Rethinking XSLT

Sam Wilmott
sam@wilmott.ca
www.wilmott.ca

August 3, 2006

XSLT is a small useful application-specific language that's garnered a lot of popularity. It's been around long enough for there now to have a variety of support tools and literature. So it's a good time to look at it, look at how it's being used, and think how it could be better, especially from a user point of view. This paper addresses the most visible aspect of XSLT, its syntax, and asks does it really need to be written in XML? Suggesting that the answer may be "no", it presents one possible syntactic alternative.

1 Introduction

After working with other programming language issues for the last couple of years, I've recently started working with XSLT. I'm not a fan of using XML as a programming language, so I've been thinking about how XSLT could be improved syntactically – it's a good language from a functional point of view, but I find it overly verbose syntactically.

And we've got to remember why XML was developed in the first place. The W3C folk claimed that they were replacing SGML with XML because markup languages were for machine-to-machine communication, that human beings would use higher-level data entry tools, not plain text editors, and that the common user would never see the markup language. So why the heck are people programming XSLT programs using XML syntax?

Watching others scribble bits of XSLT code on their whiteboards, I notice that there's a tendency to leave out the XML syntactic components. For example, they'll write something like:

```
element "title" ...
```

on the board. (Although in most cases I've seen they'll add a bit of sugar, as in "xsl:elem name 'title'".) So I've been thinking that it's time to work on an alternative syntax for XSLT:

- something closer to what people naturally write down,
- something easy to read, and
- something easy to implement.

What follows is a development of a new syntax for XSLT, that I call RXSLT, for "real" XSLT or "rethought" XSLT or "revised" XSLT, whatever you want. Something that is easier to write and read than the current XML-encoded XSLT language. It's an experiment, and might require more work before it settles down.

The current RXSLT is based on XSLT 1.0. XSLT 2.0, especially with the major changes in XPath 2.0, probably requires a different approach.

This isn't the only attempt at improving XSLT syntacticly. Paul Tchistopolskii has created a good list of XML and XSLT alternatives[1], including his own XSLScript language[2]. Where RXSLT differs is in its attempt to match what people familiar with XML-encoded XSLT already write for their own use.

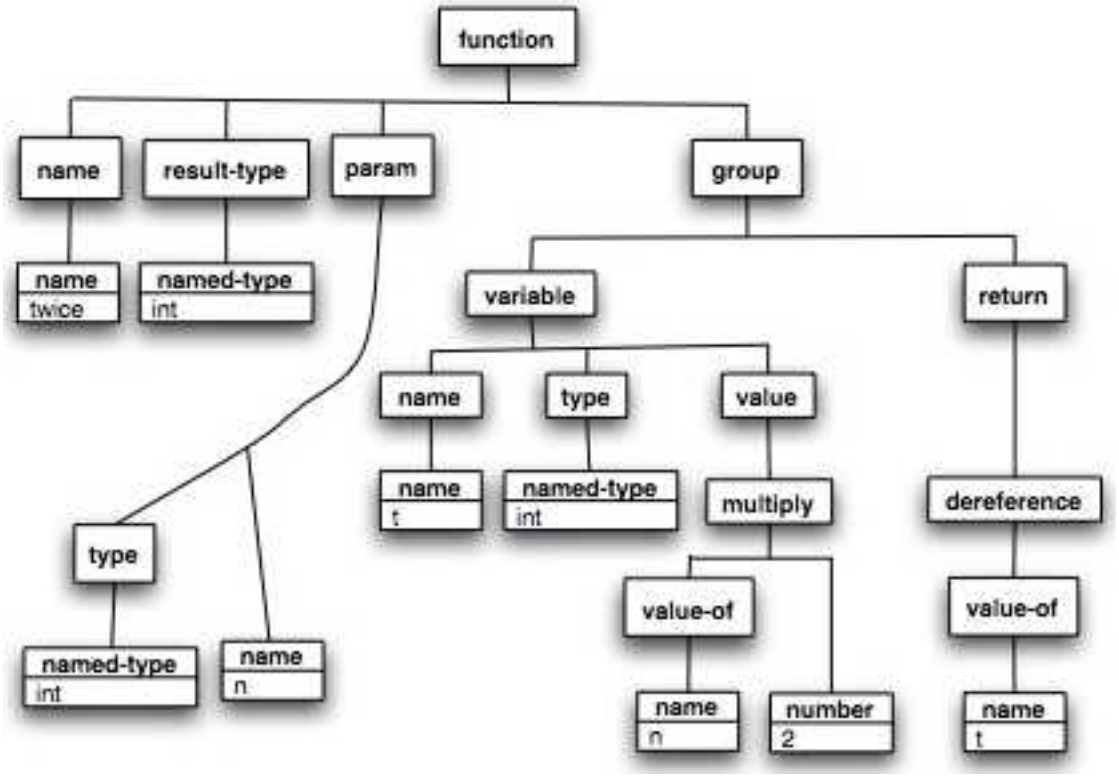
[1] Paul Tchistopolskii. XML Alternatives. www.pault.com/pault/pxml/xmlalternatives.html
[2] Paul Tchistopolskii. XSLScript. www.pault.com/xslscript.html

