XDML - an extensible markup language and processor for XDM

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

David Lee, Epocrates, Inc. <dlee@epocrates.com >

Abstract

XDML is a set of rules how XDM values can be built which are more useful entities as compared to ordinary XDM values. The key idea is to insert into the XDM values control information which guides the interpretation and processing of the data. In particular, it structures the XDM value into named parts and associates these parts with metadata. The control information is evaluated by an XDML processor, which reports and processes the data accordingly. The processor is viewed as a set of processing components controlled by matching components of metadata. The net result of this approach is to enable the creation of self-describing XDM values: they encode the way how they are presented to applications, as well as how they should or might be processed. This means that the XDM producer - e.g. XQuery programs - can emit "rich" data whose downstream processing is significantly simplified.

Table of Contents

Introduction

XDM [W3C XDM] is the data model of the major XML processing languages – XPath, XQuery and XSLT. The model is marked by a bold simplicity: (a) every value is a sequence of items, (b) an item is either an XML node or an atomic value, (c) there are seven kinds of XML nodes and (d) a few dozens

atomic types. This means that the size and complexity of an XDM value is virtually unlimited, and at the same time that any value can be decomposed into a linear sequence of building blocks, the items. "XDM item" is an abstraction enabling us to regard a single byte and a huge XML document as just two instances of the same building block.

One can look at the XDM from three different perspectives. The first one regards XDM as a component of those processing languages, concerning only writers of XPath expressions, XQuery scripts or XSLT stylesheets. We suspect that the majority of software developers and architects would subscribe to this view.

The second perspective takes into account that input and output of those languages is also XDM and accepts the XDM as a player in the game of process integration. This perspective pays attention to the issue of translating information back and forth between XDM values and other data models, for example the data models of general purpose programming languages. It should also take an interest in the serialization of XDM values.

A third perspective makes a step from looking at the XDM as either a local affair of specialized languages or a challenge for data mapping. This new perspective regards the XDM as the foundation for building a new kind of resource, offering some particular advantages in comparison to other resource types – e.g. XML documents, relational tables or CSV files. At the same time it gives a boost to XQuery, as XQuery is the XDM producer par excellence. Increased importance of XDM means increased importance of XQuery.

Ironically, the key step toward a new appreciation of the XDM is awareness of its fundamental limitation: there is no structure – only a flat sequence of items; there is no meta information – only items and nothing else. An XDM value has something in common with a string – no limitation of size and complexity, but unless a creative step is taken there is no general way how to impose and detect a structure (above the level of its building blocks, that is). Concerning strings, the creative step was the invention of markup: divide the sequence of atoms (characters) into sections of primary information and those of meta information, the latter also known as markup. One might consider doing something equivalent with XDM values, where the atoms are XDM items, rather than characters. We want to explore the potential of such an approach. Based on prior experimental work, we propose a simple markup language and an infrastructure evaluating it. A prototypic reference implementation is a work in progress, and our main intent is to open a discussion.

Why ask for XDM (if we have XML)?

Let us assume a consumer's perspective. Scenario: some processing yields a result. This might be an XML document, a sequence of XML documents, or an XDM value. The last alternative is clearly the most general one, as any sequence of XML documents is an XDM value. But do we really need this alternative, if we consider the expressiveness of XML?

From a theoretical point of view, the answer is no: whatever you can encode as an XDM value you can translate 1:1 into an equivalent representation consisting of a single XML document. For example, the following rules would suffice: (a) the XDM items are represented by children of the document element; (b) a dedicated type attribute on these children encodes the item type. Clearly - XDM values cannot express more than a single XML document, if some simple conventions are accepted. We turn to the practical side and consider the *usage* of the results. Can XDM under certain circumstances provide more convenient access to the units we need, or can it deliver units which are a closer fit to what is actually needed?

Atomic values

A striking difference between XML and XDM is that the latter supports atomic values. This is a concrete advantage: if the desired result is one or several atomic values, then XDM can explicitly deliver them as such, whereas in the case of an XML result they must be extracted. Extraction requires knowledge about the result document structure and involves non-trivial instruments like an XPath or DOM API. A further drawback of the XML variant is computational overhead. Conclusion: in cases where the result consists wholly of atomic values, XDM is probably the more suitable format.

Collection-like data

The second difference between XML documents and XDM values is that documents are logical *trees* within which everything is related to everything; whereas an XDM value is a *collection* of independent entities. What if the result is just that, conceptually, a collection? Then the main concern is fast and convenient access to the individual parts, as well as the possibility to process them – e.g. update them – in safe isolation. Typical examples for collection-like results are:

- a heterogeneous result, the parts of which are used in different ways
- a large result, only selected parts of which are used

So the need for differential or selective processing calls for a collection-like result. Arrays and maps come to mind, supporting index or name based access to self-contained units. As we have seen, it is easy to mimic collections with XML documents. This amounts to an "XML-as-a-container" approach. Under many circumstances, this may be a perfect solution. But there are issues that may become important:

- the access to parts is XPath-based, rather than name- or index-based
- the whole result tree must be constructed in memory (unless streaming processing is used)
- local modification of the result means updating a large document

XPath-based access is inconvenient, compared to name- or index-based access. It may also be less efficient. The need to construct the whole result tree is a real drawback *if* such a construction is not required for other reasons anyway. This must not be the case. If the result is available in serialized form, then it makes a big difference if the whole result must be turned into an in-memory tree, or if small, independent parts can be located and selectively expanded. And the required parsing may be extremely fast if the parser is able to locate the desired parts without parsing the details of the preceding parts.

