1] = "value of ns2:UnorderedArray-of-Simple[1]"
        [2] = "value of ns2:UnorderedArray-of-Simple[2]"
    ns2:OrderedArray-of-Struct  (0x600 : isOrdered isArray)
        [1]  (0x100 : isStruct)
        [2]  (0x100 : isStruct)
    ns2:AlternateArray-of-UnorderedArray  (0xE00 : isAlt isOrdered isArray)
        [1]  (0x200 : isArray)
        [2]  (0x200 : isArray)
```

Canonical RDF

```
<?xpacket begin="" id="W5M0MpCehiHzreSzNTczkc9d"?>
<x:xmpmeta xmlns:x="adobe:ns:meta/">
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Description rdf:about=""
    xmlns:ns1="ns:myName/1/" xmlns:ns2="ns:myName/2/" xmlns:ns3="ns:myName/3/">

    <ns1:Simple>value of ns1:Simple</ns1:Simple>

    <ns1:Struct>
        <rdf:Description>
            <ns1:SimpleField>value of ns1:Struct/ns1:SimpleField</ns1:SimpleField>
            <ns2:StructField>
                <rdf:Description>
                    <!-- Inner fields in the obvious manner -->
                </rdf:Description>
            </ns2:StructField>
            <ns3:ArrayField>
                <rdf:Bag>
                    <rdf:li>value of ns1:Struct/ns3:ArrayField[1]</rdf:li>
                    <rdf:li>value of ns1:Struct/ns3:ArrayField[2]</rdf:li>
                </rdf:Bag>
            </ns3:ArrayField>
        </rdf:Description>
    </ns1:Struct>

  </rdf:Description>

  <rdf:Description rdf:about="" xmlns:ns2="ns:myName/2/">

    <ns2:UnorderedArray-of-Simple>
        <rdf:Bag>
            <rdf:li>value of ns2:UnorderedArray-of-Simple[1]</rdf:li>
            <rdf:li>value of ns2:UnorderedArray-of-Simple[2]</rdf:li>
        </rdf:Bag>
    </ns2:UnorderedArray-of-Simple>
```

```
    <ns2:OrderedArray-of-Struct>
        <rdf:Seq>
            <rdf:li>
                <rdf:Description>
                    <!-- Fields in the obvious manner -->
                </rdf:Description>
            </rdf:li>
            <rdf:li>
                <rdf:Description>
                    <!-- Fields in the obvious manner -->
                </rdf:Description>
            </rdf:li>
        </rdf:Seq>
    </ns2:OrderedArray-of-Struct>

    <ns2:AlternateArray-of-UnorderedArray>
        <rdf:Alt>
            <rdf:li>
                <rdf:Bag>
                    <!-- Items in the obvious manner -->
                </rdf:Bag>
            </rdf:li>
            <rdf:li>
                <rdf:Bag>
                    <!-- Items in the obvious manner -->
                </rdf:Bag>
            </rdf:li>
        </rdf:Alt>
    </ns2:AlternateArray-of-UnorderedArray>

  </rdf:Description>

</rdf:RDF>
</x:xmpmeta>
    <!-- Padding whitespace goes here -->
<?xpacket end="w"?>
```

Qualifiers

The treatment of qualifiers in RDF is not uniform and somewhat awkward. The **`xml:lang`** qualifier is treated specially, written as an XML attribute of the property that it qualifies. Other qualifiers cause the RDF for the qualified property to look like a struct with a special **`rdf:value`** field. The presence of **`rdf:value`** is what denotes this as a qualified property.

The xml:lang qualifier

This example shows **`xml:lang`** qualifiers on XMP Data Model nodes. RDF has special rules about **`xml:lang`**; it is written as an attribute of the property element that it qualifies. Use of an **`xml:lang`** qualifier on nodes that are not leaf text should probably be avoided, XMP does not define any specific interpretation of this. It is, however, valid XMP. While an **`xml:lang`** qualifier on a struct or array node might seem reasonable as an overall hint, recognizing and acting on that hint would complicate UI code.

The language values used here (such as `x-lang0`) are artificial and without meaning. The only purpose of this example is to show the use of the **`xml:lang`** attribute.

XMP Data Model

```
ns:myName/  ns:  (0x80000000 : schema)
    ns:Simple = "value"  (0x50 : hasLang hasQual)
        ? xml:lang = "x-lang0"  (0x20 : isQual)

    ns:Struct  (0x150 : isStruct hasLang hasQual)
           ? xml:lang = "x-lang1"  (0x20 : isQual)

        ns:SimpleField = "value"  (0x50 : hasLang hasQual)
           ? xml:lang = "x-lang2"  (0x20 : isQual)

        ns:StructField  (0x150 : isStruct hasLang hasQual)
           ? xml:lang = "x-lang3"  (0x20 : isQual)

        ns:ArrayField  (0x250 : isArray hasLang hasQual)
           ? xml:lang = "x-lang4"  (0x20 : isQual)
           [1] = "value"  (0x50 : hasLang hasQual)
           ? xml:lang = "x-lang5"  (0x20 : isQual)
           [2] = "value"  (0x50 : hasLang hasQual)
           ? xml:lang = "x-lang6"  (0x20 : isQual)

        ns:UnorderedArray-of-Simple  (0x250 : isArray hasLang hasQual)
           ? xml:lang = "x-lang7"  (0x20 : isQual)
           [1] = "value"  (0x50 : hasLang hasQual)
           ? xml:lang = "x-lang8"  (0x20 : isQual)
           [2] = "value"  (0x50 : hasLang hasQual)
           ? xml:lang = "x-lang9"  (0x20 : isQual)
```

Canonical RDF

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Description rdf:about="" xmlns:ns="ns:myName/">

    <ns:Simple xml:lang="x-lang0">value</ns:Simple>

    <ns:Struct xml:lang="x-lang1">
        <rdf:Description>
           <ns:SimpleField xml:lang="x-lang2">value</ns:SimpleField>
           <ns:StructField xml:lang="x-lang3">
              <rdf:Description>
                 <!-- Inner fields in the obvious manner -->
              </rdf:Description>
           </ns:StructField>
           <ns:ArrayField xml:lang="x-lang4">
              <rdf:Bag>
                 <rdf:li xml:lang="x-lang5">value</rdf:li>
                 <rdf:li xml:lang="x-lang6">value</rdf:li>
              </rdf:Bag>
           </ns:ArrayField>
        </rdf:Description>
    </ns:Struct>

    <ns:UnorderedArray-of-Simple xml:lang="x-lang7">
        <rdf:Bag>
           <rdf:li xml:lang="x-lang8">value</rdf:li>
           <rdf:li xml:lang="x-lang9">value</rdf:li>
        </rdf:Bag>
    </ns:UnorderedArray-of-Simple>

  </rdf:Description>
</rdf:RDF>
```

Other simple qualifiers

This example shows other simple qualifiers on XMP Data Model nodes. While the XMP Data Model is essentially identical to the one above for **`xml:lang`** qualifiers, the RDF is quite different. For qualifiers other than **`xml:lang`**, RDF forces the qualified property to look like a struct with a special **`rdf:value`** field that contains the actual value from the XMP Data Model. The other fields of the fake struct are the qualifiers. The presence of **`rdf:value`** is what distinguishes a real struct from a qualified property.