2 A Short Trip Into The C Compiler Wilderness

It might seem at first that the C programming language is a strange place to start in a discussion of the syntax of XSLT. But you'll see why in a minute or two. Here's a simple function in the C language:

```
int twice (int n) {
    int t = n * 2;
    return t;
}
```

The first task of a C compiler (a program that translates a C program into a form in which it can be performed or executed) is to discover the identity of its components (a.k.a. lexical analysis) and the structure of the program (a.k.a. syntactic analysis). The result of this discovery will be something like this:



This is the "parse tree". (Depending on the language being compiled, the identification and structure may be a more complex process.)

There are many ways in which this structure can be stored during processing. If it desired to save this intermediate result of processing, one possibility is to store it in an XML coding. For example:

```
<c:function name="twice">
  <c:named-type name="int"/>
  <c:param name="n">
    <c:named-type name="int"/>
  </c:param>
  <c:group>
    <c:variable name="t">
      <c:ref>
        <c:named-type name="int"/>
      </c:ref>
      <c:multiply>
        <c:value-of>
          <c:name name="n"/>
```

```
        </c:value-of>
        <c:number value="2"/>
      </c:multiply>
    </c:variable>
    <c:return>
      <c:deref>
        <c:value-of>
          <c:name name="t"/>
        </c:value-of>
      </c:deref>
    </c:return>
  </c:group>
</c:function>
```

3 How XSLT is Different

Life is a little easier for XSLT compiler writers than for C compiler writers, because the first stage of the conversion is already done by the user. In the XSLT world, the programmer/user directly codes the parse tree in an XML encoding. The only thing left for the computer to do is to make the program do something.

This isn't quite the case – it's a bit of an oversimplification – there are those XPath expressions that aren't XML-encoded at all, but look a lot more like C. So XSLT is actually a mixture of an XML encoding of (what would be in another programming language) a program's parsed structure, and bits of unparsed program (the XPath stuff). The result for the XSLT programmer is as if a C programmer wrote the above C function as something like:

```
<c:program version="current C version" xmlns:c="a URI for the C language">
<c:function name="twice" as="int">
  <c:param name="n" as="int"/>
  <c:group>
    <c:variable name="t" as="int*" value="n*2"/>
    <c:return value="*t"/>
  </c:group>
</c:function>
</c:program>
```

This is much more compact than the fully parsed XML encoding above, but is also less compact than using the C language itself. It's also distinctly less easy to read than the C language encoding. Mind you, there's reason for the syntactic differences between XSLT and other programming languages. XSLT has two major features that differentiate if from most programming languages:

- It's a rule-based language. Its functions and data structures (well, XSLT 1.0 doesn't have any, of course) are secondary to its rules. Most of its major processing is (or is intended to be) encoded in the matching logic of its template rules.
- It's a template language. The shape of the output can be directly represented in the XSLT program by "result elements". Large chunks of output can be specified all at once, as-is, rather than by a sequence of instructions in the language.

It's in its being a template language that XSLT most differs in its syntactic requirements from other programming languages. Integrating the requirements of XSLT itself and those of its result elements is where the major syntactic difference lie between XSLT and other languages.

4 Some Notes On Programming Language Design

4.1 Lexical and Syntactic Analysis

Conversion of C source code into the intermediate form described earlier is the simplest task of a C compiler. It's lexical and syntactic analysis. It's a process that's been well understood since the early days of software development. Converting the intermediate form to something that will run on a computer and do something, especially if the goal is for it to run fast and to provide a wide range of features, is much more work, and research is still active in this area.

So if lexical and syntactic analysis is so easy, why have implementers reverted to using XML parsers for this task? Well:

- Because XML is popular. It's "there" for the taking.
- In the W3C community, because they are keen on promoting the use of XML.
- Because alternative tools for lexical and syntactic analysis aren't commonly available or commonly known. Most people still seem to be using Lex[3] and Yacc[4] derivatives, or hand-coding parsers in inappropriate languages like the C++ or Java.

4.2 Program Logic and Data

All computer programs consist of a mixture of program logic and data. In most programming languages, the data is either distinguished by the characters that appear in it (like numbers) or by being lexically delimited (like string literals). Some languages do it the other way around: the still popular TeX language wraps its imperative instructions using
{ and }, and assumes everything else is data. It's been traditional to go one way or the other on this issue: delimit the data or delimit the program logic. The decision as to which way to go depends mostly on what there's more of and what there's less of – you delimit what there's less of – numbers and strings in C, program logic in TeX.

