Finalising a (small) Standard

John Lumley
jwL Research, Saxonica
<john@jwlresearch.com>

Abstract

This paper discusses issues and lessons that arose during the finalisation of a standard (library) for XSLT/XPath/XQuery extension functions to manipulate binary data. This process took place during 2013 in the EXPath community, through shared (mailing-list) commenting, specification redrafting, implementation experimentation and test suite development. The purpose, form and specification of the library (which isn't technically difficult) are described briefly. Lessons and suggestions arising from the development are presented in four broad categories: establishing policies, concurrent implementation and application, using tools and declarative approaches, and pragmatic issues. None of these lessons are new, but bear reinforcement. This work was performed under the auspices of the EXPath community and was funded by Saxonica Ltd.

1. Standards – let's have plenty

Since its inception in the mid 90s, the world of XML has been governed by standards. Originally attempting to regularise the extension of web pages, XML was developed as a meta-syntax for markup, aimed at using a strict tree-based representation of propertied element nodes containing sub-trees or text nodes. Very soon work started on developing XSL as a formatted and paginated alternative to HTML for documents with professional appearance. As we're all well aware, the two aspects of XSL, formatting and variable document generation, split into two orthogonal standards - XSL-FO and XSLT. The latter developed, with XPath (and associated XQuery), into a full declarative XML-transformational language. In most recent versions XSLT/XPath/XQuery have become full functional programming languages, with XML trees as central data type.

Significant work under the auspices of W3C has developed and finalised these standards for XSLT/XPath/XQuery[1] through three major versions over the past 15 years. In each case a very comprehensive specification has been developed, reviewed, criticised and modified in cycles that are typically 3 years long and involve a few

[1]Whilst there are differences, for the rest of this paper, unless stated otherwise, the term XSLT or XPath is used to refer to any of the three.

dozen contributors. Examples, test cases and experimental/operational implementations are all used to develop and finalise the specifications, which is often followed by a further 3 years of polishing.

Once a standard (version) has been finalised, a degree of stability should then encourage developers of both implementations and applications to build and support software, without the language's syntax or semantics altering. Of course it is exactly such full-scale use of a language that could expose shortcomings or new features that are needed to increase utility. The art of developing standards is to anticipate as much as is needed to get a useful and robust set of features that i) is adequate for significant application use but ii) not too complex for implementation or application.

In the case of XSLT, the first version (1.0) was developed very quickly, concentrated on defining a model for transforming XML through pull-based selection (using XPath to select from sub-trees of the input document) or push-based case generation, using pattern-matching templates. A minimal necessary set of functions and instructions was also defined (e.g. `count()`, `translate()`, `xsl:number`) to support necessary computation.

Whilst the functionality of XSLT 1.0 was sufficient for many initial application purposes, additional requirements appeared slowly. Some of these could be satisfied by interesting but somewhat complex coding techniques (such as 'Muenchian grouping'); others would require an extension to the language itself. In the case of XSLT it had been anticipated that extension functions or even extension instructions could be added to an implementation, to provide additional functionality. These could be very application specific (`my:jpg.size($uri)`) or somewhat more general (`math:cosec()`) and implementations were encouraged to provide support mechanisms for such extensions.

Over time a few common libraries of such (XPath) extension functions (but not XSLT instructions) were developed, by mechanisms described later, in topics such as mathematical functions, or even something as language-central as being able to reuse generated sub-trees in XSLT. Gradually such libraries increased the firmness of their specification and influenced some of the additional requirements for subsequent versions of the main standard. A specific example is the incorporation into XPath3.0 of mathematical (`math:pow()`...) and transcendental (`math:sin()`...) functions, which had been developed originally by the EXSLT group[4]) as a library for XSLT1.0 operating on values of type `xs:double`.

1.1. Community spirit

Large standards usually grow under rigorous and well-controlled frameworks, such as W3C, but these additions often arise from some small group of enthusiasts identifying common ground and interest and collaborating on an informal basis. These developments are usually a 'community effort' with a group of (world-dis-

tributed) volunteers who propose, define, refine, criticise and revise some 'specification' document, whilst discussing and developing implementations and tests. The tests are usually built as large sets of small test cases, and become a key part of proving the specification, especially when multiple implementations are checked against them. But whilst there are varying degrees of formality in the process of development of such specifications, all developments have some aspect of being a social process, subject to personalities, biasses and individual interests.

These efforts sometimes succeed, sometimes they peter out, sometimes they stall, or the effort is abandoned and a new direction chosen. The degree of formality of such a standards effort can be variable – from simple discussions and documents, right up to near-W3C levels of rigour. As mentioned previously, the EXSLT group was active and influential in the period 2001-6 exploring features like mathematical functions, dates and times and regular expressions. Several of these made their way into the larger standards in subsequent years, though many still exist in a form of limbo – half-finished, partially supported by one or two implementations, and used in very few applications.

