Resolving Layout Interdependency with Presentational Variables[♦]

John Lumley, Roger Gimson, Owen Rees
Digital Media Systems Laboratory
HP Laboratories Bristol

HPL-2006-107
August 22, 2006*

XML, XSLT, SVG, document construction, functional programming

In the construction of variable data documents, the layout of component parts to build a composite section with heterogeneous layout functions can be implemented by a tree-evaluating layout processor. This handles many cases with well-scoped structure very smoothly but becomes complex when layout relationships between components cut across a strict tree. We present an approach for XML-described layouts based on a post-rendering set of single-assignment variables, analogous to XSLT, that can make this much easier, does not compromise layout extensibility and can be a target for automated interdependency analysis and generation. This is the approach used in the layout processor associated with the Document Description Framework (DDF).

* Internal Accession Date Only
♦ ACM Symposium on Document Engineering, 10-13 October 2006, Amsterdam, The Netherlands

Resolving Layout Interdependency with Presentational Variables

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

Roger Gimson
Hewlett-Packard Laboratories
Filton Road, Stoke Gifford
BRISTOL BS34 8QZ, U.K.
roger.gimson@hp.com

Owen Rees
Hewlett-Packard Laboratories
Filton Road, Stoke Gifford
BRISTOL BS34 8QZ, U.K.
owen.rees@hp.com

ABSTRACT

In the construction of variable data documents, the layout of component parts to build a composite section with heterogeneous layout functions can be implemented by a tree-evaluating layout processor. This handles many cases with well-scoped structure very smoothly but becomes complex when layout relationships between components cut across a strict tree. We present an approach for XML-described layouts based on a post-rendering set of single-assignment variables, analagous to XSLT, that can make this much easier, does not compromise layout extensibility and can be a target for automated interdependency analysis and generation. This is the approach used in the layout processor associated with the Document Description Framework (DDF).

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, Document construction, Functional programming

1. INTRODUCTION & MOTIVATION

In the development of the *Document Description Framework*[1] (DDF), a major component was an extensible layout processor [2] which interpreted a tree of layout instructions to create composite graphical presentations, the result being described in an SVG tree. This approach has proved very successful in our experimentation, as the tree nature of the 'construction program' maps well to a logical structural view of documents and scoping can be used to considerable advantage where most layout relationships are between a parent and direct children. However, there are some cases where relationships are required that cut across the strict tree. Thus it has been necessary to add means of resolving such relationships, but as the set of layout functionalities available is extensible, a primary challenge was to make a 'universal' system that would be capable of resolving acyclic dependencies.

The basic model of layout used in DDF is that of a declarative tree, where presentation 'instructions' are represented as nodes and the components to be laid out are children of that node. All instructions are expected to evaluate to a canonical form (in our case SVG groups, with a bounding rectangle) and can equally expect their children do. Evaluation is through a recursive tree-descender, choosing a suitable agent to process the given node. In our case this processor is written primarily in XSLT2.0 with a very few Java-based extensions to handle complex layouts, such as linear constraints or specialist leaf nodes, such as line-wrapped text blocks.

Figure 1 is an example of resolving a simple compound layout:

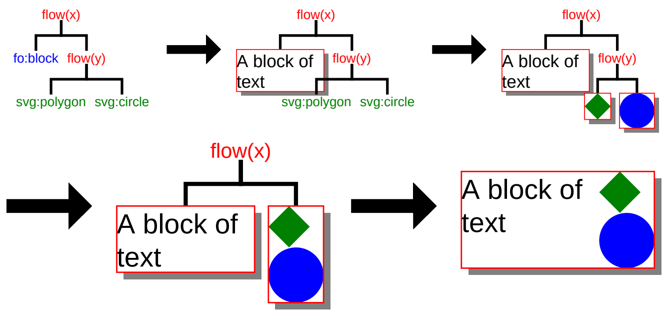


Figure 1. Successive evaluation of a composite layout

This technique can be extended to some apparently complex problems of layout, upto and including forms of pagination, as used in this document. But there are situations in which the layout intent cuts across the tree such as in Figure 2

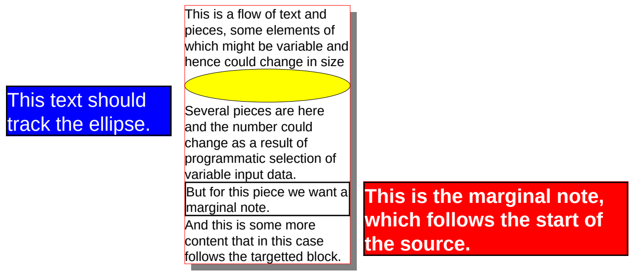

This is a flow of text and pieces, some elements of which might be variable and hence could change in size
This text should track the ellipse.
Several pieces are here and the number could change as a result of programmatic selection of variable input data.
But for this piece we want a marginal note.
And this is some more content that in this case follows the targetted block.
This is the marginal note, which follows the start of the source.