In XML-encoded XSLT, the program logic is wrapped in XSLT namespace prefixed elements (e.g. <xsl:apply-templates/>). So long as there's not clearly a lot more program logic than data – which is generally the case for XSLT 1.0 – this makes sense.

4.3 Breavity and Comprehensibility

Another issue in designing every programming language is that there's a trade-off between breavity and comprehensibility. A programming language that's too cryptic – as many people find Perl, for example – can be hard to read. A programming language that's unnecessarily verbose – as many of us find XSLT – can be equally hard to read.

The requirements of an XML encoding constrain the syntactic design of of XSLT. This means that there are many things that are there for XML's sake, and not for the sake of the user. For example one has to go:

```
<xsl:value-of select="name"/>
```

rather than the briefer but non-XML alternatives:

```
<xsl:value-of name/>
{value-of name} # This is RXSLT, described below.
```

This is where things could be improved for the XSLT user.

[3] Lesk ME. Lex – a lexical analyzer generator. Computer Science Technical Report 39, Bell Laboratories, Murray Hill, NJ, USA. 1975.

[4] Johnson SC. Yacc – yet another compiler-compiler. Computer Science Technical Report 32, Bell Laboratories, Murray Hill, NJ, USA. 1975

4.4 Dynamic vs. Static Programming Languages

Dynamic programming languages are a currently a very popular topic. Perl, Python and Ruby, the current leaders in the field – pop-wise if not technically – have wide-spread use. Dynamic languages are popular because they facilitate rapid development of light-weight applications.

Dynamic languages are characterized by placing a light burden on the programmer: you just do what you want to do and don't have to spend a lot of time specifying what you "really" mean by it. This light burden is most notable in the area of detailing what values are allowed at different points in a program: dynamic languages have minimalistic type systems.

Type systems, class/object systems and the like have their place. The additional work they demand is amply repaid when you're developing large-scale applications with long life times, or where even on modern hardware you need (not just want) to your program to run as fast as possible.

XSLT 1.0 is a dynamic language. This is its most endearing characteristic. The syntactic revision described in this paper is in keeping with this functional aspect of the language: reducing the programmer's burden for small applications.

4.5 Indentation and Instructions

Python uses indentation to indicate nested structure and separate instructions: nested code (within the "then" or "else" part of an "if", for example) is indicated by indentation. And sequences of instructions at the same level are indicated by indenting at the same level. I'm generally not a fan of this technique, because it limits the flexibility of the language's syntax, and can make it hard to extend the language. But it does work well for small languages, and XSLT 1.0 is a small language.

Small languages, like XSLT 1.0, are designed for specific application classes. Their application-specific focus means that a language's implementation can take advantage of specific application properties, and doesn't need the flexibility required of general programming languages.

5 RXSLT

So we've got XML-encoded XSLT with its half-parsed and XML-constrained syntax on one hand, and the syntactic requirements of a rule-based and template language on the other. The question arises, are there other ways of satisfying XSLT's syntactic requirements? The rest of this paper describes one alternative to the current syntax.

I'm calling this new language (actually just a new syntax for an existing language) "RXSLT", for "Real XSLT". "Real" because if this were XSLT, then XML-encoded XSLT could be an intermediate representation for it. Alternatively you can call it "Rethought XSLT" or "Revised XSLT".

Here's a simple example of RXSLT:

```
template doc
  apply-templates
template doc/title
  <H1>{apply-templates}</H1>
template doc/para
  <P>{apply-templates}</P>
```

The equivalent XML-encoded XSLT is:

```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="doc">
    <xsl:apply-templates/>
  </xsl:template>
  <xsl:template match="doc/title">
    <H1><xsl:apply-templates/></H1>
  </xsl:template>
```

```
  <xsl:template match="doc/para">
    <P><xsl:apply-templates/></P>
  </xsl:template>
</xsl:stylesheet>
```

The most obvious features of RXSLT are:

- Only the result elements are XML-encoded. They are really XML, so that's fine. You can put multiple result elements, XML processing instructions and comments anywhere where result elements are allowed.
- No XML tags, namespace, version number or "xsl:" prefixes for XSLT language components.
- Where an XSLT component has one or a primary property, there's no need for specifying attribute names – a template matches, it need not be said that it does.
- XPath expressions are not quoted. They are part of the RXSLT's syntax.
- Names need not be quoted unless they have structure (are attribute value templates, for example). But they can be if you wish.
- Indentation is used to describe structure. Line breaks are used to separate program components. This feature is loosely based on Python. You can alternatively use a colon to prefix nested program components and semicolons to separate components at the same level of nesting.
- Result elements don't need special delimiting: their XML syntax makes them easy enough to recognize for the RXSLT's compiler.
- RXSLT instructions are embedded in result elements are delimited by { ... } in the same manner as XPath expressions in attribute value templates.

This first attempt at a "real" XSLT language is based on XSLT 1.0.