The EXPath community[1] is such a group, attempting to develop suites of rigorous extensions for the XPath/XQuery/XSLT world. It started in 2010 and has perhaps a dozen or so active contributors. Early efforts include specifications for file, http client, packaging and ZIP manipulation. The intention was to create such suites to standards of rigour approaching those demanded by W3C for its full specifications, whilst having a more agile platform for developing new functionality. In particular the specification documents it publishes use the same format and organisation as those of W3C[2]. However, until very recently none of the specifications had stabilised enough to warrant being labelled 'Version 1.0', despite one of them (File[3]) having been used extensively by a number of developers over the past few years.

2. Fiddling with bits – Binary Module

In the summer of 2013 the author was invited by Saxonica to work on EXPath's Binary module[2], intended to support binary and bit-level manipulation of data within XSLT/XPath/XQuery[3]. An initial draft of the module specification had been created by Jirka Kosek (University of Economics, Prague) in the spring of 2013. Over the next four months, the specification was revised several times, discussion on features and criticisms were fielded through a mailing list, test suites were developed

[2]It operates under the W3C Community Final Specification Agreement [http://www.w3.org/community/about/agreements/final/].

[3]The W3C Working Group developing XSLT had agreed such features would be useful, but were out of scope for XSLT3.0

and tested. Eventually a specification (whose rigour and structure is based on those of W3C) was finalised in early December as 'Version 1.0'.

The module eventually ended up with a library of 26 functions, in four broad classes – generating binary constants, basic operations of breaking, joining, extending and searching, encoding and decoding text and numeric values and finally standard bitwise operations.

A simple use case was finding the size of a JPEG image. Many independent instances of such an extension function will have already been written in Java, but with the facilities from the Binary module this could be written directly in XSLT:

```
<xsl:variable name="binary"
  select="file:read-binary(@href)" as="xs:base64Binary"/>
<xsl:variable name="location"
  select="bin:find($binary,0,bin:hex('FFC0'))"/>
<size width="{bin:unpack-unsigned-integer($binary,
                $location+5,2,'most-significant-first')}"
      height="{bin:unpack-unsigned-integer($binary,
                $location+7,2,'most-significant-first')}"/>

      => <size width="377" height="327"/>
```

A JPEG image consists of a series of *segments*, starting with a marker consisting of `0xFF` followed by a single byte type identifier, which is never `0x00`. (`0xFF` in data is byte-stuffed with a trailing `0x00`, so two-byte sequences starting with `0xFF` followed by non-null always indicate start of segment.) Identifier `0xC0` denotes a *Start Of Frame* segment and contains the size, number of (colour) components and subsampling type of the image, all with defined byte lengths.

We need the binary of the JPEG image, which in this case we've read from a file using `file:read-binary()` from the EXPath File module, but could have been web-uploaded as base64 data. Our method is to i) find the offset location of the Start Of Frame segment and then ii) decode the byte-positioned values for the width and height at that location. This needs just two functions: `bin:find()`, which searches for the first occurrence of a contiguous byte sequence inside another and `bin:unpack-unsigned-integer()`, which returns an integer of specified length from a range of bytes in the input.

Even the most evangelistic wouldn't class this module as requiring 'rocket science' skills to implement in an XPath processor – the Saxon implementation is a single Java class with about 1000 (sparse) source lines[4]. The interest is not what this module is really about, but how its specification is developed to consensus, what lessons might be learned from the experience and what appear to be effective decision-making processes.

[4]Saxon already contained a class to represent items of `xs:base64Binary` type, with data storage and (de-)serialisation machinery.

Whilst the author had been working in software research for many many years, and been a very heavy user of XSLT (including building large extension functions[5]) in the field of document engineering for much of the last decade, he had not been involved in the W3C or related standards activities. Being asked to work not just on the implementation (inside a very well established software product, *viz* Saxon), but also helping drive the specification itself to a 'standard' position would undoubtedly be educational. And so it proved.

The initial work covered three main areas:

- Studying the current specification and producing a modified draft in the light of previous discussions, further comments from below and evident points of consensus. This led to the publication of a revised draft at the end of July 2013.
- Instigating further discussion in the EXPath community to push the specification forward. This involved summarising what appeared to be the main issues that had been discussed beforehand and those that were apparent from my reading. For example it was uncertain which of two different binary types, `xs:hexBinary` or `xs:base64Binary` would be supported or both[6]. These discussions were carried out entirely through the group's mailing list.
- Building a skeletal Java extension class to use with Saxon (including gaining familiarity with the company's build environment), creating a few sample test applications, getting it all running and generating early test-sets and exercising and testing them.

