# Automated Extensible XML Tree Diagrams

John Lumley

HP Laboratories
HPL-2009-137

**Keyword(s):**
XSLT, SVG, XML trees, Functional programming

**Abstract:**

XML is a tree-oriented meta-language and visual description of XML structures often involves the construction of visual trees. These trees may use a variety of graphics for chosen elements and often condense or elide sections of the tree to aid focus, as well as adding extra explanatory graphical material such as callouts and cross-tree links. We outline an automated approach for building such trees with great flexibility, based on the use of XSLT, SVG and a functional layout package. This paper concentrates on techniques to declare and implement such flexible decoration, rather than the layout of the tree itself.

External Posting Date: October 21, 2009 [Fulltext]
Internal Posting Date: June 21, 2009 [Fulltext]

# Automated Extensible XML Tree Diagrams

John Lumley
Hewlett-Packard Laboratories
Long Down Avenue, Stoke Gifford
BRISTOL BS34 8QZ, U.K.
john.lumley@hp.com

## ABSTRACT

XML is a tree-oriented meta-language and visual description of XML structures often involves the construction of visual trees. These trees may use a variety of graphics for chosen elements and often condense or elide sections of the tree to aid focus, as well as adding extra explanatory graphical material such as callouts and cross-tree links. We outline an automated approach for building such trees with great flexibility, based on the use of XSLT, SVG and a functional layout package. This paper concentrates on techniques to declare and implement such flexible decoration, rather than the layout of the tree itself.

## Categories and Subject Descriptors

I.7.2**[Computing Methodologies]**: Document Preparation — *desktop publishing, format and notation, languages and systems, markup languages, scripting languages*

**General Terms:** Languages

**Keywords:** XSLT, SVG, XML trees, Functional programming

## 1. INTRODUCTION & MOTIVATION

Visualising XML structures in graphical forms such as two-dimensional trees is an important part of research and development in document engineering. Being able to examine structures graphically with suitable visual emphases and foci helps understanding and explanation, both in development (e.g. debugging) and in more formal situations such as writing technical papers on XML.

Since XML is principally a tree-based representation, we wished to generate visual trees, with an ability to i) declare flexible graphics for different parts of the structure, ii) elide and de-emphasise sections of the tree, often to reduce the effect of tree 'bulk' and iii) annotate the visualisation with other attention-grabbing graphics, such as callouts or cross-tree links. And to do this automatically on real, sizeable and potentially highly unbalanced XML trees.

This paper describes an approach using SVG[1] as an output display format, an extensible functional graphic layout framework to generate node graphics, exisiting algorithms to lay out the actual tree and XSLT to declare what should be displayed and how.

Figure 1 shows an example plain tree version of a sample XML structure (from Abdul-Rahman[2]) - here we are just displaying the name of the element nodes, but of course there is additional information (attributes, text, namespaces..) attached to that XML.

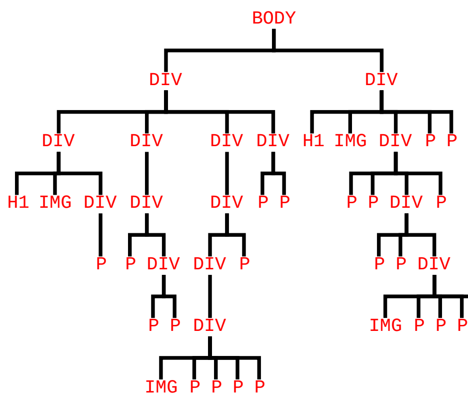
BODY
DIV DIV
DIV DIV DIV DIV H1 IMG DIV P P
H1 IMG DIV DIV DIV P P P P DIV P
P P DIV DIV P P P DIV
P P DIV IMG P P P P
IMG P P P P

**Figure 1.An example tree**

Our primary goal was to develop a highly flexible, declarative and completely XML-based system to support the sort of decorations we might want to make to aid understanding - Figure 2 shows a variety of the techniques outlined above employed on the example: some nodes have an alternate caption, some are represented by a compound graphical structure. Decorations are placed on points on the tree, linking a sequence of some nodes and pointing at others.