XMP Data Model

```
ns:myName/  ns:  (0x80000000 : schema)
    ns:Simple = "value"  (0x10 : hasQual)
        ? ns:Qual = "qualifier"  (0x20 : isQual)

    ns:Struct  (0x110 : isStruct hasQual)
           ? ns:Qual = "qualifier"  (0x20 : isQual)
        ns:SimpleField = "value"  (0x10 : hasQual)
           ? ns:Qual = "qualifier"  (0x20 : isQual)
        ns:StructField  (0x100 : isStruct)
        ns:ArrayField  (0x200 : isArray)
    ns:Array  (0x210 : isArray hasQual)
        ? ns:Qual = "qualifier"  (0x20 : isQual)
```

Canonical RDF

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Description rdf:about="" xmlns:ns="ns:myName/">

    <ns:Simple>
        <rdf:Description>
            <rdf:value>value</rdf:value>
            <ns:Qual>qualifier</ns:Qual>
        </rdf:Description>
    </ns:Simple>

    <ns:Struct>
        <rdf:Description>
            <rdf:value>
                <rdf:Description>
                    <ns:SimpleField>
                        <rdf:Description>
                            <rdf:value>value</rdf:value>
                            <ns:Qual>qualifier</ns:Qual>
                        </rdf:Description>
                    </ns:SimpleField>

                    <ns:StructField>
                        <rdf:Description>
                            <!-- Inner fields in the obvious manner -->
                        </rdf:Description>'
                    </ns:StructField>
```

```
                <ns:ArrayField>
                    <rdf:Bag>
                        <!-- Items in the obvious manner -->
                    </rdf:Bag>
                </ns:ArrayField>
            </rdf:Description>
        </rdf:value>
        <ns:Qual>qualifier</ns:Qual>
    </rdf:Description>
  </ns:Struct>

  <ns:Array>
    <rdf:Description>
      <rdf:value>
        <rdf:Bag>
          <!-- Items in the obvious manner -->
        </rdf:Bag>
      </rdf:value>
      <ns:Qual>qualifier</ns:Qual>
    </rdf:Description>
  </ns:Array>

 </rdf:Description>
</rdf:RDF>
```

Compound qualifiers

Qualifiers with compound (struct or array) values are straightforward generalizations of the above examples.

XMP Data Model

```
ns:myName/  ns:  (0x80000000 : schema)
    ns:Simple = "value"  (0x10 : hasQual)
        ? ns:Qual1  (0x120 : isStruct isQual)
        ? ns:Qual2  (0x220 : isArray isQual)
```

Canonical RDF

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Description rdf:about="" xmlns:ns="ns:myName/">

    <ns:Simple>
      <rdf:Description>
        <rdf:value>value</rdf:value>
        <ns:Qual1>
          <rdf:Description>
            <!-- Fields in the obvious manner -->
          </rdf:Description>
        </ns:Qual1>
        <ns:Qual2>
          <rdf:Bag>
            <!-- Items in the obvious manner -->
          </rdf:Bag>
        </ns:Qual2>
      </rdf:Description>
    </ns:Simple>

  </rdf:Description>
</rdf:RDF>
```

Qualified qualifiers

The XMP Data Model allows qualifiers themselves to be qualified. This is fully recursive and general. There are no artificial nesting limitations.

XMP Data Model

```
ns:myName/  ns:  (0x80000000 : schema)
    ns:Simple = "value of ns:Simple"  (0x10 : hasQual)
        ? ns:Qual1 = "value of ns:Qual1"  (0x70 : hasLang isQual hasQual)
            ? xml:lang = "x-lang"  (0x20 : isQual)
        ? ns:Qual2 = "value of ns:Qual2"  (0x30 : isQual hasQual)
            ? ns:Qual3 = "qualifier of ns:Qual2"  (0x20 : isQual)
```

Canonical RDF

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Description rdf:about="" xmlns:ns="ns:myName/">

    <ns:Simple>
        <rdf:Description>
            <rdf:value>value of ns:Simple</rdf:value>
            <ns:Qual1 xml:lang="x-lang">value of ns:Qual1</ns:Qual1>
            <ns:Qual2>
                <rdf:Description>
                    <rdf:value>value of ns:Qual2</rdf:value>
                    <ns:Qual3>qualifier of ns:Qual2</ns:Qual3>
                </rdf:Description>
            </ns:Qual2>
        </rdf:Description>
    </ns:Simple>

  </rdf:Description>
</rdf:RDF>
```

Supporting all equivalent input

An XMP processor must support a wide range of alternative input forms allowed by XML and RDF. All equivalent forms of XML must be supported. Almost all equivalent forms of RDF must be supported, although a couple of the alternative forms of arrays are not allowed in XMP.

Equivalent forms of XML

An XMP processor must comply with the XML 1.0 specification and allow all equivalent XML representations of an XMP packet. This includes but is not limited to:

- The end tag of an empty element can be elided: `<ns:empty></ns:empty>` becomes `<ns:empty/>`.
- Quotes (") or apostrophes (') can be used for attribute values.
- Allow and ignore unknown processing instructions. In particular, early versions of XMP in Adobe applications wrote an `adobe-xap-filters` processing instruction.
- The textual order of attributes within an element's start tag must not matter. (The order does matter for the "attributes" of the XMP Packet header processing instruction - note that those are not formally XML attributes.)

- The location of **xmlns** attributes and specific namespace prefixes must not matter as long as the resultant XML is equivalent. The location of **xmlns** attributes and specific namespace prefixes used for output can differ from the corresponding input, as long as the resultant XML is equivalent. However, when an XMP packet is placed within an XML document, all necessary **xmlns** attributes must be within the XMP packet. It must be possible to parse the XMP packet, including namespace identification, as an isolated fragment.
- XMP property values must be escaped as necessary on output to preserve all characters. In addition to escaping XML markup in values, this must account for the whitespace and newline handling of XML 1.0 sections 2.10 and 2.11, and the attribute value normalization of section 3.3.3. It must also account for the well-formed document and character rules of XML 1.0 sections 2.1 and 2.2. This latter bans the **RestrictedChar** set of characters.

Equivalent forms of RDF

An XMP processor must support a variety of alternative forms for RDF. An XMP processor should parse all equivalent RDF. However two of the alternative array forms are not allowed in XMP; see "Non-typedNode form of arrays" on page 37 and "Indexed form of array items" on page 37. The specific RDF used on output need not match the input, as long as it represents the same XMP Data Model.

All of these RDF alternatives are data driven. That is, whether or not they can be applied depends only on the specific Data Model, not on any external schema rules. Most of these alternatives are explicitly described in "Top-down parsing of RDF" on page 40.

Use of top-level rdf:Description

RDF and XMP allow arbitrary mixing of the top level properties among the top level rdf:Description elements. Grouping properties by the top level namespace is just a convention to improve human readability. The following examples all represent the same XMP Data Model:

Separate rdf:Description by XMP schema

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

    <rdf:Description rdf:about="" xmlns:ns="ns:myName/">
        <ns:Info>Examples of top level rdf:Description mixing</ns:Info>
    </rdf:Description>

    <rdf:Description rdf:about="" xmlns:ns2="ns:myName/2/">
        <ns2:Prop1>value 2.1</ns2:Prop1>
        <ns2:Prop2>value 2.2</ns2:Prop2>
    </rdf:Description>

    <rdf:Description rdf:about="" xmlns:ns3="ns:myName/3/">
        <ns3:Prop1>value 3.1</ns3:Prop1>
        <ns3:Prop2>value 3.2</ns3:Prop2>
    </rdf:Description>

</rdf:RDF>
```

Single rdf:Description

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
```

```
  <rdf:Description rdf:about=""
      xmlns:ns="ns:myName/" xmlns:ns2="ns:myName/2/" xmlns:ns3="ns:myName/3/">
    <ns:Info>Examples of top level rdf:Description mixing</ns:Info>
    <ns2:Prop1>value 2.1</ns2:Prop1>
    <ns2:Prop2>value 2.2</ns2:Prop2>
    <ns3:Prop1>value 3.1</ns3:Prop1>
    <ns3:Prop2>value 3.2</ns3:Prop2>
  </rdf:Description>