This was the bulk of the continuous work on the project, taking 10 days of effort spread over a calendar month. However it was not the end of the affair - two more intermediate drafts were published over the next three months, until convergence to a version 1.0 recommendation followed at the beginning of December 2013. During this process of refinement, which of course took place in bursts of frantic activity and oh-so-silent lulls and involved ~100 mailing-list postings as well as private correspondence, there were several issues to be resolved. Four of these were significant:

- What was the main binary type to be used?
- 'Endianness' for numeric (de-)coding, including what the default should be.
- Overhauling the error code naming.
- Issues arising from access to the 'end' of data, and behaviour when arguments are empty sequences.

[5] SVG-PDF converters, text-block paragraph wrappers that preserve tree isomorphism, constraint-based layout resolvers – that sort of thing.

[6] `xs:base64Binary` was chosen – it is more efficient in serialisation, can be cast to and from `xs:hexBinary` cheaply, as both are usually implemented as wrapper classes around the primary byte sequence data, and was the *de facto* binary type used in the File module.

The following sections discuss lessons from that work, in four broad areas – concurrent implementation and application, establishing policies, using tools and declarative approaches, and pragmatic issues.

3. Prove the specification – do it all together

A specification is a document intended to be read and understood by a human. But it also needs to be unambiguous, with little doubt in what it is defining, since later on applications will rely on processors behaving very closely to that specification indeed. And to check such lack of ambiguity, a specification needs to be *proven* – tested that it describes what was intended, and not have nasty surprises lurking for the unwary, be they either those implementing processors that meet the specification ("That operation is $O(n^4)$"), or those writing applications ("There's no way to check that condition without triggering an error.").

A specification that was purely mathematical might of course be susceptible to being proven mathematically, but the practical and readable specifications we deal with are not quite that rigorous. So how do we carry out such proof while the specification is being developed? Unsurprisingly I suggest techniques that these days might be referred to as *agile*: i) use the computer early and often, ii) find some medium-sized examples, both to illustrate and test and iii) build specification, implementation, examples and tests concurrently.

All the parts of the development – discussion, specification, implementation, tests and applications – are related and whilst there are consequential dependencies, some of these are circular and refining. Thus there are distinct advantages if you can close the loops quickly, which requires progress on all almost fronts simultaneously.

3.1. Don't just think – use a computer

You can attempt to build a standard purely by thought, but it really helps if you have an *idiot savant* to assist and check your validity. When you have to explain all the rules to someone who will follow them slavishly *to the letter* then you will certainly get useful feedback. Luckily we have such an assistant, though we have to transcribe the rules we wish to verify into utterances in some form of programming language they understand, and we might conceivably make errors[7] in such transcription.

A specification *editor* really should be developing a working implementation in parallel with the specification. Not only does that give them early indications of the complexity and size of the problems being tackled, but it also helps gain understanding of what the proposed functions really mean, and with suitable examples,

[7] Otherwise known as *bugs*

whether there are significant shortfalls in functionality. Corner cases in input data quickly reveal issues either in the implementation ('null pointer exception' !) or which are unaddressed within the specification.

3.2. The power of the medium-sized example

During the development of the specification many small examples will be suggested, and often used as notes within the specification document itself. Usually these are to illustrate typical anticipated usage of the feature being discussed. For example:

```
bin:shift(bin:hex("000001"), 17) => bin:hex("020000")
```

appears in the Binary specification where a 'long' bit-shift is demonstrated – actually this showed clearly that shifts that moved significantly across byte boundaries (`$shift > 8`) were supported, partly in response to a query from another implementer[8]. Often these simple examples are added to the test suites.

But such examples don't really show *why* the features being described in the specification are useful and how they might be applied to solve useful problems. For this we need examples that *combine* several different features. Obviously the original suggesters of the standard usually have their own ideas for large applications, but to be successful in showing the initial reader how the standard features work together, a few medium-sized examples can be very helpful.

For the Binary module, one of these medium-sized examples involved coding and decoding long ASN.1 integers[9]:

```
<xsl:function name="bin:int-octets" as="xs:integer*">
  <xsl:param name="value" as="xs:integer"/>
  <xsl:sequence select="
     if($value ne 0)
     then (bin:int-octets($value idiv 256),$value mod 256)
     else ()"/>
 </xsl:function>

<xsl:function name="bin:encode-ASN-integer" as="xs:base64Binary">
  <xsl:param name="int" as="xs:integer"/>
  <xsl:variable name="octets" select="bin:int-octets($int)"/>
  <xsl:variable name="length-octets" select="
    let $l := count($octets) return (
      if($l le 127) then $l
      else (let $lo := bin:int-octets($l)
            return (128+count($lo),$lo)))"/>
  <xsl:sequence select="bin:from-octets((2,$length-octets,$octets))"/>
```

[8] Needed for example in a PDP11 emulator written in XSLT.