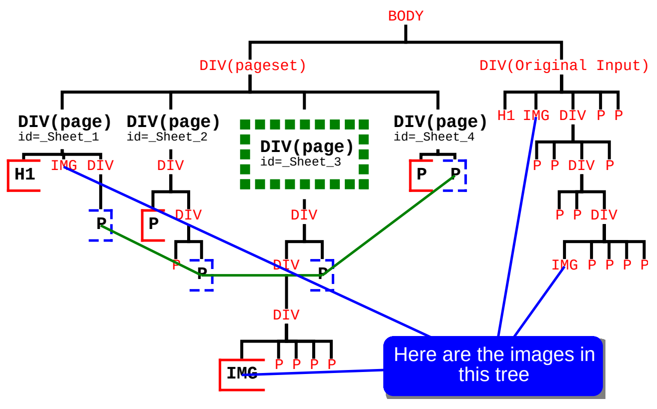

**Figure 2.An example tree with additional decoration**

## 2. METHOD

Our approach involves three steps:

- Preparation of all the graphical atoms
- Layout of the tree of these atoms and interconnection by suitable lines denoting tree structure.
- Addition of any other top-level graphical decorations.

Most of this paper is concerned with flexible techniques for the first and last. To carry out the layout of the final tree, it's been essential to use some algorithm that can build condensed trees, i.e. ones where deep-wide sections of the tree can lie ‘underneath’ shallower sections. For our work here we've used the doubly-recursive technique of Kennedy[3], implemented in XLST2.0 - other methods such as those outlined in Marriott[4] could also be used.

There is a key component to this approach - an extensible declarative layout system. We've used that employed in DDF[5] which supports XML declarations of graphical *combinators* coupled with XLST-based resolution agents, and produces SVG as the canonical output. This can be used to construct significantly complex graphical components, and has a particular agent that supports the layout of parts in a tree-based manner.

We'll briefly describe the layout of the tree (which uses existing technologies) before proceeding to the novel methods for defining graphical atoms and super-decoration.

## 2.1. Tree Layout

Laying out the tree requires a data structure (itself a tree) describing node graphics and parent-child relationships. Later sections describe how this may be formed from an original source XML component. We need to determine the geometric placement of all the graphical pieces, following up by joining with suitable lines. Any suitable algorithm could be used, but condensed (under-running) forms are highly desirable. We use the doubly-recursive algorithm of Kennedy[1] which roughly proceeds as follows:

- For a given node determine the isolated layout for each of its children (by recursion).
- Take the ‘layed-out’ children by adjacent pairs and determine the minimum horizontal separation between their ‘roots’, i.e. when their adjacent vertical ‘edges’ just interfere. (Deeper trees can ‘underhang’ shallower trees this way - these vertical edges are not monotonic over depth). This separation is determined by an inner recursion, describing the (left and right) edges of a tree as a sequence of steps quantised in (geometric) depth - this hides intricacy in the edge shape of graphical atoms - the influence of geometrically ‘deeper’ elements is rounded up to the next level as shown in Figure 3.
- The sum of these separations over the children is taken to be the overall extent of the ‘roots’ of these children - lay them out (average a left-to-right and a right-to-left pass to ‘balance’) .

[1]Implementors should be warned that the coding of such doubly-recursive algorithms is *exceptionally* sensitive to having the correct internal data structures - there are few half-measures: either the layout works or you're left with a bag of bits, often located at 0,0

- Add the display for the node itself above these placed children and determine the edges of the generated ‘layed-out’ node. (This node is then only translated as a whole - relative positions of children are not further modified.)