Is XDM a good alternative? Not or not yet. The lack of structure and metadata turn XDM into an awkward format: it resembles a Java array of type Object[]. And there is not yet a serialization format available, let alone a parser to read such a format. If XDM is to excel as a collection-like format, these problems – no structure, no metadata, no serialization – would have to be solved.

Updatable result

In pipelined processing, it is a common requirement to receive the result of a preceding step, modify it locally and pass it on to the next step. If the result is a collection of self-contained parts, such local updating is easier in several aspects, compared to the updating of a monolithic document. XDM looks promising for such purposes, but the difficulty of selective access – no structure, no metadata – reduces the attractiveness.

Continuous result

Some resources grow continuously by appending more data. Log data are the classical example. Such data, as any other data, may be desired to be XML, so as to enable XML processing. But continuous resources must not be an XML document, as it is impossible to append data to a document, they must be inserted, which is much more difficult. In this case, XDM (a lossless serialization provided) is an obvious solution, as you can append items to an XDM value without difficulty.

Result as an XDM provider

XDM is the input format for XPath, XQuery and XSLT. In pipelined processing, one step might produce a result which provides XDM input for another step – either the value as a whole is used, or one or more subsequences of it. In this scenario, an XDM result is convenient and natural. Dependent on the type of the required input XDM , an XDM result may be a better alternative than an XML result.

Wrapping up

XML documents should not be the only option for encoding the result of XML processing. No native representation of atomic values, the tight coupling implied by overall tree structure and the inability for plain appending must not be ignored. XDM is an interesting alternative, as it is a superset of XML and addresses those issues. But XDM is, as we said, an awkward format due to lack of structure and metadata. Thus we came to explore the possibility of *augmenting* XDM: add to it control information which imparts structure and enriches the data with metadata. The goal is to combine XDM's built-in advantages – support for atomic values, collection-like nature, being appendable and being a natural XDM provider – with structure and metadata enabling convenient and guided access to the contents, as well as safe and efficient local update.

XDM structure

Partitioning an XDM value

Consider the situation that an XDM value should convey two code lists, each one represented by a sequence of string items. XDM offers no way to tell where one list ends and the other begins. Similar example: the XDM value is a sequence of XML documents which represent the log data gathered during one hour - how to identify the subsequence corresponding to one day of operation? A quick and simple solution is to insert into the XDM value additional items which delimit subsequences. These items can be regarded as *control items*, to be distinguished from the original data items. The subsequences are parts of the XDM value which have been turned into new units of information. In order to give names to these parts, we add a "name" attribute to the respective control item.

```
Example: a simple way to delimit subsequences and associate them with names.

<xm:part xmlns:xm=”http://www.xdml.org/ns” name=”alpha-codes”/>,
abc,
bcd,
cde,

<xm:part xmlns:xm=”http://www.xdml.org/ns” name=”beta-codes”/>,
c001,
c002
```

And if the uniqueness of part names is not guaranteed, an optional "partID" attribute may accompany the mandatory "name" attribute.

Imposing tree structure

The shown use of control items defines parts of an XDM value in an intuitive way: the contents of a part is simply all items following the part definition and preceding the next part definition, or all following items, if this was the last part definition. But we might also allow "complex parts" - parts containing parts, to be distinguished from simple parts which contain only data items. To encode this structural model, we choose a simple rule: the contents of a complex part ends before an item explicitly "closing" the part, whereas the contents of a simple part is delimited implicitly: it ends before the next control item defining a new part (simple or complex) or closing the surrounding complex part. Note that these parts - simple or complex - are defined in a "streaming" fashion - contents are not children, but a subsequence of items delimited by an item recognized as start point and another item explicitly or implicitly meaning an end point (or the end of the XDM value, as a special case).

In order to keep things simple, we constrain the definition of complex parts: they must not contain data items outside of contained parts. In other words: parts must not be mixed, their content is either a sequence of data items, or a sequence of parts which may be simple or complex.

Leaving out namespace declarations

For brevity, all further examples will leave out the namespace declaration xmlns:xm=”…”.

```
Example of an XDM value with a two-level structure.

<xm:complexPart name="code-lists"/>,
<xm:part name="alpha-codes"/>,
abc,
bcd,
cde,
<xm:part name="beta-codes"/>,
c001,
c002,
<xm:complexPartEnd/>,

<xm:complexPart name="logs"/>,
<xm:part name="log" />,
<log>...</log>,
<xm:part name="log" />,
<log>...</log>,
<xm:complexPartEnd/>
```

Concept: Information units

We have seen how the insertion of control items can partition an XDM value into parts. To denote the concept of such parts we introduce the term *information unit*. An information unit is encoded by a sequence of XDM items. According to whether whose items contain nested units, two kinds of information units are distinguished. A *simple* information unit contains only data items, but not any nested information units. A *complex* information unit, on the other hand, contains other information units, but no data items outside of nested units.

An information unit has the following properties:

- a name
- a part ID (optional)
- metadata
- value

Name and part ID we constrain to be a QName and NCName, respectively; metadata are introduced in the next section. The value is

- a sequence of data items - in the case of a simple unit
- an unordered collection of information units - in the case of a complex unit

Note that this definition renders the sequence of child information units irrelevant, as these units are associated with names. This corresponds to the modeling practice of XML attributes or JSON members.

XDM metadata

Why and how add metadata?

We saw that control items may structure XDM values into information units, which are groups of items or of other information units. These units are entities which do not exist in XDM values without control items. Often they will serve as units of processing, and it is reasonable to expect that different units may be subjected to different processing. Such considerations suggest the usefulness of metadata.