[9] Used in telecommunications, where numbers of arbitrary length are minimally encoded – but the integers can be so long (e.g. cryptokeys) that even their byte-length values need variable-length encoding.

```
</xsl:function>

<xsl:function name="bin:decode-ASN-integer" as="xs:integer">
  <xsl:param name="in" as="xs:base64Binary"/>
  <xsl:sequence select="
    let $lo := bin:unpack-unsigned-integer($in,1,1,'BE')
    return (
      if($lo le 127)
      then bin:unpack-unsigned-integer($in,2,$lo,'BE')
      else (let $lo2 := $lo - 128,
                $lo3 := bin:unpack-unsigned-integer($in,2,$lo2,'BE')
            return bin:unpack-unsigned-integer($in,2+$lo2,$lo3,'BE')))"
      />
</xsl:function>
```

which has results:

```
bin:encode-ASN-integer(0) => "AgA="
bin:encode-ASN-integer(1234) => "AgIE0g=="
bin:encode-ASN-integer(12345678912345678912345678912345678 9)
  => "Ag8XxuPAMviQRa10ZoQEXxU="
bin:encode-ASN-integer(123456789.. 900 digits... 123456789)
  => "AoIBdgaTo....EBF8V"
bin:decode-ASN-integer(xs:base64Binary("AgA=")) => 0
bin:decode-ASN-integer(xs:base64Binary("AgIE0g==")) => 1234
bin:encode-ASN-integer(xs:base64Binary("Ag8XxuPAMviQRa10ZoQEXxU="))
  => 12345678912345678912345678912345678 9
bin:encode-ASN-integer(xs:base64Binary("AoIBdgaTo....EBF8V"))
  => 123456789.. 900 digits... 123456789
```

This example not only exercised a number functions collectively, it also had the unexpected benefit of testing issues of scale. Early tests showed that small integers were handled correctly (the ASN.1 coding results could be checked by hand for numbers a few digits long...). But the ASN.1 integer was designed to handle numbers of arbitrary size and by using encode/decode combinations we could explore larger usage. The example at 40 digits worked, showing that `BigInt` integer values were no problem, but the next at 900 digits was seriously large-scale, and would involve a variable-length data length field[10]. It simply worked.

The essential requirements on such examples are that they i) are sufficiently complex that compound and possibly non-obvious combination of features are required, ii) involve a subject that should be clear enough to the average reader, iii) small enough that the reader can mentally walk through the example and understand

[10]The 900-digit number requires 503 octets to encode, needing two bytes to encode the octet length, which needs a byte to describe *its* length.

its operation[11] and iv) have clear examples of invocation and result. Needless to say, the example results contained in the specification should be generated *by machine*, and in addition these examples test the ability of an implementation to combine successfully results from several parts of the specification.

In retrospect I feel that a slightly larger example would have been helpful – one that exploited a few more of the library features in combination. The author had worked on parts of a simple SVG-PDF generator (about 100 lines of XSLT), that might have been useful as an appendix in the specification.

4. Fix policy early (and not too often)

Most libraries have a degree of regularity about them – certain types of function have similar signatures, behave (and fail) in similar ways and might be expected to share common semantics about aspects of their behaviour. Some of these can be classed as policies that the library (and indeed other libraries and standards that are of related forms) might impose on its members. Four particular issues which could be considered to relate to policy arose during the course of the development: function names, error handling (and naming), null or empty arguments and access to the 'edge' of binary sequences, and versioning and future-proofing.

Such policies should also take into account other similar policies from the execution environment. For example error codes could have used the prefix-numeric style of XSLT, but in this case a different coherent style from EXPath was used.

Getting these policies explosed and discussed early increases the awareness of the community to the importance and consequences of these common decisions and reduces the risk of large-scale and fundamental changes having to be made across the specification at some late stage.

4.1. The naming of parts

Developers invoke facilities such as functions by *names*. For the library such a name is some form of index that identifies which feature is requested – any set of enumerations would suffice (`fn1()`, `fn2()`...). But for us humans these names should be meaningful, making the general nature of the function being requested clear. They should also be relatively concise, to aid both reading and writing. And if a function has *strong* similarities to another function in a well-known parallel domain, then some similar name might be attractive.