</rdf:RDF>
```

Arbitrary mixing

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

  <rdf:Description rdf:about=""
      xmlns:ns="ns:myName/" xmlns:ns2="ns:myName/2/" xmlns:ns3="ns:myName/3/">
    <ns:Info>Examples of top level rdf:Description mixing</ns:Info>
    <ns2:Prop1>value 2.1</ns2:Prop1>
    <ns3:Prop1>value 3.1</ns3:Prop1>
  </rdf:Description>

  <rdf:Description rdf:about=""
      xmlns:ns2="ns:myName/2/" xmlns:ns3="ns:myName/3/">
    <ns2:Prop2>value 2.2</ns2:Prop2>
    <ns3:Prop2>value 3.2</ns3:Prop2>
  </rdf:Description>

</rdf:RDF>
```

Top-level simple unqualified properties

Top level simple unqualified properties can be written as attributes of a top level **`rdf:Description`** element:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <rdf:Description rdf:about="" xmlns:ns="ns:myName/" ns:Prop="value"/>
</rdf:RDF>
```

The rdf:resource attribute

This is not really an alternative form of RDF, but an aspect of RDF for which the mapping to XMP is not immediately obvious. The value of a simple property can be written as the value of an **`rdf:resource`** attribute. This says that the property value is a URI. The fact that the value is a URI is not a qualifier in the XMP Data Model, it is sideband information. See also the **`emptyPropertyElt`** part of the RDF syntax discussion.

XMP Data Model

```
ns:myName/  ns:  (0x80000000 : schema)
    ns:Prop1 = "http://www.adobe.com/"  (0x2 : URI)
    ns:Prop2 = "http://www.adobe.com/"
```

Example RDF

```
<rdf:RDF xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'>
    <rdf:Description rdf:about="" xmlns:ns='ns:myName/'>
        <ns:Prop1 rdf:resource="http://www.adobe.com/"/>
        <ns:Prop2>http://www.adobe.com/</ns:Prop2>
    </rdf:Description>
</rdf:RDF>
```

Alternative forms of structs

There are several variations in the way structs can be represented in RDF. In the RDF syntax this shows up in the derivations from the **`nodeElement`** production. The example below shows four distinct forms plus one combination.

- The first is the canonical form, using an inner **`rdf:Description`** element.
- The second replaces the inner **`rdf:Description`** element by an **`rdf:parseType="Resource"`** attribute on the element naming the XMP struct.
- The third writes simple unqualified fields as attributes of the inner **`rdf:Description`** element.
- The fourth form removes the **`rdf:Description`** element of the third form, moving the field attributes to the XML element representing the struct. In this form the struct element must be an XML empty element. The field attribute shorthand cannot be combined with the **`rdf:parseType="Resource"`** attribute.
- The fifth form combines the first and third. The fifth example here is artificial, but this form would be useful if the struct had some simple unqualified fields and some fields with compound values and/or qualifiers.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <rdf:Description rdf:about="" xmlns:ns="ns:myName/">

        <ns:Struct1>
            <rdf:Description>
                <ns:Field1>value1</ns:Field1>
                <ns:Field2>value2</ns:Field2>
            </rdf:Description>
        </ns:Struct1>

        <ns:Struct2 rdf:parseType="Resource">
            <ns:Field1>value1</ns:Field1>
            <ns:Field2>value2</ns:Field2>
        </ns:Struct2>

        <ns:Struct3>
            <rdf:Description ns:Field1="value1" ns:Field2="value2"/>
        </ns:Struct3>

        <ns:Struct4 ns:Field1="value1" ns:Field2="value2"/>

        <ns:Struct5>
            <rdf:Description ns:Field1="value1">
                <ns:Field2>value2</ns:Field2>
            </rdf:Description>
        </ns:Struct5>

    </rdf:Description>
</rdf:RDF>
```

Order of struct fields

The order of struct fields is not significant in the XMP Data Model. An XMP processor is free to create a parsed representation in any order. Comparisons of structs must disregard order; that is, compare sorted fields, or compare the number of fields then iterate one struct for the field names and look up the corresponding field in the other by name. The following example shows equivalent, if not identical, structs:

XMP Data Model

```
ns:myName/  ns:  (0x80000000 : schema)
    ns:Struct1  (0x100 : isStruct)
        ns:Field1 = "value1"
        ns:Field2 = "value2"

    ns:Struct2  (0x100 : isStruct)
        ns:Field2 = "value2"
        ns:Field1 = "value1"
```

Example RDF

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <rdf:Description rdf:about="" xmlns:ns="ns:myName/">

        <ns:Struct1 rdf:parseType="Resource">
            <ns:Field1>value1</ns:Field1>
            <ns:Field2>value2</ns:Field2>
        </ns:Struct1>

        <ns:Struct2 rdf:parseType="Resource">
            <ns:Field2>value2</ns:Field2>
            <ns:Field1>value1</ns:Field1>
        </ns:Struct2>

    </rdf:Description>
</rdf:RDF>
```

Placement of qualifiers

The **`rdf:value`** element does not have to precede the qualifiers. As with struct fields in general, they can appear in any order. All of the alternate struct forms can be applied to the pseudo-struct for a qualified property.

The **`xml:lang`** attribute is not an explicit part of the RDF syntax productions. XMP allows it to be placed on any "named element" (the XML element that names a top level property or a struct field), on the **`rdf:li`** element for an array item, or on the **`rdf:value`** element for a qualified property.

There is some ambiguity about the placement of an **`xml:lang`** attribute for properties that also have other qualifiers. It may be placed on either the outer named element, or on the **`rdf:value`** element. An XMP processor must accept either, and report an error if it appears in both. Placement on the named element is recommended.

The RDF syntax also allows nested use of **`rdf:value`** elements in ways that in effect distribute and nest the qualifiers of the XMP Data Model. An XMP processor must flatten this usage as shown in the following example.

XMP Data Model

```
ns:myName/  ns:  (0x80000000 : schema)
    ns:Info = "RDF has ambiguities about qualifier placement"
    ns:Preferred = "value"  (0x50 : hasLang hasQual)
        ? xml:lang = "en-US"  (0x20 : isQual)
        ? ns:Qual1 = "qual 1"  (0x20 : isQual)
        ? ns:Qual2 = "qual 2"  (0x20 : isQual)
        ? ns:Qual3 = "qual 3"  (0x20 : isQual)
    ns:AsAttributes = "value"  (0x50 : hasLang hasQual)
        ? xml:lang = "en-US"  (0x20 : isQual)
        ? ns:Qual1 = "qual 1"  (0x20 : isQual)
        ? ns:Qual2 = "qual 2"  (0x20 : isQual)
        ? ns:Qual3 = "qual 3"  (0x20 : isQual)
    ns:Accepted = "value"  (0x50 : hasLang hasQual)
        ? xml:lang = "en-US"  (0x20 : isQual)
        ? ns:Qual1 = "qual 1"  (0x20 : isQual)
        ? ns:Qual2 = "qual 2"  (0x20 : isQual)
        ? ns:Qual3 = "qual 3"  (0x20 : isQual)
    ns:Perverse = "value"  (0x50 : hasLang hasQual)
        ? xml:lang = "en-US"  (0x20 : isQual)
        ? ns:Qual1 = "qual 1"  (0x20 : isQual)
        ? ns:Qual2 = "qual 2"  (0x20 : isQual)
        ? ns:Qual3 = "qual 3"  (0x20 : isQual)
```

Example RDF

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <rdf:Description rdf:about="" xmlns:ns="ns:myName/">
        <ns:Info>RDF has ambiguities about qualifier placement</ns:Info>
            <ns:Preferred xml:lang="en-US" rdf:parseType="Resource">
                <rdf:value>value</rdf:value>
                <ns:Qual1>qual 1</ns:Qual1>
                <ns:Qual2>qual 2</ns:Qual2>
                <ns:Qual3>qual 3</ns:Qual3>
            </ns:Preferred>

        <ns:AsAttributes xml:lang="en-US">
            <rdf:Description
                rdf:value="value"
                ns:Qual1="qual 1" ns:Qual2="qual 2" ns:Qual3="qual 3"/>
        </ns:AsAttributes>

        <ns:Accepted rdf:parseType="Resource">
            <ns:Qual1>qual 1</ns:Qual1>
            <rdf:value xml:lang="en-US">value</rdf:value>
            <ns:Qual2>qual 2</ns:Qual2>
            <ns:Qual3>qual 3</ns:Qual3>
        </ns:Accepted>

        <ns:Perverse rdf:parseType="Resource">
            <rdf:value rdf:parseType="Resource">
                <rdf:value rdf:parseType="Resource">
                    <rdf:value xml:lang="en-US">value</rdf:value>
                    <ns:Qual1>qual 1</ns:Qual1>
                </rdf:value>
                <ns:Qual2>qual 2</ns:Qual2>
            </rdf:value>
            <ns:Qual3>qual 3</ns:Qual3>
        </ns:Perverse>
```

```
    </rdf:Description>
</rdf:RDF>
```