In fact, it is very simple to associate the units with as many metadata as one would like. Every unit is preceded by a control item which amounts to a convenient container where to place those metadata, either as attributes or as child elements. Come to think of it, the control item can be regarded as a full-scale XML document which is still hardly constrained in its contents: only the name of the root element and the use of a "name" and a partID" attribute are specified, so far. This document is dedicated to defining a unit, and it is ready to be filled with metadata describing the unit.

Returning to the example given above, the units containing a single document of log data may be associated with metadata "startTime" and "devices". To accommodate such data, we can use attributes and child elements of the markup item.

```
Example of information units with metadata.

<xm:complexPart name="logs"/>,

<xm:part name="log" xmlns:e=”e.com” e:startTime="2010-12-30T08:00:00" xmlns:e="
   <e:devices>…</e:devices>
</xm:part>,
<log>...</log>,

<xm:part name="log" xmlns:e=”e.com” e:startTime="2010-12-30T09:00:00" xmlns:e="
   <e:devices>…</e:devices>
</xm:part>,
<log>...</log>,

<xm:complexPartEnd/>
```

A component model of metadata

We have arrived at a very simply method how to impose structure on XDM values, and we have found a slot into which one might throw any amount of metadata pertaining to the emerging units. Now we face two alternatives. We might stop here and regard the semantics of metadata as the realm of proprietary extensions of our simple, general model, in the same way as XML Schema allows annotation attributes. We might, for example, say that any additional attributes and child elements of control items are meta information, to be evaluated in a proprietary way.

But we can also take a different path and attempt to arrive at a generic model of XDM metadata and its processing by a responsive infrastructure. This approach does not remove the option of proprietary extensions, but factors them out and constrains them in a way which allows a generic "XDM parser" to report them in a structured way. The basic principle of such a model is to distinguish metadata meant to control a specific processing from other metadata. The latter might be called "descriptive metadata" and is available for variable uses. The former – "control metadata" – has a defined impact on a defined processing.

Why should one associate data with information which controls their processing? We note an interesting analogy. A key concept of object orientation is to associate data sets with behavior. This is similar to what we try to do. The behavior of objects is implemented by methods; the "behavior" of information units resides in control metadata which define a processing. Control metadata is behavior encoded as data, as opposed to methods which are behavior encoded as code. To get a more practical motivation, imagine writing an XQuery program and regretting the limitations of XQuery. For example, one cannot call XSLT to accomplish some finalization, one cannot trigger actions with side effects (like the execution of the SQL just composed), and one cannot create a map object which the calling application would really like to receive. In this situation there is a way out: let the query code rely on a postprocessing of the query result which is *defined* by the query and *executed* by infrastructure. Our model of XDM metadata amounts to a framework for this approach.

Obviously, control metadata and the responsive infrastructure must be modeled as a coherent whole. We assume a set of *infrastructure components* with specific capabilities and group metadata

accordingly, defining *metadata components*. Later we will make a concrete proposal for a standard set of metadata components, to be evaluated by a standard processor which implements the required processing components. But at this point of our argument we want to separate the general idea from our elaboration of it, as we want to protect the value of the idea from the possible weaknesses of our attempts to refine it. For the time being, we remain abstract. We assume an infrastructure supporting n types of XDM processing, P1, …, Pn. Each one of them is controlled by a matching component of metadata, M1, … Mn. When processing the XDM value or parts of it, the infrastructure component inspects the matching metadata component and lets it control the processing details.

XDML - the concept

By now we have collected a set of ideas which can be assembled into a comprehensive concept how XDM is turned into a *language* designed to encode information content as well as information processing. XDM is turned into a language by defining and constraining the way how control items can be used within an XDM value. To denote this language we use the acronym "XDML" (short for: "XDM markup language"). An XDML value is then an XDM value which uses control items in a way consistent with the rules of the language.

We distinguish between the *concept* of an XDML language and a concrete specification of the language. While we offer a first proposal for such a specification, we attempt to factor out basic principles. These principles should be simple and intuitive to a degree which a concrete elaboration cannot attain.

Informal style

For the sake of readability, we do not embark on any formal definition. Rather, we want to convey the definition in a natural style which concentrates on ideas and intent at the expense of formal exactness and completeness.

Goals

XDML is a set of rules how XDM values can be designed in order to become more useful entities as compared to ordinary XDM values. The key idea is to insert into the XDM values control information which guides the interpretation and processing of the data. An XDM value thus augmented is called an XDML value. Its usefulness is provided by an XDML processor, which is a generic program evaluating the control information. XDML addresses the following major goals:

- to structure XDM values into nestable parts
- to enable name-based access to XDM parts
- to associate XDM parts with metadata
- to process XDM parts as guided by their metadata

Structure model

XDML structures XDM values by *grouping* the XDM items. The resulting groups are units of usage in a broad sense: conceptual units of information, units of data retrieval and units of data processing. Item groups are called *information units*. The grouping approach distinguishes:

- simple information units – do not contain other units
- complex information units – contain other units

and introduces the following constraints:

- complex units do not contain data items which are not contained by nested units

- the information content of a complex unit is regarded as unordered collection of units

Metadata model

Information units can be associated with metadata. XDML uses a simple metadata model which

- distinguishes between descriptive data and control data
- distributes control data into distinct sets, called metadata components
- associates each metadata component with a component of data processing
- defines how metadata components control the processing components

Processor model

XDML values are submitted to an XDML processor, a generic program which evaluates the control information and is responsible for reporting and processing the data accordingly. The processor is viewed as a set of components, each one of which is responsible for a particular kind of processing. The general concept of XDML distinguishes

- an XDML parser
- further XDML processing components

An XDML parser delivers the information encoded as XDML value in a structured way. Other processing components enable other kinds of processing. A concrete specification of XDML must identify those components, describe their functionality and define the control which the matching metadata component exerts.