As an example, the first draft contained a function `bin:binary-subsequence()`, which returned a section of some input binary data. In the final specification this was now called `bin:part()`. The `binary-` section was dropped as the `bin:` already

[11] The reader is assumed i) fully competent in the language and ii) cognisant of common paradigms, e.g. recursion and higher-order constructs.

implied it was concerned with binary data. `subsequence` implied strong similarity with the XPath function of similar name, which wasn't quite correct – the `substring()` function had better parallels in the meaning, but '`subbinary`' didn't quite ring true. Hence `bin:part()`.

Often several functions have generally similar behaviour, effectively with some parametric or type variation. The issue then is whether a single function with control parameter(s) should be specified, or several with differing names. We chose to define `bin:binary()`, `bin:octal()` and `bin:hex()` rather than a single `bin:constant($in as xs:string,$base as xs:integer)`, because i) meaning is clearer, ii) there's no need to check whether the base is supported[12]and iii) if a developer does need to choose programmatically between 2, 4, or 8, she can use `xsl:choose`.

4.2. Errors, and how to live with them

Any useful program element can be invoked under erroneous conditions. Some errors might be considered warnings, where graceful degradation is possible. Some errors are comparatively trivial and a default result can be chosen. Some errors are seriously fatal.

Some errors will be generic and likely to occur in a number of functions, such as indexed access outside the range of some binary data. Others will be very specific to a very small number of functions, such as the types of decoding errors that can be encountered in creating strings from binary forms. To be effective error codes used should be specific enough help trap different types of failure for appropriate types of response (e.g. fatal termination, fallback, warning...) and collected into common cases amongst the functions.

It really helps if the error codes are meaningful to the reader. Luckily during the development, a change from prefix-numeric to textual codes was suggested for the specifications of the EXPath community. Hence `err:BINA0006`[13] changed to `bin:octet-out-of-range` as the error raised when a value beyond 8 bits was being used as an octet.

But too many codes can swamp the definition or make error trapping in an application unduly cumbersome. The author probably proposed too many fine-grain error codes such as `bin:index-before-start` and `bin:index-after-end` which were subsequently rationalised into the single `bin:index-out-of-range`. XSLT 3.0's `try/catch` supports providing more detailed information (such as what was the index and data size) through bindings to the `$err:*` variables.

[12]Base 9 is unlikely, base 1 might *just* be credible. Even octal was questioned, though critics hadn't built a PDP11 emulator in XSLT.
[13]A naming protocol deriving originally from W3C/QT tests

Some argue that when a function is being initially defined error cases should be outlined concurrently with functionality. The declarative function catalog (see Section 5.1) encourages that, by including a list of errors raised for each function definition. [Currently the error codes and descriptions are included as a narrative list within the specification body and cross-referred from the function definitions. It might be more coherent to define those codes and their descriptions in the declarative body of the catalog. Then the error codes themselves can be consulted by other tools, for example to check that all error codes have been exercised within a test-suite.]

4.3. Arguments 'on the edge'

Just as the author thought the specification was completed, questions were raised about some edge cases in accessing the ends of data, or in cases with 'empty' data. The subsequent discussions (which involved a little frustration!) overturned some of the already defined, implemented and tested error behaviour in a number of functions. An example situation was with the function:

```
bin:insert-before($in as xs:base64Binary?,
                  $offset as xs:integer,
                  $extra as xs:base64Binary?) as xs:base64Binary?
```

whose functional summary was simply:

> *The `bin:insert-before` function inserts additional binary data at a given point in other binary data.*

This seems comparatively simple but some of its edge cases, and similar situations in related functions, caused extensive rework until quite late in the specification. Normal use of the function is straightforward – concatenate the first `$index` bytes of `$in` with all the bytes of `$extra`, then followed by the remainder of `$in` after the `$index`th byte. However, the problems come when these arguments are not so accommodating:

- `$in` is empty – this can occur in two ways: it could have no binary data (similar to `xs:string('')`) or the argument could be an empty sequence, i.e. the XPath equivalent of null.
- `$extra` is similarly empty.
- `$index` points outside any binary data of `$in`.

The behaviour already defined was very conservative – any access outside the strict limits of the data of `$in` raised an error - even if the index pointed to just before or just after the data of `$in`. Parallels with the function `fn:insert-before()` were not terribly helpful, as its behaviour was liberal, and dated from earlier versions of XSLT where error management wasn't available. Such erroneous conditions on an index were allowed to degrade gracefully, defaulting to pointing to the appropriate

end. After much discussion the behaviour was modified to accommodate 'just on the edge' values for the index and similarly on related functions, However by this time the rework required was not trivial: extensive changes had to be made in three places at once – the functional signature definitions, the working implementations and the by-now-extensive test suites.