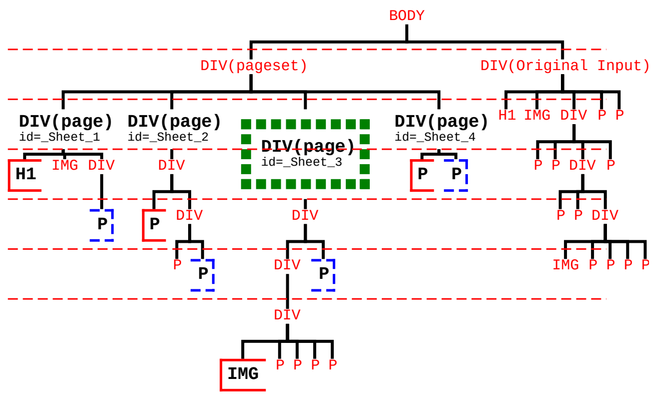

**Figure 3.Quantisation of tree geometric depth**

## 2.2. Building Graphic 'atoms'

An XML tree is simply a set of nodes linked in parent/child/attribute relationships. Display of the parent/child relationship is left to the ‘tree layout’ described above but how do we want to represent a given node graphically ? There are at least three techniques:

- A set of default rules for generating graphical pieces from the source XML.
- Define a set of specific display directives, either as attributes or as dedicated child elements, which can be added to the XML tree to declare required graphic display within sections of the tree.
- Develop transforms that generate specific graphical pieces for chosen sections of arbitrary XML source trees.

Figure 4 shows an example of the use of a set of ‘built-in’ default rules - in this case, using the (element) name of a node to form a text graphic and colour-coding the different namespaces of those elements. These default rules provide a useful fallback.

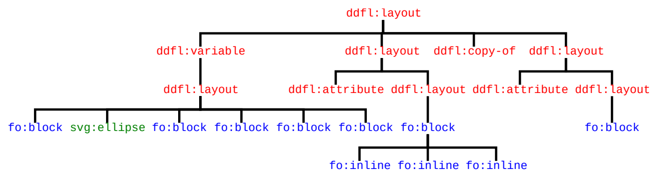

**Figure 4.A simple example tree**

We can define an intermingled format for declaring specific graphics to use for a given node, or some means of overriding the default behaviour. Figure 5 illustrates a case where captions can be declared and specific graphic constructs included - some of the elements have alternative captions related to their meaning, and another has a suitable graphic, in this case an arrow. Figure 6 shows some of the defining XML, with attributes and elements named `ddfl:tree-`

`caption` carrying the definition of the caption or element requested. Using a reserved namespace for such components would reduce interference between these decorations and other manipulation - for historical reasons we've used reserved names in the namespace defining document layout.

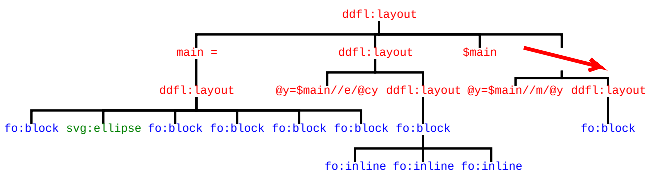

**Figure 5.A tree with altered captions and node graphics**

```
<ddfl:layout function="flow" direction="x">
  <ddfl:variable name="main" ddfl:tree-caption="main =">
    <ddfl:layout function="flow"> .... </ddfl:layout>
  </ddfl:variable>
  <ddfl:layout function="position">
    <ddfl:attribute ddfl:tree-caption="@y=$main//e/@cy"/>
    <ddfl:layout function="encap"> .... </ddfl:layout>
  </ddfl:layout>
  <ddfl:copy-of ddfl:tree-caption="$main" select="$main"/>
  <ddfl:layout function="position">
    <ddfl:tree-caption>
      <svg:polyline stroke="red" fill="none"
                    points="-10 0 10 5 7.5 3 10 5 7 6"/>
    </ddfl:tree-caption>
    <ddfl:attribute ddfl:tree-caption="@y=$main//m/@y"/> ....
    </ddfl:layout>
</ddfl:layout>
```

**Figure 6.Tree graphic declarations**

As we build the overall tree within an extensible functional layout framework, these node graphic element declarations need not be just ‘grounded’ components (such as the `svg:polyline` defining the arrow above) but they can be constructional ‘programs’ (flows, alignments, even trees) that combine various elements together. This is very useful in visualising layout descriptions as XML trees, enabling layouts to be partially evaluated.