6 A Side-By-Side Comparison of RXSLT and XML-Encode XSLT

The simplest way of describing RXSLT is to describe it in terms of XML-encoded XSLT – primarily because a substantial description of the latter and lots of books on the subject already exists. These examples don't cover the whole of the RXSLT language as will be obvious to XSLT 1.0 users, but are intended to give a sense of how the language works. A more detailed description of the language can be found at http://www.wilmott.ca/rxslt.

6.1 Style Sheets

XSLT style sheets are usually separate from their input data, so that they can be applied to multiple documents. So there's no need to say that a style sheet is a style sheet, in the same way that there's nothing in a C program that says that it's a C program. An RXSLT compiler knows it's getting RXSLT code, and a C compiler knows its getting a C program. And if that's wrong, then there's an error, and the compiler will probably find it.

5.1.1 Explicit Style Sheets

In RXSLT, you can specify the start of the program with specifying the XSLT version, one of the following two right now:

```
xslt1.0
xslt1.1
```

Following the version specification you can specify the top-level attributes similarly to how they're given in the XML-encoded XSLT "stylesheet" element, different only in that you don't need to quote the attribute values if they're just names. You can also specify a list of names separated by commas:

```
exclude-result-prefixes = axslt, bxslt, cxslt
# a.k.a. exclude-result-prefixes = "axslt bxslt cxslt"
```

Another kind of option is binding namespace prefixes to namespace URIs. It's like an attribute specification, but with the namespace prefix prefixed by the keyword "xmlns", as in:

```
xmlns fo = "http://www.w3.org/1999/XSL/Format"
```

These attributes and the space option can be placed on the same line as the XSLT version, or on following lines.

Finally, you can specify the end of a program:

```
end
```

but you don't have to.

5.1.2 Implicit Style Sheets

In XML-encoded XSLT, you can provide a result element instead of an "xsl:stylesheet" element, in which case the result element is implicitly wrapped in an "xsl:stylesheet" element, and a nested "xsl:template" element with a "match="/"" attribute. In RSXLT you can go a bit further:

- There's no need to specify the "xstl1.0" or "xslt1.1" instruction at the start of the program if the version is "1.0" and there are no attributes other than the "xsl:version" and "xmlns:xsl" ones.
- The "xsl:version" and "xmlns:xsl" attributes for the program as a whole are provided by the RXSLT compiler. Don't specify them yourself.
- If the program consists of instructions and result elements that would otherwise appear in a "template" instruction that would be specified as:

  ```
  template /
  ```

  then you can omit the template instruction and just provide the instructions and result elements within it.

  This means that a result element can be whole of a RXSLT program, just as in XML-encoded XSLT, except that it must be entered without the "xsl:version" and "xmlns:xsl" attributes. Alternatively, you can enter RXSLT instructions.

6.2 Defining Templates

There are two kinds of templates: those identified by matching a pattern, and those identified by names. Here's the two forms together with their XML-encoded XSLT equivalences:

```
NUMBER template PATTERN
  TEMPLATE-CONTENT

<xsl:template match="PATTERN" priority="NUMBER" mode="MODE-NAME">
  TEMPLATE-CONTENT
</xsl:template>

named-template NAME
  TEMPLATE-CONTENT

<xsl:template name="NAME">
  TEMPLATE-CONTENT
</xsl:template>
```

The NUMBER is the priority, and is optional. Putting it in front makes it easier to spot when you're scanning through an RXSLT program. MODE-NAME is associated with templates as described later. In RXSLT, unlike XML-encoded XSLT, you can't have a template with both a name and a matching pattern. This makes the language a bit simpler without reducing its functionality.

6.3 Invoking Templates

Here's how you invoke templates:

```
apply-templates with-nodes EXPRESSION with-mode MODE-NAME
  WITH-PARAMS
# "with-nodes EXPRESSION" and "with-mode MODE-NAME" are both optional.
```

It's equivalent to the XML-encoded:

```
<xsl:apply-templates select="EXPRESSION" mode="MODE-NAME">
  WITH-PARAMS
</xsl:apply-templates>
```

Likewise:

```
call-template NAME
  WITH-PARAMS
```

is equivalent to:

```
<xsl:call-template name="NAME">
  WITH-PARAMS
</xsl:call-template>
```

where WITH-PARAMS is zero or more of the "with-param" forms described later, with parameters and variables, and where EXPRESSION is (of course) an XPath expression, unquoted in the RXSLT program.

For example:

```
apply-templates with-nodes title with-mode title

mode title

template *
  <title>{apply-templates}</title>
```

equivalent to:

```
<xsl:apply-templates select="title" mode="title"/>
<xsl:template match="*" mode="title">
  <title><xsl:apply-templates/></title>
</xsl:template>
```

6.4 Modes

Modes are specified for applying templates using the "with-mode" option. Associating modes with templates is done by the "mode" instruction. "mode" is specified at the top level, along with template definitions:

```
mode MODE-NAME
```

The MODE-NAME is associated with each matching template following it until another mode name is specified or "no-mode" is specified, as in:

```
no-mode
```

"mode" allows templates with the same mode to be grouped together in a natural way. (And means you don't have to specify it individually for every template.)