This is a case where deciding general principles early in the process (and perhaps deliberately describing them in the specification) would have i) started discussion about these issues early, ii) fixed significant parts of the error model before code was written for it and iii) avoided late and repeated rework.

4.4. Future-proofing – which version am I?

Again, very late in the development process, some 'nice-to-have' features (such as decoding *sequences* of numbers) were proposed, but then agreed to be 'postponed to version 1.1'. The issue arose about how we could i) define and examine which version of the specification was supported and ii) how other future-proofing and backwards-compatibility would be approached. Such mechanisms (required for example with fallback options and conditional compilation in the target language) require something beyond the strict extension library (e.g. adding cases to the response of `system-function()`). In this case pragmatic considerations and perhaps some fatigue, postponed such version identification to a later version[14]!

5. The declarative tool-user

As software engineers we live with Wirth's maxim: "Algorithms + Data Structures = Programs", in that the design of suitable data structures can make algorithms necessary to meet program goals more efficient, robust and flexible. A similar sentiment can be employed in this case – by designing declarative structures for some parts of the specification (as opposed to narrative sections) and employing modest tools to process these structures, we can increase the robustness and most importantly the *flexibility* of the specification considerably. Some forms of late change can require nothing more than alteration to a declaration and automatic reprocessing. To put it another way:

Copy and Paste is not necessarily your friend

5.1. Anything to declare? – Plenty!

XSLT is at heart a declarative, rather than an imperative, language. We're encouraged to define statements (in a tree form) about *what* is true, rather than intricate formulae to compute such information. Features such as tables, maps, archetypical (tree) data

[14] An interesting example of 'self-non-reference' ?

structures and the like can be used comparatively easily to define useful relationships and can be consulted with XPath and small fragments of XSLT code. They also encourage definition *in one place only*.

For the Binary module there was a 'function catalog', using the format employed for the standard XPath built-in function library[15]. This contains a declarative list of functions, detailing their names, signatures, semantic summary, detailed rules of behaviour, error conditions, examples and notes. An example entry was:

```
<fos:function name="insert-before" prefix="bin">
  <fos:signatures>
    <fos:proto name="insert-before" return-type="xs:base64Binary?">
      <fos:arg name="in" type="xs:base64Binary?"/>
      <fos:arg name="offset" type="xs:integer"/>
      <fos:arg name="extra" type="xs:base64Binary?"/>
    </fos:proto>
  </fos:signatures>
  <fos:summary>
    <p>The <code>bin:insert-before</code> function inserts additional
      binary data at a given point in other binary data.</p>
  </fos:summary>
  <fos:rules>
    <p>Returns binary data consisting sequentially of the data from
     <code>$in</code> up to and including the <code>$offset - 1</code>
     octet, followed by all the data from <code>$extra</code>,
     and then the  remaining data from <code>$in</code>.</p>
    <p>The <code>$offset</code> is zero based.</p>
    <p>The value of <code>$offset</code>
      <rfc2119>must</rfc2119> be a non-negative integer.</p>
    <p>If the value of <code>$in</code> is the empty sequence,
     the function returns an empty sequence.</p>
    <p>If the value of <code>$extra</code> is the empty sequence,
     the function returns <code>$in</code>.</p>
    <p>If <code>$offset eq 0</code> the result is the binary
     concatenation of <code>$extra</code> and <code>$in</code>,
     i.e. equivalent to <code>bin:join(($extra,$in))</code>.</p>
  </fos:rules>
  <fos:errors>
    <p><bibref ref="error.indexOutOfRange"/> is raised if
      <code>$offset</code> is negative or <code>$offset</code> is
      larger than the size of the binary data of <code>$in</code>.</p>
  </fos:errors>
  <fos:notes>
```

[15]The File module, with a genesis some 2 years earlier than Binary, contains the function definitions as narrative text (albeit of regular form) in the specification itself. Fortunately a 70-line XLST program can attempt some reasonable reverse-engineering.

```
      <p>Note that when <code>$offset gt 0 and $offset lt
        bin:size($in)</code> the function is equivalent to:</p>
      <eg>bin:join((bin:part($in,0,$offset - 1),
                    $extra,bin:part($in,$offset)))</eg>
    </fos:notes>
  </fos:function>
```

The specification generation tools (XSLT stylesheets from W3C, with minor additions) can generate a function definition section from such a declaration, which can be requested from the main specification via a processing instruction (`<?function bin:insert-before?>`). Thus for example the groupings and order of such functions in the final document (or even whether the function is to be presented at all) is separated from the actual definition of the function itself.