The first two methods are suitable for illustrating specific example trees, such as shown in this paper. In general if large numbers of trees are anticipated, for example as a visual debugging aid, an obvious approach is to develop sets of transforms to add appropriate information to or otherwise modify, the source XML tree before it gets handed to the ‘tree-drawing’ package. Figure 7 shows transform fragments used in Figure 2 which add the blue closing ‘bracket’ to elements marked as ‘end’ in the tree.

```
<xsl:template match="*[@mark='end']" mode="caption">
  <xsl:copy>
    <ddfl:tree-caption>
      <fo:block border-style="dashed"
                border-left-style="none"
                border-color="blue"
                font-weight="bold">
        <xsl:value-of select="name()"/>
      </fo:block>
    </ddfl:tree-caption>
    <xsl:apply-templates select="*|text()" mode="#current"/>
  </xsl:copy>
</xsl:template>
```

**Figure 7.Decorating classes of node through transform templates**

Similar techniques can of course be used to modify the tree topology significantly. For example the use of the ellipsis (...) to reduce unimportant repetitive sections of the tree is commonly required. In Figure 8 we've reduced any long contiguous sequence of similar (sibling) elements into a start node with an ellipsis and count.

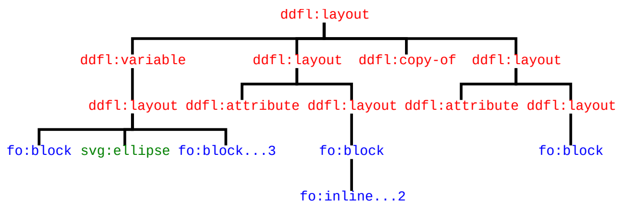

**Figure 8.A tree with elided components**

The simple templates shown in Figure 9 do this, where the first of the sequence has a caption directive added and any of its children and the rest of the following elements are deleted.[2]

```
<xsl:template match="*[name(following-sibling::*[1]) =
name()]">
  <xsl:copy>
    <xsl:sequence select="@*"/>
    <xsl:variable name="n" select="name()"/>
    <xsl:variable name="rest"
    select="following-sibling::*[name()=$n]"/>
    <xsl:attribute name="ddfl:tree-caption"
    select="concat($n,'...',count($rest)"/>
  </xsl:copy>
</xsl:template>
<xsl:template match="*[name(preceding-sibling::*[1]) =
name()]"/>
```

**Figure 9.Templates for eliding repetitive sequences**

Clearly more complex transforms can be developed to give a variety of similar effects, which involve modification of the source tree *before geometry is assessed.*

## 2.3. Additional Super Decoration

The tree itself is the major graphical form of a visualisation, but we might wish to add additional graphical components *between* parts of the tree or *from* parts of the tree to external graphics - examples might be to link sections that involve a cross-reference, or extra-tree callouts. Figure 2 shows some of these, where links between an external label and different nodes in the tree are drawn.

To achieve this we have to i) layout the target tree, ii) identify the final geometric positions of (graphic) nodes of interest and iii) draw the appropriate decorative elements to ‘touch’ those nodes. If such requirements are *acyclic* we can do this using single assignment *presentational variables*[6] to record the result of the layout and then interrogate this result via XPath interpolation to generate or position the final decoration. Figure 10 shows an example where we've drawn the blue dashed line between two leaf nodes:

[2]The ‘rest’ selecting XPath actually has to be slightly more complex to choose only the *current* contiguous sequence - that shown here is illustrative.