6.5 Element and Attribute Creators

Elements and attributes can be created by RXSLT instructions:

```
element NAME in URI use ATTRIBUTE-SET-NAMES
  TEMPLATE-CONTENT

attribute NAME in URI
  TEMPLATE-CONTENT
```

equivalent to the XML-encoded:

```
<xsl:element name="NAME" namespace="URI"
             used-attribute-sets="ATTRIBUTE-SET-NAMES">
  TEMPLATE-CONTENT
</xsl:element>

<xsl:attribute name="NAME" namespace="URI">
  TEMPLATE-CONTENT
</xsl:attribute>
```

The element or attribute name or namespace can be an "attribute value template", that's specified in the same way as in an XML-encoded XSLT program. The "in URI" is optional, as is the "use ATTRIBUTE-SET-NAMES". ATTRIBUTE-SET-NAMES is a list of names, comma separated, or quoted and space separated. For example:

```
template chapter
  element "{$ns}:div" use chapter-level-atts, all-element-atts
    attribute id
      "div_"
      value-of position()
    apply-templates
```

equivalent to:

```
<xsl:template match="chapter">
  <xsl:element name="{$ns}:div"
               use-attribute-sets="chapter-level-atts all-element-atts">
    <xsl:attribute name="id">
      <xsl:text>div_</xsl:text>
      <xsl:value-of select="position()"/>
    <xsl:apply-templates>
  </xsl:element>
</xsl:template>
```

6.6 Character Instructions

Here's a quick overview of some of the character-level instructions:

```
"TEXT"
unescaped "TEXT"
value-of EXPRESSION
unescaped value-of EXPRESSION
copy-of EXPRESSION
copy USE-NAMES SPACE
  TEMPLATE-CONTENT
for-each EXPRESSION
  TEMPLATE-CONTENT
choose
  when EXPRESSION
```

```
    TEMPLATE-CONTENT
  otherwise
    TEMPLATE-CONTENT
if EXPRESSION
  TEMPLATE-CONTENT
```

equivalent to the XML-encoded:

```
<xsl:text>TEXT</xsl:text>
<xsl:text disable-output-escaping="yes">TEXT</xsl:text>
<xsl:value-of select="EXPRESSION"/>
<xsl:value-of select="EXPRESSION" disable-output-escaping="yes"/>
<xsl:copy-of select="EXPRESSION"/>
<xsl:copy use-attribute-sets="USE-NAMES">
  TEMPLATE-CONTENT
</xsl:copy>.
<xsl:for-each select="EXPRESSION">
  TEMPLATE-CONTENT
</xsl:for-each>
<xsl:choose>
  <xsl:when test="EXPRESSION">
    TEMPLATE-CONTENT
  </xsl:when>
  <xsl:otherwise>
    TEMPLATE-CONTENT
  </xsl:otherwise>
</xsl:choose>
<xsl:if select="EXPRESSION">
  TEMPLATE-CONTENT
</xsl:if>
```

6.7 Parameters and Variables

XML-encoded XSLT supports two ways of binding a value to a variable or parameter: using an XPath-valued argument ("select=") or as more complex content of the element. RXSLT supports these two techniques via assignment and template content, in the same manner, but with somewhat less noisy syntax:

```
variable NAME = EXPRESSION
variable NAME
  TEMPLATE-CONTENT
param NAME = EXPRESSION
param NAME
  TEMPLATE-CONTENT
with-param NAME = EXPRESSION
with-param NAME
  TEMPLATE-CONTENT
```

which are equivalent to the XML-encoded:

```
<xsl:variable name="NAME" select="EXPRESSION"/>
<xsl:variable name="NAME">
  TEMPLATE-CONTENT
</xsl:variable>
<xsl:param name="NAME" select="EXPRESSION"/>
<xsl:param name="NAME">
  TEMPLATE-CONTENT
</xsl:param>
<xsl:with-param name="NAME" select="EXPRESSION"/>
```

```
<xsl:with-param name="NAME">
  TEMPLATE-CONTENT
</xsl:with-param>
```

6.8 Result Elements

Result elements are used "as-is" in RXSLT programs. A less than ("<") immediately followed by a letter signals the start of a result element, and it continues to the end of the element that starts it, as in:

```
template chapter
  <h1>{value-of title}</h1>
  apply-content
```

In this example, the "value-of" is interpolated within the "h1" result element, and the "apply-content" occurs after the end of the result element.

Literal { and } characters are doubled in result element content in the same way as for attribute value templates:

```
template group
  <i>{{{value-of *}}}</i>
```

In this example, the first two {{ represent a literal {, the third { starts embeded RXSLT. Likewise the first } ends the embedded RXSLT, and the last two represent a literal }}.