RDF typedNode shorthand

The RDF syntax for a **nodeElement** says the XML element can be **rdf:Description** or anything else that is not an RDF term. (See "Top level and inner nodeElements" on page 41) Use of another element is known as a **typedNode**. This is direct textual shorthand for an **rdf:Description** element containing an **rdf:type** element. The textual substitution in RDF applies equally to the case where the **rdf:type** element appears to be the field of an XMP struct and to the case where it appears to be an XMP qualifier.

XMP Data Model

```
ns:myName/  ns:  (0x80000000 : schema)
    ns:Struct1  (0x190 : isStruct hasType hasQual)
           ? rdf:type = "ns:myName/:MyType"  (0x20 : isQual)
        ns:Field = "value"
    ns:Struct2  (0x100 : isStruct)
        rdf:type = "ns:myName/MyType"  (0x2 : URI)
        ns:Field = "value"

    ns:Prop1 = "value"  (0x90 : hasType hasQual)
           ? rdf:type = "ns:myName/:MyType"  (0x20 : isQual)
           ? ns:Qual = "qualifier"  (0x20 : isQual)
    ns:Prop2 = "value"  (0x10 : hasQual)
           ? rdf:type = "ns:myName/MyType"  (0x22 : isQual URI)
           ? ns:Qual = "qualifier"  (0x20 : isQual)
```

Example RDF

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <rdf:Description rdf:about="" xmlns:ns="ns:myName/">

        <ns:Struct1>
            <ns:MyType>
                <ns:Field>value</ns:Field>
            </ns:MyType>
        </ns:Struct1>

        <ns:Struct2>
            <rdf:Description>
                <rdf:type rdf:resource="ns:myName/MyType"/>
                <ns:Field>value</ns:Field>
            </rdf:Description>
        </ns:Struct2>

        <ns:Prop1>
            <ns:MyType>
                <rdf:value>value</rdf:value>
                <ns:Qual>qualifier</ns:Qual>
            </ns:MyType>
        </ns:Prop1>

        <ns:Prop2>
            <rdf:Description>
                <rdf:type rdf:resource="ns:myName/MyType"/>
                <rdf:value>value</rdf:value>
                <ns:Qual>qualifier</ns:Qual>
            </rdf:Description>
        </ns:Prop2>
```

```
    </rdf:Description>
</rdf:RDF>
```

Note that, although the RDF for Struct1 and Struct2 are equivalent, their XMP Data Models differ.

The conversion from the **typedNode** form to the **rdf:Description** plus **rdf:type** form is straightforward. The value of the **rdf:resource** attribute is a catenation of the **typedNode**'s namespace URI and local element name.

The conversion to the **typedNode** form can be problematic. There is no explicit indication of where to separate the **rdf:resource** value into a URI and local name. (See also "Namespace URI termination" on page 23.)

The XMP Data Model treats **typedNodes** slightly differently from generic RDF. An XMP processor must handle a non-array **typedNode** by adding an **rdf:type** qualifier. Explicit use of an **rdf:type** element as a struct field remains as a struct field. An XMP processor is not required to detect the presence of an **rdf:type** qualifier and output a corresponding **typedNode**.

Non-typedNode form of arrays

The canonical form for an XMP array is in fact an RDF **typedNode**. The non-**typedNode** form of an array is not allowed in XMP.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Description rdf:about="" xmlns:ns="ns:myName/">

    <ns:Array1>  <!-- This is a legal XMP array -->
       <rdf:Bag>
          <rdf:li>item 1</rdf:li>
          <rdf:li>item 2</rdf:li>
       </rdf:Bag>
    </ns:Array1>

    <ns:Array2>  <!-- This is equivalent RDF, but not legal XMP -->
      <rdf:Description>
        <rdf:type rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Bag"/>
        <rdf:li>item 1</rdf:li>
        <rdf:li>item 2</rdf:li>
      </rdf:Description>
    </ns:Array2>

  </rdf:Description>
</rdf:RDF>
```

Indexed form of array items

The RDF syntax does not show it, but the RDF formal model does not preserve **rdf:li** elements as such. Instead they are transformed into elements of the form **rdf:_1**, **rdf:_2**, and so on. RDF allows them to be written that way, including use of the attribute notation. The indexed form of array items, however, is not allowed in XMP.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <rdf:Description rdf:about="" xmlns:ns="ns:myName/">
```

```
        <ns:Array1>  <!-- This is a legal XMP array -->
            <rdf:Bag>
                <rdf:li>item 1</rdf:li>
                <rdf:li>item 2</rdf:li>
            </rdf:Bag>
        </ns:Array1>

        <ns:Array2>  <!-- This is equivalent RDF, but not legal XMP -->
            <rdf:Bag>
                <rdf:_1>item 1</rdf:_1>
                <rdf:_2>item 2</rdf:_2>
            </rdf:Bag>
        </ns:Array2>

        <ns:Array3>  <!-- This is equivalent RDF, but not legal XMP -->
            <rdf:Bag rdf:_1="item 1" rdf:_2="item 2"/>
        </ns:Array3>

    </rdf:Description>
</rdf:RDF>
```

RDF parsing information

This section presents the formal grammar for RDF and a walkthrough for implementing a top-down RDF parser. The parser walkthrough includes descriptions of the mapping to XMP.

Collected RDF grammar

The RDF grammar from http://www.w3.org/TR/rdf-syntax-grammar/#section-Infoset-Grammar:

```
7.1 Grammar summary

7.2.2 coreSyntaxTerms
    rdf:RDF | rdf:ID | rdf:about | rdf:parseType |
        rdf:resource | rdf:nodeID | rdf:datatype

7.2.3 syntaxTerms
    coreSyntaxTerms | rdf:Description | rdf:li

7.2.4 oldTerms
    rdf:aboutEach | rdf:aboutEachPrefix | rdf:bagID

7.2.5 nodeElementURIs
    anyURI - ( coreSyntaxTerms | rdf:li | oldTerms )

7.2.6 propertyElementURIs
    anyURI - ( coreSyntaxTerms | rdf:Description | oldTerms )

7.2.7 propertyAttributeURIs
    anyURI - ( coreSyntaxTerms | rdf:Description | rdf:li | oldTerms )

7.2.8 doc
    root ( document-element == RDF, children == list ( RDF ) )

7.2.9 RDF
    start-element ( URI == rdf:RDF, attributes == set() )
        nodeElementList
    end-element()
```

```
7.2.10 nodeElementList
   ws* ( nodeElement ws* )*

7.2.11 nodeElement
   start-element ( URI == nodeElementURIs,
       attributes == set ( ( idAttr | nodeIdAttr | aboutAttr )?,
                           propertyAttr* ) )
      propertyEltList
   end-element()

7.2.12 ws
   A text event matching white space defined by [XML] definition
   White Space Rule [3] S in section Common Syntactic Constructs.