Whilst this declaration was originally written for use in the specification, its utility is potentially much wider. It can provide data for an online reference (as Saxon's documentation does) or auto-hinting in editors. Simple XSLT tools can collect signatures, or sets of error codes, or generate empty templates for test sets. It can even be referenced (as a signature) from another specification that suggests the use of the given function for some compound purpose.

5.2. Tools help you rework

Despite all the measures outlined earlier, there will always be some rework necessary. With a bit of forethought and some very modest tools, the effort required for such reward can be minimised and the flexibility of the development components (spec., tests, implementation) increased. Here is a very simple example from the test suite whose final format will be QT3:

```
<expand name="binary-to-octets"
         function-name="to-octets" prefix="bin">
  <created by="John Lumley" on="2013-07-18"/>
  <environment ref="binary"/>
  <test-case>
    <description>Octets from a zero-length binary</description>
    <test> $FUNCTION(xs:base64Binary("")) </test>
    <result>
      <assert-empty/>
    </result>
  </test-case>
  <test-case>
    <description>Generate octets from a 4-length</description>
    <created by="Jirka Kosek" on="2013-10-06"/>
    <test> $FUNCTION($man.base) </test>
    <result>
      <all-of>
```

```
          <assert-type>xs:integer*</assert-type>
          <assert-deep-eq>(77,97,110)</assert-deep-eq>
        </all-of>
      </result>
    </test-case>
    ...
  </expand>
```

This will end up being expanded into test cases in QT3 format such as:

```
<test-case name="binary-to-octets-002">
  <environment ref="binary"/>
  <description>Octets from a zero-length binary</description>
  <created by="Jirka Kosek" on="2013-10-06"/>
  <test> bin:to-octets($man.base) </test>
  <result>
    <all-of>
      <assert-type>xs:integer*</assert-type>
      <assert-deep-eq>(77,97,110)</assert-deep-eq>
    </all-of>
  </result>
</test-case>
```

The substitutions involved are extremely trivial and perhaps a good re-factoring editor could make the changes. But the flexibility shown here is i) the actual name for the function can be altered in just one place, ii) common elements within the tests (environment reference, base test name...) are defined *just once*. Code necessary to achieve the expansion is simple - as all data is an XML tree, XPath accessors such as `$common[not(name() = current()/*/name())]` will select all those elements in `$common` (the common elements of the `expand` parent) that are not overridden in the specific test-case.

There are many other cases where simple tools operating on declarative descriptions can increase flexibility. In some cases it can even be worthwhile developing a generic macro processor to assist, such as one that supports buried XSLT pull-trees (e.g. `for-each select="2,4,8">.....`)

6. Be realistic – we haven't got all day

We all like our creations to be useful. We also like them to be elegant, long-lived and peer-respected. Paymasters like them to be robust, high-performing, patentable and *cheap*. Some of these goals are invariably in conflict. We only have finite time and resources to refine the specification, and the most effective feedback, actual use, will only appear when real implementations are available. So we need to consider priorities and focus early effort where absolutely necessary (e.g. firming core functions), or which will be cost-effective in the medium term (interpreting declar-

ative representations). But some requirements may have a fundamental conflict with other features of the application environment, such as the purity of the target language.

6.1. Pragmatism vs Purity

Programming languages vary in their theoretical purity from *ad hoc* affairs, such as BASIC or Perl, through to languages that are really frameworks of mathematical declarations and theorems. Extensions in the forms described in this paper can lead to tensions with the underlying semantics of the base language, especially with those of higher theoretical purity. In the case of the binary module and XSLT/XQuery, as the functions are totally pure (i.e. have no side-effects at all), these extensions do not compromise the functional nature of the underlying language.

But we don't have to stray far to find such tension appearing even in something this innocuous. The Binary module provides no facilities to read or write binary data to or from files – for this it relies on other libraries, most notably the EXPath sibling File module[3] which provides three functions, `file:read-binary()`, `file:write-binary()` and `file:append-binary()`. The last two of course *do* have side-effects – it's their sole purpose, to write a file in the outside world.