Figure 2. A tree-breaking example

In this case we want the marginal pieces to be placed next to the

relevant paragraph or element, but that component is under a vertical flow (of hetereogenous components, not just a text flow) and hence can move up and down. So we cannot establish the vertical position of the marginal note until the flow has been evaluated, which is carried out in its own tree. To do this we must both schedule the computational order accordingly (evaluate the flow before positioning the marker) and work out how to transfer necessary information from source to target.
Often in document layout there will be 'constructed' components that are repeated in several locations within the document. For efficiency we would like a mechanism to 'render' the component once and reuse the result many times. PPML has 'reusable objects' specifically to support this, primarily for 'bitmap rendering'. It would help if any such mechanism used tree-scoping, in line with the whole philosophy of layout within DDF.

2. PRESENTATIONAL VARIABLES

What we sought was a basic mechanism to declare such interdependencies in the layout 'programs', regardless of what the layout instructions actually were. This should follow tree-based scoping (i.e. we needed some locality of action to avoid having to give unique 'ids' and permit local temporary computations). Supporting cyclic interdependency (i.e. two pieces depending mutually on the other for evaluation) would require knowledge of the layout functionality of these pieces, but for *acyclic* relations, which are very common indeed, such was not required. Consequently our design ony resolves acyclic interdependency.

We wanted this mechanism to be robust, 'universal' with respect to all the other operations involved within layout and implemented relatively easily with reasonable performance. Specific description of interdependency and reuse could be written directly in instructions for this mechanism or created from higher-level declarations and analysis.

Our design was influenced very heavily by the implementation environment of the layout processor, viz. XSLT2.0, its functional semantics and the Saxon processor[3] used. XSLT programs are described within templates or functions as trees, using tree scoping for control of naming. The means of communicating between these trees is via the use of the `<xsl:variable/>` construct, which defines a single assignment (i.e. non-modifiable) binding of a value to a name. These variables may be interpolated via XPath expressions involving both the variables accessed via names and target trees ('context'). The scope of names of these variables follows the XLST program tree. This mechanism is extremely powerful if handled correctly, and with appropriate use of recursion gives much more robust program semantics.

This suggested to us that if we could use a similar mechanism threaded within the layout processor, we should gain some considerable power. Where in the layout processor we had presentational instructions typically of the form `<ddfl:layout function="X"/>` could we introduce variable-assignment instructions of the form: `<ddfl:variable name="NAME"/>`? Unsuprisingly - yes we could, and moreover support this with an implementation that itself was written exclusively within XSLT and a single Saxon extension. The following is a very simple example and its result, where we reuse a pair of components:

```
<ddfl:layout function="flow">
  <ddfl:variable name="rectC">
    <svg:rect stroke="red" fill="none" width="40"
              height="15"
              stroke-width="2"/>
    <svg:circle fill="red" cx="3" cy="3" r="3"/>
  </ddfl:variable>
  <svg:ellipse cx="25" cy="10" rx="25" ry="5"
stroke="black"
               fill="yellow"/>
  <ddfl:copy-of select="$rectC"/>
  <svg:ellipse cx="25" cy="10" rx="25" ry="5"
stroke="green"
               fill="none"/>
  <ddfl:copy-of select="$rectC"/>
</ddfl:layout>
```

Figure 3. Simple variable binding and interpolation

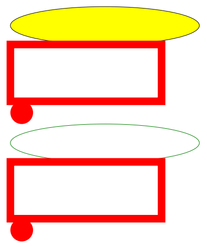


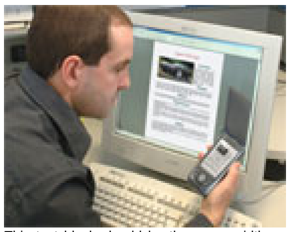
Figure 4. The layout corresponding to Figure 3

This is of course pretty simple and if this fragment was generated by XSLT, using the XSLT equivalents ('variable' and 'copy-of') would give exactly the same result. And since the result is SVG, leaving `use` and `defs` constructs in the result achieves the same 'reuse'. (Both still rely on the layout instruction for 'flow' to determine positions. If the *rectC* piece was replaced by an image of defined width but height determined by aspect ratio, then this would have to be carried out 'post-rendering' of the image. Or *rectC* could be a computed layout of its children, such as a centering, that would only need be computed once.)

But the variable system is capable of more than just simple interpolation, because the primary selection mechanism, like that of XSLT, is through XPath expressions, usually buried in `select` attributes of instructions. This then allows us to 'fish' around inside generated constructions to transfer information into other renderings. Here is a simple example where we set the width of some text to be the same as an image defined only by height:

```
   <ddfl:layout function="flow">
       <ddfl:variable name="image" as="element()">
         <svg:image xlink:href="myimage.jpg"
                       height="65"/>
       </ddfl:variable>
       <ddfl:copy-of select="$image"/>
       <fo:block font-family="Helvetica"
                font-size="4" text-align="justify">
         <ddfl:attribute name="width"
                  select="$image/@width"/>
         This text block should.....
         <ddfl:value-of select="$image/@width"/>
         units wide.</fo:block>
     </ddfl:layout>
```

Figure 5. Variable binding and post-rendering information dependency



This text block should be the same width as the image above, but as the image has only its height defined, then its width depends on the aspect ratio. We transmit that discovered width into the text block for its line wrapping. We find that the image was 82.73 units wide.