7.2.13 propertyEltList
   ws* ( propertyElt ws* )*

7.2.14 propertyElt
   resourcePropertyElt | literalPropertyElt |
   parseTypeLiteralPropertyElt | parseTypeResourcePropertyElt |
   parseTypeCollectionPropertyElt | parseTypeOtherPropertyElt |
   emptyPropertyElt

7.2.15 resourcePropertyElt
   start-element ( URI == propertyElementURIs, attributes == set ( idAttr? ) )
      ws* nodeElement ws*
   end-element()

7.2.16 literalPropertyElt
   start-element ( URI == propertyElementURIs,
                   attributes == set ( idAttr?, datatypeAttr?) )
      text()
   end-element()

7.2.17 parseTypeLiteralPropertyElt
   start-element ( URI == propertyElementURIs,
                   attributes == set ( idAttr?, parseLiteral ) )
   literal
   end-element()

7.2.18 parseTypeResourcePropertyElt
   start-element ( URI == propertyElementURIs,
                   attributes == set ( idAttr?, parseResource ) )
      propertyEltList
   end-element()

7.2.19 parseTypeCollectionPropertyElt
   start-element ( URI == propertyElementURIs,
                   attributes == set ( idAttr?, parseCollection ) )
      nodeElementList
   end-element()

7.2.20 parseTypeOtherPropertyElt
   start-element ( URI == propertyElementURIs, attributes
                       == set ( idAttr?, parseOther ) )
      propertyEltList
   end-element()

7.2.21 emptyPropertyElt
   start-element ( URI == propertyElementURIs,
                   attributes == set ( idAttr?, ( resourceAttr | nodeIdAttr )?,
                   propertyAttr* ) )
   end-element()
```

```
7.2.22 idAttr
   attribute ( URI == rdf:ID, string-value == rdf-id )

7.2.23 nodeIdAttr
   attribute ( URI == rdf:nodeID, string-value == rdf-id )

7.2.24 aboutAttr
   attribute ( URI == rdf:about, string-value == URI-reference )

7.2.25 propertyAttr
   attribute ( URI == propertyAttributeURIs, string-value == anyString )

7.2.26 resourceAttr
   attribute ( URI == rdf:resource, string-value == URI-reference )

7.2.27 datatypeAttr
   attribute ( URI == rdf:datatype, string-value == URI-reference )

7.2.28 parseLiteral
   attribute ( URI == rdf:parseType, string-value == "Literal")

7.2.29 parseResource
   attribute ( URI == rdf:parseType, string-value == "Resource")

7.2.30 parseCollection
   attribute ( URI == rdf:parseType, string-value == "Collection")

7.2.31 parseOther
   attribute ( URI == rdf:parseType,
                string-value == anyString - ("Resource" | "Literal" |
                "Collection") )

7.2.32 URI-reference
   An RDF URI Reference.

7.2.33 literal
   Any XML element content that is allowed according to [XML] definition
   Content of Elements Rule [43] content in section 3.1 Start-Tags, End-Tags, and
   Empty-Element Tags.

7.2.34 rdf-id
   An attribute string-value matching any legal [XML-NS] token NCName.
```

Top-down parsing of RDF

Here is a walkthrough of the desired RDF parsing support. This covers all forms for the RDF, the XMP canonical form and all alternatives. The description presumes an initial raw XML parse that creates a runtime data structure of the XML. As a simplification, **`xmlns`** attributes are presumed to have been removed; the run-time data structure for the raw XML parse has propagated the namespace URIs.

The syntax and descriptions presented here are appropriate for construction of a top down parser. They are not appropriate for a bottom up parser. For example, there are significant ambiguities in the use of XML elements that are not RDF terms - they could be the **`typedNode`** form of a **`nodeElement`** or one of the 7 **`propertyElt`** forms.

NOTE: The `xml:lang`, `rdf:about`, `rdf:ID`, `rdf:nodeID`, and `rdf:datatype` attributes are special in RDF. The use of `xml:lang` is not shown in the syntax productions. As mentioned elsewhere, it is mapped to an XMP qualifier. The other attributes are specifically represented in the RDF syntax. Other than `rdf:about` for a top level `rdf:Description` element, their use in XMP is undefined at present.

Outermost element, rdf:RDF

```
7.2.9 RDF
   start-element ( URI == rdf:RDF, attributes == set() )
      nodeElementList
   end-element()

7.2.10 nodeElementList
   ws* ( nodeElement ws* )*

<rdf:RDF>
   ...
</rdf:RDF>
```

The outermost RDF element is **`rdf:RDF`**. No attributes are allowed on the **`rdf:RDF`** element. Inside **`rdf:RDF`** is a top level **`nodeElementList`**, a sequence of zero or more whitespace separated **`nodeElements`** with "top level" restrictions.

Top level and inner nodeElements

```
7.2.5 nodeElementURIs
   anyURI - ( coreSyntaxTerms | rdf:li | oldTerms )

7.2.11 nodeElement
   start-element ( URI == nodeElementURIs,
                   attributes == set ( ( idAttr | nodeIdAttr | aboutAttr )?,
                                       propertyAttr* ) )
      propertyEltList
   end-element()

<rdf:RDF>
   <rdf:Description rdf:about="">  <!-- Top level rdf:Description nodeElement -->
      <ns:Struct1>
         <rdf:Description>  <!-- Inner rdf:Description nodeElement -->
            ...
         </rdf:Description>
      </ns:Struct1>

      <ns:Struct2>
         <ns:MyType>  <!-- Inner typedNode form of nodeElement -->
            ...
         </ns:MyType>
      </ns:Struct2>

   </rdf:Description>
</rdf:RDF>
```

In the RDF syntax, a **`nodeElement`** (top level or inner) can be an **`rdf:Description`** element, or any other element that is not an RDF term. Use of an RDF term element is an error. In XMP, a top level **`nodeElement`** can only be **`rdf:Description`** (XMP restriction). A **`nodeElement`** that is not **`rdf:Description`** is a **`typedNode`**, which must be processed as described in "RDF typedNode shorthand" on page 36.

Attributes of a nodeElement

```
7.2.7 propertyAttributeURIs
   anyURI - ( coreSyntaxTerms | rdf:Description | rdf:li | oldTerms )
```

```
7.2.11 nodeElement
   start-element ( URI == nodeElementURIs,
                   attributes == set ( ( idAttr | nodeIdAttr | aboutAttr )?,
                                       propertyAttr* ) )
      propertyEltList
   end-element()
```

The attributes of a **nodeElement** can be r**df:about**, **rdf:ID**, **rdf:nodeID**, or anything else that is not an RDF term. A top level nodeElement should have an **rdf:about** attribute; the values of **rdf:about** attributes must all match (XMP restrictions). An XMP processor must allow a top level **nodeElement** element to have no **rdf:about** attribute and treat this as identical to an **rdf:about** attribute with an empty value. The **rdf:about**, **rdf:ID**, and **rdf:nodeID** attributes are mutually exclusive in the RDF syntax.

In XMP the meaning of an r**df:about** attribute on an inner **nodeElement** is reserved for future definition. The meaning of an **rdf:ID** or **rdf:nodeID** attribute on any **nodeElement** is also reserved for future definition in XMP. The handling of these attributes by an XMP processor is undefined.

XMP does not allow an **xml:lang** attribute on a **nodeElement**.

Other attributes (**propertyAttr**) of a top level **nodeElement** become simple unqualified top level properties in the XMP Data Model. Other attributes of an inner **nodeElement** become simple unqualified fields of the XMP Data Model struct property defined by the **nodeElement**, or qualifiers if the **nodeElement** represents a qualified property.

> **NOTE:** Early versions of the XMP Specification mentioned use of an **rdf:bagID** attribute. This was a feature of RDF at that time, it has since been dropped from RDF. An XMP processor should treat **rdf:bagID** as an error.

Content of a nodeElement