Here's the XML-encoded equivalents of the last two examples just for reference:

```
<xsl:template match="chapter">
  <h1><xsl:value-of select="title"/></h1>
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="group">
  <i>{<xsl:value-of select="*"/>}</i>
</xsl:template>
```

The latter example especially shows that RXSLT is not always a big improvement on XML-encoded XSLT. But overall it still seems to win.

5.9 Documentation

XSLT 1.0 allows non-XSLT elements to occur at the top level of a stylesheet. They are ignored by the XSLT processor. These elements can serve a variety of purposes, including documenting the stylesheet.

RXSLT has a similar facility. At the top level of an RXSLT program, the keyword "documenation" can be followed by zero or more well formed XML elements, comments or processing instructions, or string literals, as in:

```
documentation "Here's some documentation:"
  <doc:info>
    Some wise words.
  </doc:info>
```

The XML elements, comments and processing instructions are passed through to the XSLT processor. The string literals are converted to XML comments: any "<", ">" or "-" characters are converted in a safe way.

(Any yes, the purpose of the "documentation"'s content can be for other than documentation.)

6.9 XPath Expressions

XPath expressions in RXSLT have the same syntax as they have in attribute values in XSLT 1.0, except that in RXSLT you can break them over lines, indent them and embed RXSLT comments within them in a variety of ways. Spaces and tabs can appear anywhere within an XPath expression where the XPath 1.0 spec says white space is allowed. Line breaks and RXSLT comments can appear anywhere white space is allowed in the following places:

- anywhere within parentheses ("(" and ")") or within brackets ("

  "and"

  "),
- immediately before or following an XPath expression operator ("or", "and", "mod", "div", "=", "+", "-", "*", "|", "!", "!=", "<", "<=", ">", ">=", "/", "//" or "::"), except immedially following an initial (root) "/",
- immediately prior to a predicate, or
- immediately prior to the parenthesized arguments of an XPath function.

There's tricky case vis-a-vis white space. An XPath expression can consist only of a single slash, as in:

```
template /
  element title
```

In this case, there's a potential ambiguity if the expression were to broken across lines after the "/", because the word "element" could be interpreted as part of the template's XPath expression. So, outside of parentheses, RXSLT doesn't allow a line break or RXSLT comment immediately following this initial slash.

XPath expressions can be indented any amount without any effect on the indention rules for RXSLT instructions.

6.10 Using White Space and Splitting Expressions Across Lines

As noted before, RXSLT uses a Python-like indentation model to describe a program's structure. To code the content of an RXSLT instruction, indent each of its content instructions more than the indent on the instruction. Amongst other advantages, the indentation model means that the XML-encoded XSLT end tags don't show up in a RXSLT program.

There are a number of ways in which RXSLT differs from Python in its use of indentation:

- One doesn't need to use anything like Python's
  line-end marker to indication continuation on to the next line. The language is designed so there aren't any cases needing this feature.
- XPath expresions can contain white space and line breaks as described in the previous subsection without any regard to indentation issues.
- A quoted literal string can contain line breaks.
- A name can be quoted or not. If it looks like a name, it doesn't need quoting. A name can also be split into parts and across lines by quoting each part and putting a plus sign between the parts.
- Where a value is known to XSLT as a sequence of names, then they can be specified separated by commas, on the same or on successive lines.

And oh yes: comments in RXSLT start with a # and end at the end of the same line.

6.11 An Alternative To Indentation

RXSLT supports an alternative to using indentation and line breaking to structure a program:

- Template content can be started with a colon (":") rather than or in addition to using indentation.
- Instructions at the same level can be separated by a semicolon (";"). An instruction following a semicolon is considered at the same indentation level as the previous instruction, no matter how it is actually indented. For example:

This alternative structural syntax is useful in a number of cases:

- Putting sequences of instructions in a result element. For example:

```
<h1>{value of $secnum; apply-templates}</h1>
```

- Folding short templates into a line:

```
template title : <h1>{apply-templates}</h1>
template fax : "fax: " ; apply-templates
```

There's a couple of things you have to be careful with when using indentation, semicolons, { and }:

- If you put a semicolon following the "attribute" instruction in the following code fragment, the "value-of" will become part of the body of the "if" – the semicolon puts the "value-of" at the same level as the "attribute".

```
if growth < 0
  attribute style : "color:red" ;
value-of growth
```

- An instruction on the same line as an opening { in a result element has an indent of -1, meaning that anything on following lines is considered to be within the content of that instruction. So if you want to use indentation for multiple embedded RXSLT instructions the best thing is to "establish" the first instructions by starting it on a new line, as in:

```
<h1>{
   if growth < 0
     attribute style : "color:red"
   value-of growth
}</h1>
```

7 Implementating RXSLT

The current RXSLT compiler translates RXSLT into XML-encoded XSLT. Because it made the implemenation easy. It could be better implemented in other ways, but with more work.