Now we get to the nub of the tension in this case - suppose we have a program that is creating a PDF file from fragments of SVG[16] by generating sections of binary data for each graphics child of an `svg:svg` element and then appending each result into an output document:

```
<xsl:template match="svg:path" mode="create-pdf" as="xs:base64Binary">...

<xsl:template match="svg:rect" mode="create-pdf" as="xs:base64Binary">...
...
<xsl:variable name="pdf" as="xs:base64Binary*">
   <xsl:apply-templates select="svg:svg/*" mode="create-pdf"/>
</xsl:variable>

...
  <xsl:sequence select="
     for $p in $pdf.parts return file:append-binary($uri,$p)"/>
```

The `file:append-binary()` is used here to accumulate a result by parts within a single file. But there is no requirement that the `for` expression evaluates for each of its iterative values in temporal sequence (as long as the order of the result, which in this case for each is an empty sequence, is preserved). So theoretically the order of pieces in the output document (and hence the apparent draw order in the resulting PDF) may not follow that of the input `svg:*` graphics pieces. This is a case where

[16] There would be more indexing required, but it illustrates the point.

there has to be considerable (higher-order?) early discussion on the approach to take[17].

6.2. Focus on the core

Comments such as "could we have a pattern-directed number-decoder?" are a two-edged sword. On the positive side it shows enough of an interest from a potential user of the library to make such a suggestion. But on the other hand it deflects focus from the core issue – defining the fundamental functions that *must* be extensions, and getting them right. These core functions operate within an environment that has complete computational capability, so a useful approach is to sketch out how such a 'bell-and-whistle' could be written as an XPath/XQuery/XSLT package using the already-defined minimally-required functions[18].

But such minimalism should be tempered with common sense. (In the extreme for the Binary module only two core functions are absolutely necessary: `bin:to-octets()` and `bin:from-octets()` – everything else can be computed in XPath around sequences of bytes, but performance and readability would plummet.) So simple convenience functions that may make code much clearer and will evidently cost little to implement when other fundamental functions are supported should be tolerated. For example, under non-error conditions, the function `bin:pad-left($in,$pad-length,$pad-octet)` is equivalent to:

```
bin:join((bin:from-octets((1 to $pad-length) ! $pad-octet), $in))
```

This is a pretty simple substitution, but given i) that *pad-left* has a clearer meaning and, more importantly, ii) all the machinery for adding byte sequences together is required in any implementation of `bin:join($parts as xs:base64Binary*)`, then the support cost for `bin:pad-left()` will be minimal. Performance will be enhanced too, avoiding the iteration across the repeated padding octets.

6.3. It doesn't *have* to be perfect

This might sound like an anathema; after all we're supposed to be building robust standards. But the real objective is to get a *useful* specification and standard – one that is going to be *used* and then perhaps further developed after substantial experience from application developers. As such there will be times when 'enough is enough', and further bickering over small details may create a great deal of unnecessary delay and frustration. In the Binary module, after the publication of the fourth draft and provided all the necessary core functions were sound, further fea-

[17] Oh dear - here we go trying to understand *monads* again.....
[18] Preferably enlist the help of the original suggester in defining and testing such a package.

tures or alternate 'edge behaviours' could always be emulated by XSLT scripts by those keen enough[19].

7. Conclusion

This paper has taken a stroll through the finalisation of a small specification/standard, trying to extract some useful lessons. What are the most valuable to the author?

- Establish as much policy as possible early on and get it written down near the head of the specification.
- Declare as much as you reasonably can and use the computer to generate therefrom.
- Get several medium-sized examples and expose them to discussion and execution.
- Build a skeletal implementation from the start and use it to run examples, and early tests.
- Build XSLT/XPath/XQuery emulations of some suggested function, using core functionality and use it for discussion, experimentation and definition.

7.1. Acknowledgements

The author must thank several individuals in the EXPath community. Jirka Kosek started the whole thing off, originally proposing the Binary module, drawing up the first draft and commenting on subsequent re-drafts. Florent Georges helped steer the whole EXPath community and gave the author much help on dealing with its systems and tools. Many others contributed in mailing-list constructive criticism, but Christian Grün should be singled out for the depth of his contributions and being amongst the first to build an implementation to the specification, complete with a separate test suite. Finally Michael Kay needs thanks for giving me the challenge of getting a standardised binary library ready for Saxon!

7.2. *Quo vadis?*

Implementations of the Binary module are in the process of publication and will hopefully find application use. From there will undoubtedly flow issues (all minor of course!) and perhaps some suggestions for 'Version 1.1'.

"No peace for the wicked" – the author has another module (on manipulating files in archive format) to be finalised within the EXPath framework.....

[19] Packaging in XSLT3.0 will make such functional extension much easier.

References

[1] *EXPath (: Collaboratively Defining Open Standards for Portable XPath Extensions :)*. http://expath.org

[2] *Binary Module 1.0*. John Lumley. Jirka Kosek. EXPath Community Group. http://expath.org/spec/binary

[3] *File Module*. Christian Grün. Matthias Brantner. EXPath Community Group. http://expath.org/spec/file

[4] *EXSLT*. EXSLT. http://exslt.org