```
7.2.11 nodeElement
   start-element ( URI == nodeElementURIs,
                   attributes == set ( ( idAttr | nodeIdAttr | aboutAttr )?,
                                       propertyAttr* ) )
      propertyEltList
   end-element()

7.2.13 propertyEltList
   ws* ( propertyElt ws* )*

7.2.14 propertyElt
   resourcePropertyElt | literalPropertyElt |
   parseTypeLiteralPropertyElt | parseTypeResourcePropertyElt |
   parseTypeCollectionPropertyElt | parseTypeOtherPropertyElt |
   emptyPropertyElt
```

The contained elements of a **nodeElement** are a **propertyEltList** (property element list), a sequence of zero or more whitespace separated **propertyElt**s (property elements). The contained elements of a top level **nodeElement** become top level properties in the XMP Data Model. The contained elements of an inner **nodeElement** become fields of the XMP Data Model struct property defined by the **nodeElement**, or items in the XMP Data Model array property defined by the **nodeElement**. This pertains to the "valid" contained elements as defined in the rules for the various forms of a **propertyElt**.

The syntax of a **propertyElt** is somewhat complex. The various forms are not generally distinguished by their XML element name, but by attributes. Exceptions are **resourcePropertyElt**, **literalPropertyElt**, and **emptyPropertyElt**, which are distinguished by a combination of attributes and XML content. For those distinguished by attribute, **xml:lang** attributes cause some small complication. The use of

xml:lang in RDF is quite special; it is not part of the syntax productions. An **xml:lang** attribute in the RDF always maps to an **xml:lang** qualifier in the XMP Data Model.

The rules for distinguishing the **propertyElt** forms are:

- If there are more than 3 attributes (counting **xml:lang**) this must be an **emptyPropertyElt**
- Look for an attribute that is not **xml:lang** or **rdf:ID**.

  If none is found:

  - Look at the XML content of the **propertyElt**
  - If there is no content this must be an **emptyPropertyElt**
  - If the only content is character data this must be a **literalPropertyElt**
  - Otherwise this must be a **resourcePropertyElt**

  Otherwise (If an attribute is found that is not **xml:lang** or **rdf:ID**):

  - If the attribute name is **rdf:datatype** this must be a **literalPropertyElt**
  - If the attribute name is not **rdf:parseType** this must be an **emptyPropertyElt**
  - If the attribute value is **Literal** this must be a **parseTypeLiteralPropertyElt**
  - If the attribute value is **Resource** this must be a **parseTypeResourcePropertyElt**
  - If the attribute value is **Collection** this must be a **parseTypeCollectionPropertyElt**
  - Otherwise this must be a **parseTypeOtherPropertyElt**

The use of "must be" above means that the only acceptable form has been identified. Further processing must perform additional error checking to verify the specific syntax.

The resourcePropertyElt

```
7.2.15 resourcePropertyElt
   start-element ( URI == propertyElementURIs, attributes == set ( idAttr? ) )
      ws* nodeElement ws*
   end-element()

<ns:Struct>  <!-- resourcePropertyElt -->
   <rdf:Description>  <!-- nodeElement -->
      <ns:Field> ... </ns:Field>
      ...
   </rdf:Description>

</ns:Struct>

<ns:Array>  <!-- resourcePropertyElt -->
   <rdf:Bag>  <!-- nodeElement -->
      <rdf:li> ... </rdf:li>
      ...
   </rdf:Bag>
</ns:Array>
```

```
<ns:Prop>  <!-- resourcePropertyElt -->
    <rdf:Description>  <!-- nodeElement -->
        <rdf:value> ... </rdf:value>
        <ns:Qual> ... </ns:Qual>
        ...
    </rdf:Description>
</ns:Prop>
```

A **resourcePropertyElt** most commonly represents an XMP struct or array property. It can also represent a property with general qualifiers (other than **xml:lang** as an attribute). These are expressed in RDF as pseudo-structs with a special **rdf:value** "field".

In XMP the meaning of an **rdf:ID** attribute on a **resourcePropertyElt** is reserved for future definition. The handling of an **rdf:ID** attribute by an XMP processor is undefined.

A **resourcePropertyElt** can have an **xml:lang** attribute, it becomes an **xml:lang** qualifier on the XMP Data Model property represented by the **resourcePropertyElt**.

```
<ns:Prop>  <!-- resourcePropertyElt -->
    <ns:Type>  <!-- typedNode form of a nodeElement -->
        ...
    </ns:Type>
</ns:Prop>
```

A **resourcePropertyElt** can contain an RDF **typedNode**, a form of shorthand that elevates an **rdf:type** qualifier to a more visible position in the XML.

Note that the canonical array form used by XMP is in fact a **typedNode**. Because of their common usage and known semantics the array forms are more easily dealt with as direct special cases. The XMP treatment of **typedNodes** is discussed in "RDF typedNode shorthand" on page 36.

The literalPropertyElt

```
7.2.16 literalPropertyElt
    start-element ( URI == propertyElementURIs,
                    attributes == set ( idAttr?, datatypeAttr?) )
        text()
    end-element()
```

```
<ns:Prop>value</ns:Prop>  <!-- literalPropertyElt -->
```

A **literalPropertyElt** is the typical element form of a simple property. The text content is the property value. Attributes of the element become qualifiers in the XMP Data Model.

In XMP the meaning of **rdf:ID** or **rdf:datatype** attributes on a **literalPropertyElt** is reserved for future definition. Their handling by an XMP processor is undefined.

A **literalPropertyElt** can have an **xml:lang** attribute. It becomes an **xml:lang** qualifier on the XMP Data Model property represented by the **literalPropertyElt**.

The parseTypeLiteralPropertyElt

```
7.2.17 parseTypeLiteralPropertyElt
    start-element ( URI == propertyElementURIs,
                    attributes == set ( idAttr?, parseLiteral ) )
        literal
    end-element()
```

```
<ns:Prop rdf:parseType="Literal">  <!-- parseTypeLiteralPropertyElt -->
    ...
</ns:Prop>
```

The **parseTypeLiteralPropertyElt** is not allowed by XMP. It is a controversial component of RDF that requires the entire XML content of the outer element to be preserved and reconstituted as a textual literal. That is in essence, the original XML input text of all contained elements, processing instructions, comments, and character data comprise the "literal". In XMP Data Model terms this would be a simple property whose value happened to contain XML markup. The XMP approach is to serialize this with escaping, not as text which is actual XML markup.

The parseTypeResourcePropertyElt

```
7.2.18 parseTypeResourcePropertyElt
    start-element ( URI == propertyElementURIs,
                    attributes == set ( idAttr?, parseResource ) )
       propertyEltList
    end-element()

<ns:Struct rdf:parseType="Resource">  <!-- parseTypeResourcePropertyElt -->
    <ns:Field> ... </ns:Field>
    ...
</ns:Struct>