Figure 6. The layout corresponding to Figure 5

In this case we extract the discovered width of the image with the XPath expression `$image/@width` and add it to the text-block through the `<ddfl:attribute/>` instruction, which is analogous to that in XSLT. Another expression interpolates this width into the text string presented in this block.

The XPath expressions employed can be anything that XPath2.0 expressions in XSLT2.0 code can be, with presentational variables substituting for those of XSLT. We can use several similar constructs from XSLT within our system of presentation variables: `ddfl:variable`, `ddfl:copy-of`, `ddfl:value-of`, `ddfl:attribute`, `ddfl:if` and `ddfl:choose`. The last two can support post-rendering choice, for example allowing a piece to be placed in a position conditional on its size.

3. IMPLEMENTATION

As described in [2] the layout processor involved is implemented as a push-driven XSLT program, using a 'modal template' recognising the presentational instruction nodes. By this means the processor itself is extensible. So our preference was to disturb this arrangement as little as possible, whilst adding the variable reference mechanism. The design has four parts: i) a 'stack frame' of variable bindings, ii) high priority templates that recognise trees with variable-defining children, evaluate those bindings and interpolate into the other children, iii) templates to support the interpolation and choice instructions and iv) an XPath evaluator which can reference named variables on the stack frame.

The variable-binding stack frame is implemented through an XSLT 'tunnelled' dynamic parameter, which grows and shrinks through the calling tree. Variable 'overriding' through scope is supported by choosing the last binding to a given name in the stack frame. This can be acheived easily by an XPath expression of the form: `$frame[@name=$ref][last()]` where *$ref* is the name of the variable being sought.

Discovering cases which contain embedded variables is merely a case of a suitable pattern, such as `node()[ddfl:variable]`. Processing one of these involves evaluating the children in turn via a recursive iteration function which adds new variable bindings to the stack frame, interpolates variable values and evaluates any 'normal' children. After this, the children contain no further variable processing, so the resulting set can be placed under the original parent instruction, and the parent re-evaluated.

Interpolation and choice is performed by templates that match the instructions, interpret embedded XPath expressions (in `select` and `test` attributes) and generate suitable result fragments.

The last key ingredient is the evaluator of XPath expressions, against a local context and a set of variable bindings. This is carried out in two parts, firstly determining all the variables referenced in the expression. For example in the sequence expression:

```
  $page//svg:rect[@name='blank']@width,
         $containers[@name=$template]/@height
```

which is presumably trying to establish some size for a part, we have three variables: `page`, `containers` and `template`. The evaluator finds these on the stack frame, returning the last value bound for each. (Thus local values can override remote) The expression is transformed to replace the variable names with reserved names ($p1..$p*n*). This expression with the 'values' of $p1.. is then evaluated using Saxon's dynamic XPath evaluator - the only extension to XSLT2.0 used in the implementation. Finally the result type is deduced from context, any necessary coercion performed and the result returned for further processing.

4. RELATED WORK

As mentioned earlier, PPML and SVG both have implicit reference mechanisms for reuse but not information extraction, but neither has specific features for a layout resolution system, being confined to component substitution. SVG's mechanism only supports a global scope of reference names (it uses 'id' which is defined to have such scope) so it clashes with our fully-scoped approach to definition and reference. CSVG[4] does provide a uniform computational machinery that can be used to modify layout, by altering dimensional attributes as a result of constraint resolution, but this does not provide a high-level mechanism of relationships between constructed components. Unlike the system proposed here, CSVG's system produces a set of simultaneous (linear) equations that may have cyclic interdependency - a case that our approach cannot handle. However our variable system can operate within discontinuous layouts, such as pagination. In XSL-FO interdependency between components is completely related to its principal model of a paginated flow.

5. STATUS AND FUTURE DEVELOPMENT

This system is used successfully within the layout processor for DDF and is instrumental in laying out *this document* - variables were used to transfer a column width from a generated page template to line-wrapped text blocks which are then paginated into that template. A next step will be to explore the addition of automated dependency analysis for 'cousin' type relationships, where two elements relate through a common grandparent node, and convert into appropriately scheduled evaluations.

6. REFERENCES

[1] Lumley, J., Gimson, R. and Rees, O. A Framework for Structure, Layout & Function in Documents . In *Proceedings of the 2005 ACM symposium on Document engineering* . 2005.

[2] 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.

[3] Kay, M. *Saxonica: XSLT and XQuery Processing* . http://www.saxonica.com/. 2005.

[4] McCormack, C., Marriott, K. and Meyer, B. *Adaptive layout using one-way constraints in SVG* . http://www.svgopen.org/2004/papers/ConstraintSVG/. 2004.