Encoding principles

XDML defines the syntax and semantics of control information embedded in XDM values. We propose four general encoding principles:

- control information is encoded by control items, to be distinguished from data items
- a control item is an XDM item which is an element information item in a particular namespace
- metadata of a unit are written into the control item marking the start of the unit
- metadata components are not mixed - each component is encoded by a distinct (possibly empty) set of elements

XDML - concrete proposal

The step from XDML as a concept to a concrete specification requires:

- Elaboration of the syntax of control items
- Elaboration of the functionality of an XDML parser
- Definition of further XDML processor components and their control by metadata components

On language binding

The XDML user communicates with the XDML processor via an API. A processor implementation is therefore bound to a programming language, whereas the concept of an XDML

processor is language neutral. Our ongoing implementation work uses Java, and API code snippets in this paper use Java as well. This representation is chosen for convenience sake and does not mandate Java in preference to other languages.

Concerning the syntax it suffices to recapitulate the rules applied in our illustrative examples:

- Control items contain elements in the XDML namespace: http://www.xdml.org/ns
- Simple information units are preceded by an <xm:part> item
- Complex information units are delimited by <xm:complexPart> and <xm:complexPartEnd> items
- Name and partID of an information unit are given by the "name" and "partID" attribute of an <xm:part> or <xm:complexPart> item
- Descriptive metadata are encoded as attributes or child elements of an <xm:part> or <xm:complexPart> item; they must be in a namespace but must not be in the XDML namespace

The remainder of this section presents a proposal for the functionality of the XDML processor. Apart from describing the behavior of the XDML parser, this means defining further processor components as well as the metadata components which control them.

Components of metadata

We distinguish the following metadata components.

descriptive metadata

These are any metadata which are not defined to control data processing in a deterministic way. *Encoding:* Attributes and child elements of <xm:part> and <xm:complexPart> items which are in a namespace different from the XDML namespace.

evaluation metadata

They specify how the *current value* of the information unit must be processed in order to obtain the *final value* of the unit. This component relies on a processing model built into XDML which distinguishes current and final values of an information unit and prescribes how a current value is replaced by the final value. *Encoding:* <xm:evaluate> child elements of an <xm:part> item.

action metadata

They specify actions which should (or might) be performed with the value of the unit. *Encoding:* <xm:action> child elements of an <xm:part> or <xm:complexPart> item.

translation metadata

They control how the value of the information unit is translated into specific non-XDM data models, for example into the data model of a particular programming language. *Encoding:* <xm:translate> child elements of an <xm:part> item.

Processor components

The XDML processor is a set of several components, dedicated to particular kinds of processing XDML values.

XDML parser

It delivers the contents of information units either randomly accessed or visited during iteration over the XDML value.

XDML evaluator

It "finalizes" an XDML value by replacing preliminary values in need of an evaluation by final values resulting from the evaluation. Controlled by *evaluation metadata* (<xm:evaluate>).

XDML executor

It performs particular actions which are related to or use the value of an information unit. Controlled by *action metadata* (<xm:action>).

XDML translator

It translates the values of information units into values of non-XDM data models, namely the values and objects of general purpose languages. Controlled by *translation metadata* (<xm:translate>).

XDML transformer

It transforms an XDML value into another XDML value. This component is not controlled by a particular metadata component; rather, it responds to the sum total of the information contained by an XDML value.

XDML parser

The XDML parser is the core functionality of an XDML processor upon which any other functionality is built. Its task is to make the information content of XDML values accessible to applications and to other processor components. Specifically, it must be able to *locate* information units and to *report* their contents. The XDML value may be consumed as in-memory representation or as serialized XDM. The parser locates units either by random access or by iterating over the information units of an XDML value.

On language binding

> The parser reports data via an API. A parser implementation is therefore bound to a programming language, whereas the concept of an XDML processor is language neutral. Our ongoing implementation work uses Java, and API code snippets in this paper use Java as well. This representation is chosen for convenience sake and does not mandate Java in preference to other languages.

Locating information units

Random access is controlled by unit name or a path of unit names (addressing nested units), or by partID. Iteration over the units of an XDML value can be with or without filter. A filter can be positive ("only these") or negative ("all but these"), deep ("everything under this") or flat ("only if a simple information unit") and is based like random access on unit name or partID. The name may be specified literally or using regular expressions for namespace and local name. *Note*: the filtering facility of the XDML parser is restricted to evaluate the names and partIDs of units; a general filter facility is provided by the XDML transformer described below (the section called “XDML transformer”).