<ns:Struct>  <!-- resourcePropertyElt -->
    <rdf:Description>  <!-- nodeElement -->
       <ns:Field> ... </ns:Field>
       ...
    </rdf:Description>
</ns:Struct>
```

A **parseTypeResourcePropertyElt** is a form of shorthand that replaces the inner **nodeElement** of a **resourcePropertyElt** with an **rdf:parseType="Resource"** attribute on the outer element. This form is commonly used in XMP as a cleaner way to represent a struct.

In XMP the meaning of an **rdf:ID** attribute on a **parseTypeResourcePropertyElt** is reserved for future definition. The handling of an **rdf:ID** attribute by an XMP processor is undefined.

A **parseTypeResourcePropertyElt** can have an **xml:lang** attribute. It becomes an **xml:lang** qualifier on the XMP Data Model property represented by the **parseTypeResourcePropertyElt**.

The parseTypeCollectionPropertyElt

```
7.2.19 parseTypeCollectionPropertyElt
    start-element ( URI == propertyElementURIs,
                    attributes == set ( idAttr?, parseCollection ) )
       nodeElementList
    end-element()

<ns:List rdf:parseType="Collection">  <!-- parseTypeCollectionPropertyElt -->
    ...
</ns:List>
```

A **parseTypeCollectionPropertyElt** is not allowed by XMP. It appeared as an addition to RDF after XMP was first delivered, and does not map well to the XMP Data Model. In RDF usage, a collection models a LISP-like sequential list, with additional RDF-specific semantics.

The parseTypeOtherPropertyElt

```
7.2.20 parseTypeOtherPropertyElt
   start-element ( URI == propertyElementURIs,
                   attributes == set ( idAttr?, parseOther ) )
      propertyEltList
   end-element()

<ns:Prop rdf:parseType="..."> <!-- parseTypeOtherPropertyElt -->
   ...
</ns:Prop>
```

A **parseTypeOtherPropertyElt** is not allowed by XMP. It is an element containing an **rdf:parseType** attribute whose value is other than **Resource**, **Literal**, or **Collection**. The RDF specification says that the content of a **parseTypeOtherPropertyElt** is to be treated as a literal in the same manner as a **parseTypeLiteralPropertyElt**.

The emptyPropertyElt

```
7.2.21 emptyPropertyElt
   start-element ( URI == propertyElementURIs,
                   attributes == set ( idAttr?, ( resourceAttr | nodeIdAttr )?,
                   propertyAttr* ) )
   end-element()

<ns:Prop1/> <!-- a simple property with an empty value -->
<ns:Prop2 rdf:resource="http://www.adobe.com/"/> <!-- a URI value -->
<ns:Prop3 rdf:value="..." ns:Qual="..."/> <!-- a simple qualified property -->
<ns:Prop4 ns:Field1="..." ns:Field2="..."/> <!-- a struct with simple fields -->
```

An **emptyPropertyElt** is an element with no contained content, just a possibly empty set of attributes. An **emptyPropertyElt** can represent three special cases of simple XMP properties: a simple property with an empty value (`ns:Prop1` above); a simple property whose value is a URI (`ns:Prop2` above); or an alternative RDF form for a simple property with simple qualifiers (`ns:Prop3` above). An **emptyPropertyElt** can also represent an XMP struct whose fields are all simple and unqualified (`ns:Prop4` above).

An **emptyPropertyElt** can have an **xml:lang** attribute. It becomes an **xml:lang** qualifier on the XMP Data Model property represented by the **emptyPropertyElt**.

In XMP the meaning of **rdf:ID** or **rdf:nodeID** attributes on an **emptyPropertyElt** is reserved for future definition. Their handling by an XMP processor is undefined.

The XMP mapping for an **emptyPropertyElt** is a bit different from generic RDF, partly for design reasons and partly for historical reasons. The XMP mapping rules are:

1. If there is an **rdf:value** attribute then this is a simple property. All other attributes are qualifiers.
2. If there is an **rdf:resource** attribute then this is a simple property with an URI value. All other attributes are qualifiers.
3. If there are no attributes other than **xml:lang**, **rdf:ID**, or **rdf:nodeID** then this is a simple property with an empty value.
4. Finally this is a struct, the attributes other than **xml:lang**, **rdf:ID**, or **rdf:nodeID** are the fields.

The XMP mapping rules must be applied in the order shown. The concurrent use of **rdf:value** and **rdf:resource** is discouraged.

In the form with an `rdf:resource` attribute, the fact that the value is a URI is not a qualifier in the XMP Data Model, it is sideband information.

5 Implementation Guidance

This chapter provides informal guidance on a variety of separate topics. These generally clarify some smaller aspects of the mappings between the XMP Data Model and RDF. The chapter includes these topics:

- Escaping XML markup in values
- Using proper output encoding
- Packet background
- Byte-oriented packet scanning
- Single packet rules
- Namespace URI termination
- Case-neutral xml:lang values
- Distinguishing XMP from generic RDF
- Client application policy

Escaping XML markup in values

The following sections of the XML 1.0 specification discuss the treatment of special characters used in element character data content or attribute values:

- Section 2.1, Well-Formed XML Documents
- Section 2.2, Characters
- Section 2.4, Character Data and Markup
- Section 2.11, End-of-Line Handling
- Section 3.3.3, Attribute-Value Normalization
- Section 4, Physical Structures
- Appendix D, Expansion of Entity and Character References (Non-Normative)

These rules require that certain characters in XMP values be escaped on output.

The rules from section 2.4 reduce to escaping of '&', '<', '>', and the other characters in the RestrictedChar set. Use of CDATA sections is discouraged in XMP, there is no way to escape the presence of "]]>" in a value.

The rule from section 2.4 prohibit all ASCII controls (U+000..U+001F) except for tab (U+0009), linefeed (U+000A), and carriage return (U+000D). The prohibited controls cannot even appear as character entities.

Using proper output encoding

Most of the file formats documented in the XMP Specification require UTF-8 encoding of the XMP. When the file format is known and has a required encoding, that encoding must be used when writing, regardless of the input encoding of the XMP. When the file format is unknown or there is no required encoding, the input encoding must be preserved.

Packet background

XMP packets are discussed in detail in *XMP Specification Part 3, Storage in Files*.

An XMP packet must be placed in a file as a normal part of the file format. Many file formats have open extension mechanisms. These generally have some file format specific information and a blob of user data. The XMP packet is just the user data portion. An XMP packet is not magic, it cannot be blindly tacked onto arbitrary files.

The XML processing instructions and **`x:xmpmeta`** element at the outside of an XMP packet should not be required when parsing. For example, if you are passing serialized XMP between modules at runtime there should be no need to use a full packet, `<rdf:RDF>...</rdf:RDF>` should be sufficient.

An XMP reader should always use file format knowledge if possible. The XML processing instructions serve mainly to identify the packet when byte-oriented packet scanning must be used. The **`x:xmpmeta`** element serves mainly to identify XMP placed within an XML document, to differentiate it from other possible use of RDF.

Byte-oriented packet scanning

It is always best to use format-aware file parsing when possible. Byte scanning for XMP packets is not recommended; it is slow and unreliable, and should be used only if absolutely necessary.

Some file formats can have multiple XMP packets. Packet scanning can, at best, find what *might* be the main packet; but what you think is the main might not be. When you must use packet scanning, apply some heuristics to improve the odds of picking the main. Do not simply pick the first or last. This general process is suggested:

- Scan for all packets, put them in a list of candidates.
- Use the **`xmpMM:Manifest`** arrays to winnow the candidates. The manifest array items are structs whose fields are properties from the referenced XMP. If packet A's manifest mentions packet B, then B can't be the main. Be careful to not remove B though until you've also removed everything referred to by B's manifest. A two step mark/remove process is best.
- If there is still more than one candidate, use the **`xmp:MetadataDate`** property to pick the latest. If a packet has no **`xmp:MetadataDate`**, keep it in the set of candidates.
- If there is still more than one candidate, pick the last writable packet (in terms of file offset), or last overall if all are read-only.

For additional details of packet-scanning strategies, see *XMP Specification Part 3, Storage in Files*.

Single packet rules

All of the XMP directly about a specific resource should be stored in a single XMP packet. The XMP specification does not describe any notion of merging multiple packets about the same resource. A single XMP packet contains information about only one resource. The XMP about distinct resources must be in separate packets.

The connection between the XMP and the resource that it is about is typically physical, given by the embedding of the XMP within a file. The XMP specification does not define any general mechanism for making this connection where the XMP is not embedded.

Some file formats are compound, or container, formats. These have an inner structure and might contain contributions from other files in whole or in part. For example a PDF file is structured in pages and can contain images that came from JPEG files. Parts of the file structure or contained content can have local XMP. There should be one XMP packet that is about the overall file, the "main" packet. There might also be separate packets about portions of the file structure or contained content.

Namespace URI termination

This section expands on information given in "RDF issues" on page 22 and "Namespace URI termination" on page 23.

The formal definition of RDF transforms the XML representation into "triples" in a manner that catenates XML namespace URI strings with the local part of XML element and attribute names. This can lead to ambiguities if the URI does not end in a separator such as '/' or '#'. This is not a problem for Adobe software, which does not utilize the triple representation. But it could be a problem in other implementations of XMP, or if the RDF form of XMP were fed to a traditional RDF processor.

Here is an artificial example of RDF that produces ambiguities in the triples:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
   <rdf:Description rdf:about="myName:example"
          xmlns:ns1="myName:namespace" xmlns:ns2="myName:name">
      <ns1:ship>value of ns1:ship</ns1:ship>
      <ns2:spaceship>value of ns2:spaceship</ns2:spaceship>
   </rdf:Description>
</rdf:RDF>
```