It's rather hard for me to estimate how long it took me to implement RXSLT: my life has been very busy with other things recently. The first full implementation, translating RXSLT into XSLT, took about a week. And then I got into dealing with white-space in the resulting XSLT and that bogged me down. Dealing with XSLT white-space issues took at least twice as long as the whole rest of the work put together.

The language was designed in parallel with its implementation – a technique that makes sense for a small experimental language like this. One needs the right tools to do this iterative kind of design. I've used Python, because it's easy to quickly write small programs in it. I've also used the algebraic pattern matching described at http://www.wilmott.ca/python/patternmatching.html because it's easy to read and flexible.

For those interested in object-oriented programming, or those not, you might find it interesting (or shocking) that the RXSLT-to-XSLT translator was implemented without any classes, objects, records, arrays, tables or any other kind of aggegative types: just some useful functions and a lot of pattern matching. This approach is appropriate for small text-based applications, although it doesn't scale up well in all cases.

The RXSLT compiler doesn't do much error checking – an invalid program will generally produce invalid XSLT without complaint. This is a shortcoming for sure and really requires work if RXSLT became used by a lot of people.

8 White Space

White space is the dirty secret of the markup language world. SGML did it one way, and it works well in a lot of cases. XML does it another way (actually two ways: preserve all-white-space nodes or not), and it works well in a lot of cases. But then again SGML and XML both fall short in many cases. The best one can do is provide a simple model that works in many cases, with an out (such as XSLT's <xsl:text>) for those cases where the model doesn't do quite the right thing.

7.1 White Space in RXSLT Programs

White space in XSLT stylesheets is especially problematic because it serves two purposes: for formatting the XSLT stylesheet itself, and for specifying where whitespace should go in the output of XSLT-processed XML data. It's not that this is always a problem – output forms such as HTML are not all that fussy about extra white space (most of the time) – but that it can be a major annoyance where the down-stream process (e.g. web browser, formatter, or data base) cares about white space.

RXSLT makes things quite a bit easier in this area by separating RXSLT's white-space issues from those of the output. Having to put output white space in string literals or in result element content is restrictive in one way, but the overall effect is to give the RXSLT programmer more control. For example, the XSLT 1.0:

```
<xsl:template match="bibref">
  <fo:inline>
    <xsl:text>[</xsl:text>
    <xsl:value-of select="key('bibitems-by-id', @refloc)/bib"/>
    <xsl:text>]</xsl:text>
  </fo:inline>
</xsl:template>
```

becomes the RXSLT:

```
template bibref
  <fo:inline>[{
    value-of key('bibitems-by-id', @refloc)/bib
  }]</fo:inline>
```

The RXSLT actually requires less "quoting" of the text parts. In the XSLT version, the <xsl:text> elements are used to say that the used-for-formatting white space shouldn't go around the "

"and"

". In RXSLT, the judicious use of "{" and "}" means that the square brackets can be placed directly in the result element's content without requiring putting a lot of instructions on the same line, as would be the case for XSLT.

Another area where RXSLT syntax seems to win is in XPath expressions. XPath expressions aren't so bad syntactically, but they can be problematic when you try to squeeze them into XML attribute values. RXSLT doesn't do much other than freed from that context, but doing so makes them quite quite manageble. The XSLT 1.0:

```
<xsl:call-template name="entry">
  <xsl:with-param name="col" select="$col+1"/>
  <xsl:with-param name="spans" select="substring-after($spans,':')"/>
```

becomes the RXSLT:

```
call-template entry
  with-param col   = $col + 1
  with-param spans = substring-after($spans, ':')
```

The extra spaces in the RXSLT XPath expressions is strictly XPath, not RXSLT. But XSLT encourages users to "clump" – not use white space in – XPath expression, to make them more easily stand out from their surroundings. RXSLT removes the surroundings and encourages the use of extra white space for readability.

There have been a few pleasant surprises in what can be done in result element content:

- The RXSLT {"some text"} is equivalent to the XSLT <xsl:text>some text</xsl:text>.
- The use of "{" and "} in result element content even when there are no RXSLT instructions can be used to control white space. For example, in:

  ```
  <fo:block>{
      <fo:block>...</fo:block>
  }</fo:block>
  ```

  the effect of the "{" and "}" is to make sure there is no extra white space in the content of the outer <fo:block>.

7.2 White-space in RXSLT Output

RXSLT takes three different approaches to placing white-space (spaces and line-breaks) in its output XSLT:

1. Setting xml:space="preserve" and not putting any extra white-space in the XSLT output. This approach works best when the user of the output is another computer program, like an XSLT compiler or interpreter. It doesn't work well if a human tries to read or work with the output.
2. Putting all text nodes in the XSLT (other than those in result elements) in <xsl:text> elements, and indenting each nesting level of the "xsl:" elements. This works, but can be a bit clumsy.
3. Indenting, but only using <xsl:text> elements where necessary. This produces more readable XSLT in most cases. It's not necessarily better than always using <xsl:text> elements, because it's not alway easy to get it right. But a computer program can get it right.