Reporting information units

To "report" an information unit means to offer structured access to its information content. This means distinct delivery of these properties:

- name: QName
- part ID: NCName (optional)
- value: either an XDM value or a sequence of information units
- a group of markup components

- descriptive markup component (optional)
- evaluation markup component (optional)
- action markup component (optional)
- translation markup component (optional)

A markup component is an ordered collection of *component items*; a component item is an unordered collection of *named properties*, where the name is a QName and the value is one of these:

- single string
- sequence of strings
- single element information item
- sequence of element information items

```
Example: XDML representation of an information unit with four markup components

- descriptive markup: name and value of the e:createdAt attribute
- evaluation markup: one component item represented by the <xm:evaluate> elemen
- action markup: three component items represented by the <xm:action> elements
- translate markup: two component items represented by the <xm:translate> eleme

<xm:part name="xbrlReport"
         xmlns:e="http://example.com"
         xmlns:xm="http://www.xdml.org/ns"
         e:createdAt="2010-12-31T09:51:20" >
  <xm:evaluate type="xproc">…</xm:evaluate>
  <xm:action type="store">…</xm:action>
  <xm:action type="httpSend">…</xm:action>
  <xm:action type="ftpSend">…</xm:action>
  <xm:translate target="java" …>…</xm:translate>
  <xm:translate target="cpp" …>…</xm:translate>
</xm:part>,
<rawData>…</rawData>,
<otherRawData>…</otherRawData>
```

Reporting unit values

If an <xm:translate> element is found whose target is the target language of the reporting, then the XDM value is *translated* into a language value as prescribed by the <xm:translate> element; otherwise an appropriate standard mapping is used. In the case of Java it is the type mapping defined by JSR225 (XQJ Spec).

```
Example: information unit whose translation markup controls the Java representa

The value is delivered as Map<String,String> object. In the absence of the
<xm:translate> element, the XDML parser would deliver the value as DOM node.

<xm:part name="abcCodeDictionary" xmlns:xm="http://www.xdml.org/ns">
  <xm:translate target="java"
                type="xqjplus"
                type="map_string_to_string_object"
```

```
                entryPath="entry"
                keyPath="@code"
                valuePath="text()"/>
</xm:part>,
<dictionary>
  <entry code="a001">foo</entry>
  <entry code="a002">bar</entry>
</dictionary>
```

Reporting metadata

The structured delivery of metadata reflects their component model. A markup component is delivered as a sequence of objects which represent the component items and provide access to their properties:

```
class InformationUnit {
   …
   DescriptiveMarkup[] getDescriptiveMarkup();
   EvaluationMarkup[] getEvaluationMarkup();
   ActionMarkup[] getActionMarkup();
   TranslationMarkup[] getTranslationMarkup();
}
```

The component classes extend a class offering generic access to a metadata component:

```
class MarkupComponent {
   …
   QName[] getPropertyNames();
   int getPropertyType(); // PRTY_STRING, PRTY_STRINGS, PRTY_ELEMENT, PRTY_ELEM
   String getStringProperty(QName name);
   String[] getStringsProperty(QName name);
   Element getElementProperty(QName name);
   Element[] getElementsProperty(QName name);
}
```

Note that proprietary meta data are represented by DescriptiveMarkup components. They are accessed in the same way as standard meta data (getXXXProperty methods). See Appendix B, *The proposed Java API used by the XDML parser for reporting information units (excerpt)* for more details about the reporting API of an XDML parser.

XDML evaluator

An XDML evaluator is an infrastructure component processing "evaluation markup". The concept of an XDML evaluator is based on a processing model which allows XDML values to contain intermediary data packaged together with a specification how to turn them into final data. A major goal of this feature is to increase the potential of XQuery, enabling it to leverage other technologies and command them to perform a specified postprocessing.

Formally, XDML distinguishes the *current value* of an information unit from its *final value*. If the unit does not contain evaluation markup, both values are identical; otherwise the final value will be obtained by executing the evaluations prescribed by the evaluation markup. In this case, the current value is the XDM value contained by the unit. (Note that only simple information units may use evaluation markup.)

Every item of the evaluation component – that is, every <xm:evaluate> child of <xm:part> - specifies an evaluation resulting in an XDM value which …

- will be the final value of the unit, if the <xm:evaluate> child is the only or last one

- will be the new current value of the unit when proceeding to evaluate the next <xm:evaluate> child

In other words: the <xm:evaluate> children are evaluated in order, where the preceding child's output is the current child's input. The current draft constrains evaluations to be the execution of an XProc pipeline which is either provided inline (as child of <xm:evaluate>), or read from another information unit ("partRef" attribute stating a partID) or referenced by document URI ("href" attribute). The evaluation result is defined to be the primary output of the pipeline. >

The semantics of XProc are slightly modified in order to …

- enable the pipeline to refer to the information unit's current value
- enable the pipeline to refer to other information units' final values

Reference to the current value

Input ports may be connected to the information unit's current value by using a reserved step name (__THIS) and an arbitrary port name:

```
<p:pipe step="__THIS" port="dummy"/>
```

References to other information units

In order to allow connections to other information units' (final) values, these values must be explicitly imported by an extension attribute of the <p:pipeline> (or <p:declare-step>) element ("xm:import-information-units" attribute) which contains a list of partID values. When imported, these partIDs can be used as steps names:

```
<p:pipeline xm:import-information-units="foo bar" …>
   …
   <p:pipe step="foo" port="dummy"/>
   …
   <p:pipe step="bar" port="dummy"/>
   …
</p:pipeline>
```

Note that it is always the final value of an information unit what is imported.

Pipeline input

The pipeline element is a child of an <xm:evaluate> element. The pipeline input is provided by *sibling* elements which use exactly the same syntax as is used within pipelines, that is: <p:input> and <p:with-option> elements. See Appendix C, *Example demonstrating the use of evaluation markup* for a complete example demonstrating the use of evaluation markup

XDML executor

An XDML executor is a processor component which "does something" with the values found in an XDML value. It performs some action which takes the value of an information unit as input. This should be distinguished from the behavior of an XDML evaluator, which also uses the unit value as input, but with the intent of replacing the current value by a final value henceforth to be used in place of the current value. (In some cases the distinction may become hazy.)