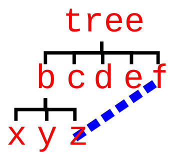

**Figure 10.Super decoration on a tree**

The code to do this is shown in Figure 11, which assigns the result laying out the main tree to a presentational variable `main` and identifies the two nodes `to` and `from` in the result geometry, through an XPath query on the result searching for the `ddfl tree-name` properties (which are copied by the tree constructor from the original source XML onto the appropriate SVG graphic element). A line is then constructed starting and ending at the geometric centres of those nodes in the result and then the tree graphic is written on top.

```
<ddfl:layout function="encapsulate"
             ddfl:evaluate="yes">
  <ddfl:variable name="main">
    <ddfl:layout function="tree" font-size="6">
      <tree>
        <b>
          <x/>
          <y/>
          <z ddfl:tree-name="from"/>
        </b>
        <c/>
        <d/>
        <e/>
        <f ddfl:tree-name="to"/>
      </tree>
    </ddfl:layout>
  </ddfl:variable>
  <ddfl:variable name="from"
  select="$main//*[@ddfl:tree-name='from']"/>
  <ddfl:variable name="to" select="$main//*[@ddfl:tree-
name='to']"/>
  <svg:line stroke-width="1.5" stroke="blue"
            stroke-dasharray="2,1">
    <ddfl:attribute name="
x1" select="$from/@x + $from/@width div 2"/>
    <ddfl:attribute name="
y1" select="$from/@y + $from/@height div
    2"/>
    <ddfl:attribute name="x2" select="$to/@x + $to/@width div
/>
    <ddfl:attribute name="
y2" select="$to/@y + $to/@height div 2"/>
  </svg:line>
  <ddfl:copy-of select="$main"/>
</ddfl:layout>
```

**Figure 11.Defining code for the diagram of Figure 10**

Clearly such techniques can be extended to considerable complexity, without modification of the underlying layout system. However the programmatic model supported in our layout engine is that of a simple 'pull mode' XSLT - involving XPath node selection, choice, iteration and single-assignment variables. Building 'smart' elements, such as 'connectors' that avoid overlap, i.e. thread their way within the tree, would in practice require additional specialist layout agents to be added.

## 3. PRIOR ART

The problem of big trees has been approached by several: Walker [7] developed an early approach; Kennedy[3] generates compact trees with 'centred parents' - the technique used in this paper. Recently Marriott et al[4] tackled denser layout with non-centred parents. There have been a number of simple tree-drawing implementations published, such as Kosek [8] - most of these i) are simple layout implementations (no attempt at condensation) and ii) use simple canonical text-based descriptions for the nodes, or sometimes with constant graphical icons (e.g. folder 'picture') for particular types of node.

None of these provide the possibility of building an arbitrary graphical construct for a node.

## 4. STATUS, FUTURE & THANKS

The techniques described in this paper were of course used to construct all the sample tree visualisations in this paper. Future work is to refine the vocabulary and develop some useful libraries of tree-decorating templates and well as common paradigms for super-decorations.

The author thanks Alfie Abdul-Rahman for posing a display problem which triggered revisiting this work on tree construction and Owen Rees and Roger Gimson for suggestions about this paper.

## 5. REFERENCES

[1] W3C, World Wide Web Consortium *Scalable Vector Graphics (SVG) 1.1 Specification* . http://www.w3.org/TR/xsl/. 2003.

[2] Abdul-Rahman, A., Gimson, R. and Lumley, J. Automatic Pagination of HTML Documents in a Web Browser . In *submitted to :Proceedings of the 2009 ACM symposium on Document engineering*. 2009.

[3] Kennedy, A. *Drawing Trees* . In *Journal of Functional Programming*Vol 6 , no 3 , pages 527 - 534 Cambridge University Press. May 1996.

[4] Marriott, K. and Sbarski, P. Compact layout of layered trees. In *ACSC '07: Proceedings of the thirtieth Australasian conference on Computer science* pages 7--14 Australian Computer Society, Inc.. 2007

[5] Lumley, J., Gimson, R. and Rees, O. Extensible Layout in Functional Documents . In *Digital Publishing, Proc. of SPIE-IS&T Electronic Imaging, Vol 6076*. 2006.

[6] Lumley, J., Gimson, R. and Rees, O. Resolving Layout Interdependency with Presentational Variables . In *Proceedings of the 2006 ACM symposium on Document engineering*. 2006.

[7] Walker, J. *A node-positioning algorithm for general trees* . In *Software Practice and Experience*Vol 20 , no 7 , pages 685 - 705 1990.

[8] Kosek, J. *Automated Tree Drawing: XSLT and SVG* . http://www.xml.com/pub/a/2004/09/08/tree.html. Sept 2004.