Which output form is used is based on command-line arguments to the RXSLT-to-XSLT translator.

All these techniques require tracking the use of xml:space attributes in result elements. Minimizing the use of <xsl:text> also requires making sure that white-space used for formatting and text specified in the RXSLT program don't appear adjacently.

Getting all this right was the bigest part of the RXSLT implementation effort.

8.1 Preserving White Space

In XSLT 1.0 you can specify whether or not white space nodes within "xsl:" elements is preserved. This is an artifact of using XML encoding for XSLT programs, and doesn't apply to RXSLT. All white space in RXSLT programs (like that in most programming languages) outside of strings or result elements is ignored. To enter preserved white space in an RSXLT program, put it in a string.

On the other hand, the "preserve-space" and "strip-space" instructions are in RXSLT, because they apply to the processed XML, not the program itself:

```
strip-space element-name1, element-name2
preserve-space element-name3, element-name4
```

equivalent to the XML-encoded XSLT:

```
<xsl:strip-space elements="element-name1 element-name2"/>
<xsl:preserve-space elements="element-name3 element-name4"/>
```

9 Sources and References

The ideas in RXSLT come from a number of places:

- Python[5] indentation as noted before.
- OmniMark[6] and its precessor, the HUGO typesetting language[7], for the general feel of the language.
- C# and other work done on .NET languages at Microsoft[8], for syntacic embedding XML into non-XML languages.
- XPath[9] and XML-encoded XSLT[10]. itself for the use of { and } in result elements, and the XPath expressions.
- XML[11]-encoded XSLT again, for the words and shape of the language. To a large extent, RXSLT is just XML-encoded XSLT stripped of its XML-encoding. One could come up with a different syntax with a different feel and the same functionality, but at present it seems best to keep RXSLT close to XML-encoded XSLT, so that people familiar with XSLT can give it a look and recognize what's going on without a lot of trouble.

10 XSLT 2.0

XSLT 1.0 is a small language. XSLT 2.0[12] isn't. XSLT 1.0 is a dynamic, rapid-development language. XSLT 2.0 can be used in this way, but taken as a whole it's getting hard to say that its a dynamic language.

The XSLT 1.0 and XPath 1.0 specifications together run to about a 100 pages. XSLT 2.0 and XPath 2.0[13] are each described in a collection of publications, and depend on other resources (XML Schema data types) that together run to over 1000 pages.

XSLT 2.0, far more than XSLT 1.0, cries out for a better way of writing/typing its programs. XPath 2.0 expressions can are way too big to squeeze into an attribute values. XPath 2.0 is itself becoming a more syntactic and functionally complex language. Again as with XSLT 1.0 and XPath 1.0, the problem is not so much with XPath 2.0, as with the practicality of embedding it within XSLT 2.0 programs (OK, stylesheets).

11 Conclusion

It turns out that there are very workable non-XML syntaxes for the XSLT language – languages that are much closer to what people would write naturally, easy to read, and easy to implement. One of these is RXSLT. RXSLT is still very much a work in progress, but playing with it up to now it seems to work well. The language has a number of potential uses:

- If people are going to type XSLT programs, and they find XSLT syntax clumsy, they can use RXSLT instead.

[5]The Python Programming Language. www.python.org

[6]Omnimark Developers' Resources. developers.omnimark.com

[7]Canadian Government Printing Office, Hull, Quebec. The HUGO Language Manual and Report, August 1980.

[8]Erik Meijer, Technical Lead, Microsoft Webdata, Redmond, WA, USA. Programming with Circles, Triangles and Rectangles, 2003 research.microsoft.com/ emeijer/Papers/XML2003/xml2003.html

[9]XML Path Language (XPath) Version 1.0. www.w3.org/TR/xpath

[10]XSL Transformations (XSLT) 1.0. www.w3.org/TR/xslt

[11]XML: Extensible Markup Language (XML) 1.0. www.w3.org/TR/REC-xml

[12]XSL Transformations (XSLT) Version 2.0 www.w3.org/TR/xslt20/

[13]XML Path Language (XPath) 2.0 www.w3.org/TR/xpath20

- If people want to scribble XSLT code (on a white board or elsewhere) in a cryptic but readable way, then RXSLT is the way to go, and it provides a "standard" notation for communicating that way.
- It's implementation shows that a small language can be implemented with a minimum of effort, and that more people should try experimenting with alternatives to XML encoding for program logic.

A more thorough description of RXSLT, its implementation (downloadable) and current status can be found at http://www.wilmott.ca/rxslt.