Some examples of actions which might be performed by an XDML executor are:

- store a value on disc or in a database
- send a value per http, ftp or smtp

- interpret the value (a string) as code to be executed (e.g. a Perl script)
- interpret the value (string sequence) as series of SQL statements to be executed

Each action type will have a specific set of parameters required. It is not clear which actions should be regarded as "standard" to be supported by a standard XDML executor, which anyway is intended to be extended by proprietary actions. The ease of implementing extensions is a major motivation for defining this component in the first place. Extension is simple, due to

- convenient access to data and metadata, offered by the XDML parser
- automatic invocation of the extensions, based on a registration facility

See Appendix D, *Example demonstrating proprietary extensions of the XDML executor* for an illustrative example of how the XDML infrastructure is extended by a proprietary XDML executor.

XDML translator

An XDML translator is a component which "translates" XDM values into values of another data model in a non-standard way. An example has been presented in the section on the XDML parser. The motivation for this component is improved integration of XML technology – namely XQuery - into general purpose programming languages like Java.

Such integration can be accomplished by a special style of writing XQuery scripts: designing them to deliver data in the format really desired by the invoking program, rather than forcing the program to evaluate standard representations of XDM values. This style regards the XDM values as a means to an end which is not XDM itself. Instead of pushing XDM values into the general purpose environment, the desired formats (e.g. a map format) pull XDM representations (e.g. a node or a sequence of strings) from the XQuery component which enable the XDML processor to manufacture the required values, guided by metadata. The machinery and the metadata are supplied by the XDML translator and translation markup, respectively. A first implementation of a preliminary version of XDML contained an elaborate XDML translator for a Java environment (Rennau 2010), supporting the translation into many map and collection types, Properties objects and even custom type objects.

XDML transformer

This processor component relates to the sum total of information found in an XDML value. The XDML transformer can create

- a *filtered* copy of an XDML value
- a *transformed* copy in which information units have been transformed
- an *extended* copy in which units have been added (appended, prepended, inserted) or extended by adding items to their values
- a copy resulting from a *combination* of filtering, transforming and extending

The basic operations are filtering, transforming and extending. Interestingly, the differences between XDML values and XML documents enable extended control of those basic operations when dealing with XDML values. Compared with an XML document, an XDML value is distinguished by two features:

- explicit separation of metadata from primary information
- units of primary information may consist of several independent entities (that is, XDM items)

We now discuss the consequences for filtering, transforming and extending.

filtering XDML values

The canonical way of filtering XDM data is to evaluate expressions – XPath or XQuery – and use the Effective Boolean Value of the result as filter criterion. Handling simple information units, we can refine and simplify the control by deciding the context in which the expressions are evaluated:

1. metadata (<xm:part>)

2. value items (iteratively)

3. a virtual aggregation of the *value items* (a document whose document element is an <xm:informationUnitValue> element whose children are the value items)

4. a virtual aggregation of the *whole* information unit (a document whose document element is an <xm:informationUnit> element whose first child is the <xm:part> item of the original value, and whose further children are the value items

Note: "children are the value items" is shorthand for "the child nodes are constructed from the value items according to the XQuery rules for element constructors".

In case (2) we must further distinguish two alternatives: requiring at least one item to satisfy the condition, or requiring every item to satisfy the condition (corresponding to XQuery's "some" and "every" expressions).

transforming XDML values

Any tool consuming and producing XDM values can be used to transform a simple information unit, especially: XPath, XQuery, XSLT, XProc. As was the case with filtering, we can refine and simplify the control by choosing one of the four contexts available when filtering (see above). Here, the choice determines both, the evaluation context and what is to be replaced by the evaluation result. Interesting, transformations targeting metadata may be used to associate units with actions. Note that in case (4) ("context is the unit as a whole") the result of the transformation must be a sequence whose first item is an <xm:part> element. Note also that in case (2) ("context is the value items, iteratively") there is no distinction between modes corresponding to "some" or "every" expressions.

extending XDML values

The design of XDM has taken care to make XDM values resilient to updates: add or remove any number of items at any place within the XDM value – you always end up with a valid XDM value. This resilience is reduced when moving over from XDM to XDML, especially if complex information units are used. For example, the removal of a leading <xm:part> item or of any <xm:complexPartEnd> item produce invalid XDML. However, compared with an XML document the ease of adding or removing information is drastically increased.

1. Complete information *units* can always safely be appended, prepended or inserted between two existing units

2. Complete information *units* can always safely be removed.

3. If the currently last unit is a simple unit, one can always *append items* to the XDM value which will implicitly extend the value of that last information unit.

4. Any simple unit can always be extended by *inserting items* before the first following item which is an <xm:part>, <xm:complexPart> or <xm:complexPartEnd> item.

combining filter, transformation, extension

An important possibility is the combination of filtering, transforming and extending. Combining filtering and transforming, the transformation is applied only to those information units which the filter selects. Combining filtering and extending can be done in two ways: using the filter in order to determine before or after which units new units are to be inserted; and using the filter in order to determine which simple units are to be extended by new items.

usage pattern: applying actions to selected units by combining filtering and transformation

The special combination "filter units – transform metadata" enables an elegant way of "injecting" actions into selected units: the filter identifies the units, and the transformation adds the appropriate <xm:action> elements to the metadata. All that remains to be done is to submit the result to the XDML executor.