Here are the ambiguous RDF triples from the RDF Validator (http://www.w3.org/RDF/Validator/), notice that the two predicates are the same:

```
Subject: myName:example
Predicate: myName:namespaceship
Object: "value of ns1:ship"
```

```
Subject: myName:example
Predicate: myName:namespaceship
Object: "value of ns2:spaceship"
```

Case-neutral xml:lang values

The values of xml:lang qualifiers, and some standard XMP properties, must obey the rules for language identifiers given in IETF RFC 3066 (http://www.ietf.org/rfc/rfc3066.txt). Implementers should pay particular attention to these aspects:

- RFC 3066 supports both 2 and 3 letter primary subtags as defined by ISO standard 639 and ISO standard 639-2.
- When a language has both an ISO 639-1 2-character code and an ISO 639-2 3-character code, you must use the tag derived from the ISO 639-1 2-character code.
- All tags are to be treated as case insensitive; there exist conventions for capitalization of some of them, but these should not be taken to carry meaning. For instance, ISO 3166 recommends that country codes are capitalized (MN Mongolia), while ISO 639 recommends that language codes are written in lower case (mn Mongolian).

Since the values must be treated as case insensitive, XMP processors are allowed to normalize them on input and to output the normalized values. The recommendations of ISO 639 and ISO 3166 are preferred. The XMP Specification does not define or require a normalization policy. Since comparisons must be case insensitive, differences in policy can have no substantive effect.

Implementers must not assume that all real language tags are in the language-country or language-dialect format. It is also not always meaningful to consider the primary subtag as a generic base. Appropriate values must be used for individual languages and countries. For example, Norwegian has distinct dialects that cannot be interchanged, a reader or speaker of one cannot necessarily comprehend another.

Distinguishing XMP from generic RDF

It is important to understand XMP by starting with the XMP Data Model. This can be serialized as RDF, using a subset of RDF and with specific semantic restrictions. All instances of XMP serialized as RDF are valid RDF. One standard test of XMP is to pass it through the RDF Validator: http://www.w3.org/RDF/Validator/.

There are portions of RDF that are outside of XMP usage, and not accepted by XMP processors. Of the RDF that XMP does allow, there are aspects that XMP processors tend to not use or to use differently from the broader RDF community - especially from those primarily interested in Semantic Web development. Some of these differences arise from the local file orientation of XMP, versus the web orientation of RDF. Others arise from the XMP preference to control implementation complexity, promote usage consistency, and operate without formal schema knowledge.

Specific examples of differences include:

- XMP's preference for an empty `rdf:about` value
- XMP's packet-oriented requirement for consistent `rdf:about` values
- XMP's requirement for arrays instead of repeated properties
- XMP's indifference towards `rdf:ID`, etc.

The XMP Specification outlines the restrictions on RDF usage. The RDF parsing description in this document provides clarification and additional details. General incorporation of outside RDF into XMP is not seen as a customer priority, making the advantages of the restrictions worthwhile.

As noted in relevant subsections of "Top-down parsing of RDF" on page 40, the meaning of `rdf:about` on inner `nodeElements` and of `rdf:ID`, `rdf:nodeID`, and `rdf:datatype` on any element is reserved in XMP for future definition. Use of these attributes should be avoided until defined.

One area worth being aware of involves XMP's requirement for explicit arrays instead of repeated properties. This shows up in differences between XMP's representation of some Dublin Core properties and the RDF representation of Dublin Core found elsewhere. XMP requires explicit arrays for properties that have multiple values, such as `dc:creator` (authors) or `dc:subject` (keywords). Other parts of the RDF and Dublin Core communities allow these to be a sequence of simple values. Or even a mixture of arrays and simple values.

```
<!-- Required XMP usage -->
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Description rdf:about=""
      xmlns:dc="http://purl.org/dc/elements/1.1/">

    <dc:subject>
       <rdf:Bag>
          <rdf:li>one</rdf:li>
          <rdf:li>two</rdf:li>
       </rdf:Bag>
    </dc:subject>

  </rdf:Description>
</rdf:RDF>


<!-- Other usage, not allowed in XMP -->
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <rdf:Description rdf:about=""
          xmlns:dc="http://purl.org/dc/elements/1.1/">

       <dc:subject>one</dc:subject>
       <dc:subject>two</dc:subject>

    </rdf:Description>
</rdf:RDF>
```

Client application policy

This section documents some issues of client application policy. These are typically issues beyond the domain of basic XMP services. It includes these topics:

- Avoiding the xml: and rdf: namespaces
- Using local time values
- Handling all newlines in user interfaces

Avoiding the xml: and rdf: namespaces

The **`xml:`** (`http://www.w3.org/XML/1998/namespace`) and **`rdf:`** (`http://www.w3.org/1999/02/22-rdf-syntax-ns#`) namespaces should only be used for well defined structural purposes. They should not be used for general user-defined XMP properties or qualifiers. An XMP processor might treat their general use as an error.

Using local time values

It is recommended that you use local times, with a time zone designator of `+hh:mm` or `-hh:mm`, instead of `Z`. This promotes human readability. For example, for a file saved in Los Angeles at 10 pm on July 23, 2005, a timestamp of `2005-07-23T22:00:00-07:00` is understandable, while `2005-07-24T05:00:00Z` is confusing.

Handling all newlines in user interfaces

The way a user interface handles newlines in text values is important to the global and cross-platform portability of XMP. When displaying text, applications should recognize common newline characters and sequences and ensure that they display as such. One technique is to modify the displayed text, substituting appropriate local newlines. You must take care, however, that the stored XMP value is not modified simply as a result of display.

Typical newlines are a single ASCII linefeed (LF, U+000A), a single ASCII carriage return (CR, U+000D), or ASCII CR-LF. Section 2.11 of the XML 1.0 specification includes other sequences as recognized newlines for normalization purposes: U+0085, U+2028, and the pair U+000D U+0085.

It is recommended that applications store all newlines in XMP text values as ASCII linefeed.