XDML processor - conformance

We regard the XDML parser as mandatory and all other components – evaluator, executor, translator, transformer - as optional. An implementation of any of these components MUST support a component-specific set of *processing types*, whose definition is in progress. The markup represents processing types by a "type" attribute of <xm:evaluate>, <xm:action> and <xm:translate>. For example, we propose that an evaluator MUST support the processing type "XProc evaluation", recognized by the "type" attribute of <xm:evaluate>:

```
<xm:evaluate type=”xproc”>…</xm:evaluate>
```

In addition to this, a conformant component must be extensible in a standard way. It MUST implement a standard interface for the registration of proprietary handlers in charge of a non-standard processing type:

```
interface ExtensibleComponent {
   void registerExtension(XDMLComponent extension, QName[] processingTy
}
```

Extensions are triggered by markup stating the registered processing type, like this:

```
<xm:evaluate type=”e:foo” …>…</xm:evaluate>
```

The implementation must guarantee that if the component is invoked for a given information unit, the occurrence of such markup triggers the invocation of the registered "foo" handler in the same way as standard markup triggers the standard processing. This registration feature amounts to a generic "wiring" of proprietary processing types to proprietary handlers. For an illustrative example of the extension mechanism see Appendix D, *Example demonstrating proprietary extensions of the XDML executor*. Note that this extensibility lives in the context of a standardized component - e.g. "XDML executor" or "XDML evaluator". The component uses the values received from proprietary handlers in the component-specific way. The XDML evaluator, for example, uses the values as a replacement of the current value of the unit, whereas the XDML translator uses the values as non-standard representations of the unit value.

Implementation

A prototypic implementation of an XDML processor is a work in progress. It refactors and extends XQJPLUS (Rennau 2010), a Java API designed to enable XQuery programs which deliver to a Java client an "infotray". This is a collection of named objects which have been constructed from XDM subsequences. Object construction is guided by information read from control items which precede the data items within the query result. Translated into the new terminology: an infotray is an object representation of an XDML value, and the processing which created it can be assessed as the work of an XDML parser which is backed by an XDML translator.

XQJPLUS is an incomplete implementation of an XDML processor consisting of an XDML parser and an XDML translator. It is incomplete in terms of functionality, and non-conformant as the encoding of metadata is not yet organized as prescribed by the present work. This non-conformance is not due to serious difficulties. Rather, it reflects the fact that the present work is a *generalization* of the approach taken by XQJPLUS. There, a basic principle has been explored: control items may partition XDM

values into named parts and associate them with metadata which can control the processing of those units. XQJPLUS has discovered this principle as a means to a rather particular end: enable XQuery to deliver complex results in a really useful and convenient form - as named units which correspond well to the entities actually needed, rather than echoing the XDM data model and its standard translation into the Java type system. The present work, on the other hand, identifies the functionality of XQJPLUS as the task of a *particular* component, which is one of a set of components all based on the principle of control items which partition XDM sequences and associate subsequences with metadata.

Discussion

The concept of XDML can be devided into two somewhat independent parts. One is the principle of using control items in order to structure XDM values and associate the emerging units with metadata. The other part is the idea to package primary data together with control data which configure a predefined processing, implemented by a generic infrastructure and triggered by the receptor of the data. One may even argue that these two principles are *completely* independent. This is true in so far as one could apply the approach to XML documents just as well. One may define control data to be inserted into XML documents - rather than XDM values - and to be evaluated by an infrastructure processing single XML documents, rather than XDM values. But we think that this principle of "data + control data" seems more attractive in the case of XDM values, which are collections, rather than a single tree. In an XDM value data and control information can be strictly separated, as they reside in different XDM items. In the case of XML, the insertion of control data among primary data may easily disturb. To begin with, they will almost always imply a violation of the data schema; and they tend to create confusion, complicate queries and cause errors of various kinds. Seen from this point of view it is really the use of XDM what invites to this approach - and the other way around, it is an approach which has the potential to make XDM more attractive.

A. Formal definitions: control items, information units, XDML

We define a few concepts formally.

control items

[control items] are those items of an XDM value which have a name in the namespace “http://www.xdml.org/ns”. Their local name must be a member of a set of names which currently comprises: part, complexPart, complexPartEnd.

information unit

An **[information unit]** is the information content of a subsequence of XDM items which begins with an <xm:part> or with an <xm:complexPart> control item and ends …

- if the begin is an <xm:part> item: before the first following <xm:part>, <xm:complexPart> or <xm:complexPartEnd> item, if any occurs, or at the end of the XDM value, otherwise
- if the begin is an xm:complexPart item: with the matching <xm:complexPartEnd> item (matching understood in the sense of “proper nesting”)

simple and complex information unit

A **[simple information unit]** is a unit beginning with an <xm:part> item; a **[complex information unit]** is a unit beginning with an <xm:complexPart> item.

value of an information unit

The **[value]** of an information unit is …

- if the unit is simple: the sequence of non-control items contained by the unit
- if the unit is complex: the sequence of (top-level) information units contained by the unit

Note that complex units may contain simple and complex units, so that units can be nested to any depth. Also note that the value of a simple information unit is an XDM value, as any sequence of XDM items is an XDM value.

XDML value

An **[XDML value]** is an XDM value which contains control items in a way implying conformance to the following rules:

- every non-control item belongs to the value of a simple information unit
- the names and contents of attributes and children of control items satisfy a set of rules not yet defined in a comprehensive way but including these rules:
  - <xm:part> and <xm:complexPart> items have a mandatory "name" attribute whose value is a valid QName
  - <xm:part> and <xm:complexPart> items have an optional "partID" attribute whose value is an NCName and is unique among all partID values found in the XDM value

Note that partID values enable cross references between units as well as external references to individual units.

XDML

[XDML] denotes the comprehensive set of rules which an XDM value must satisfy in order to be an XDML value. XDML stands for "XDM markup language".

B. The proposed Java API used by the XDML parser for reporting information units (excerpt)

```
class InformationUnit {

   // *** section: name and ID ***
   QName getName();
   String getPartID();
   QName[] getNamePath();

   // *** section: report value ***
   // provide a generic representation (based on XQJ)
   XQSequence getXdmValue();

   // dedicated methods for homogeneous units
   Node getNodeValue();
   Node[] getNodesValue()
   String getStringValue();
   String[] getStringsValue();
   int getIntegerValue();
   int[] getIntegersValue();
   …
```

```
   // *** section: report meta data ***
   DescriptiveMarkup[] getDescriptiveMarkup();
   EvaluationMarkup[] getEvaluationMarkup();
   ActionMarkup[] getActionMarkup();
   TranslationMarkup[] getTranslationMarkup();
}

class MarkupComponent
   int getComponentType(); //MCTY_DESCRIPTIVE | MCTY_EVALUATION | MCTY_ACTION |
   String getProcessingType();
   QName[] getPropertyNames();
   int countProperties();
   int getPropertyType(); // PRTY_STRING, PRTY_STRINGS, PRTY_ELEMENT, PRTY_ELEM
   String getStringProperty(QName name);
   String[] getStringsProperty(QName name);
   Element getElementProperty(QName name);
   Element[] getElementsProperty(QName name);
}

class EvaluationMarkup extends MarkupComponent {
   String getEvaluationType();
   Element getPipeline();
   String[] getImportedUnits(); // list of part IDs
   …
}

class ActionMarkup extends MarkupComponent {…};
class TranslationMarkup extends MarkupComponent {…};
class SemanticMarkup extends MarkupComponent {…};
```

C. Example demonstrating the use of evaluation markup

```
Example: XDML value containing evaluation markup.

<xm:part name="foo2zoo" partID="xslt02"/>,
<xsl:transform ...>...</xsl:transform>,

<xm:part name="xyz" partID="bar"/>,
<bar>...</bar>,

<xm:part name="xbrlReport" xmlns:xm=”http://www.xdml.org/ns”>
   <xm:evaluate xmlns:p="http://www.w3.org/ns/xproc">

      <!-- pipeline input -->
      <p:input port="source">
         <p:pipe step="__THIS" port="dummy"/>
      </p:input>
      <p:with-option name=”format” select=”’xhtml’”/>

      <!-- the pipeline -->
      <p:pipeline xm:import-information-units="xslt02 bar"
                  version="1.0">
```

```
         <p:input port="stylesheet">
            <p:pipe step="xslt02" port="dummy"/>
         </p:input>
         <p:input port="source">
            <p:pipe step="bar" port="dummy"/>
         </p:input>
        ...
      </p:pipeline>
   </xm:evaluate>
</xm:part>,
<rawData>...</rawData>
```

D. Example demonstrating proprietary extensions of the XDML executor

Consider the case that in some environment the proprietary actions "a1" and "a2" are useful, and that they are accomplished by passing an XML document to a library function (doA1, doA2), in one case along with some other data. To incorporate this functionality into the XDML infrastructure, one would

1. define the proper use of an <xm:action> element for controlling these actions (either by XML schema or informally).

2. write an extension of XDMLexecutor (say, class MyExecutor)

MyExecutor might look like this:

```
class MyXDMLexecutor extends XDMLexecutor {
   exec(ActionMarkup currentAction, InformationUnit iu) {
      String type = currentAction.getActionType();
      if (type.equals("a1") execA1(currentAction, iu);
      else if (type.equals("a2") execA2(currentAction, iu);
   }
   execA1(ActionMarkup currentAction, InformationUnit iu) {
      mylib.doA1(iu.getDocumentValue());
   }
   execA2(ActionMarkup currentAction, InformationUnit iu) {
      String foo = currentAction.getStringProperty("foo");
      Node[] bar = currentAction.getNodesProperty("bar");
      mylib.doA2(iu.getDocumentValue(), foo, bar);
   }
}
```

At runtime it will suffice to instantiate and register the proprietary XDML executor:

```
QNames[] actionTypes = new QName[] {new QName("a1"), new QName("a2")};
XDMLexecutor.registerExtension(new MyXDMLexecutor(), actionTypes);
```

From now on, any action markup specifying as action type "a1" or "a2", for example

```
<xm:part=″partX″>
   <xm:action type=″a1″/>
</xm:part>,
<xdata>…</xdata>,
```

```
<xm:part name="partY">
   <xm:action type="a2">
      <foo>hello</foo>
      <bar><a>…</a></bar>
   </xm:action>
</xm:part>,
<ydata>…</ydata>
```

will trigger the automatic invocation of the registered executor. (Provided, of course, that the processing of action markup is triggered as a whole.)

Bibliography

[Rennau 2010] Hans-Juergen Rennau. Java Integration of XQuery - an Information-Unit Oriented Approach. Presented at Balisage: The Markup Conference 2010, Montréal, Canada, August 3 - 6, 2010. In Proceedings of Balisage: The Markup Conference 2010. Balisage Series on Markup Technologies, vol. 5 (2010). doi:10.4242/BalisageVol5.Rennau01. http://www.w3.org/TR/xpath-datamodel/.

[XQJ Spec] Jim Melton et al, eds. JSR 225: XQuery API for JavaTM (XQJ) 1.0 Specfication. http://jcp.org/en/jsr/detail?id=225.

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