Extensible layout in functional documents♦

John Lumley, Roger Gimson, Owen Rees
Digital Media Systems Laboratory
HP Laboratories Bristol
HPL-2005-223
December 22, 2005*

XML, XSLT, SVG, document layout, functional programming

Highly customised variable-data documents make automatic layout of the resulting publication hard. Architectures for defining and processing such documents can benefit if the repertoire of layout methods available can be extended smoothly and easily to accommodate new styles of customisation. The Document Description Framework incorporates a model for declarative document layout and processing where documents are treated as functional programs. A canonical XML tree contains nodes describing layout instructions which will modify and combine their children component parts to build sections of the final presentation. Leaf components such as images, vector graphic fragments and text blocks are 'rendered' to make consistent graphical atoms. These parts are then processed by layout agents, described and parameterised by their parent nodes, which can range from simple layouts like translations, flows, encapsulations and tables through to highly complex arrangements such as constraint-solution or pagination. The result then becomes a 'molecule' for processing at a higher level of the layout tree. A variable and reference mechanism is included for resolving rendering interdependency and supporting component reuse. Addition of new layout types involves definition of a new combinator node and attachment of a suitable agent.

* Internal Accession Date Only
♦ IS&T/SPIE Symposium Electronic Imaging, January 2006, San Jose, CA, USA
Approved for External Publication

Extensible layout in functional documents

John Lumley, Roger Gimson, Owen Rees
Hewlett-Packard Laboratories, Filton Road, Stoke Gifford, BRISTOL BS34 8QZ, U.K.

ABSTRACT

Highly customised variable-data documents make automatic layout of the resulting publication hard. Architectures for defining and processing such documents can benefit if the repertoire of layout methods available can be extended smoothly and easily to accommodate new styles of customisation. The Document Description Framework incorporates a model for declarative document layout and processing where documents are treated as functional programs. A canonical XML tree contains nodes describing layout instructions which will modify and combine their children component parts to build sections of the final presentation. Leaf components such as images, vector graphic fragments and text blocks are 'rendered' to make consistent graphical atoms. These parts are then processed by layout agents, described and parameterised by their parent nodes, which can range from simple layouts like translations, flows, encapsulations and tables through to highly complex arrangements such as constraint-solution or pagination. The result then becomes a 'molecule' for processing at a higher level of the layout tree. A variable and reference mechanism is included for resolving rendering interdependency and supporting component reuse. Addition of new layout types involves definition of a new combinator node and attachment of a suitable agent.

Keywords: Document layout, functional programming, XML, XSLT, SVG

1. CONTEXT - DDF AND ITS LAYOUT MODEL

The production of highly customised publications, such as personalised brochures, tuned marketing collateral and custom-branded templates, presents many interesting challenges within document engineering. The customisation required can often involve incorporation into the publication of variable elements whose 'size' is not fixed, which can be used selectively in different document instances and where 'layout' could involve rather complex decisions. This paper describes an approach to defining and processing publication layout in a particular document engineering architecture, based around considering documents as functions of variable data and content.

The Document Description Framework (DDF) is a representation for variable data documents [1], which was designed from the outset to support an extensible family of layout techniques. Briefly, a DDF *document* is a self-contained XML object with application data, logical document structure and graphical presentation spaces, these being linked via XSLT functional programs. The document is also considered a function of external variables. Binding an instance of the document to a particular variable value ('customisation') involves evaluating the programs contained within the DDF document to build new data, structure and presentation components progressively.

The DDF architecture is intended to act as an 'encapsulator' of presentation instructions from a variety of other formats, which can be rendered and potentially manipulated within combinator instructions acting as declarations of layout. It was considered crucial that this set of combinators could be extended to meet new demands from new document applications, in a way that will not disturb previous implementations. In effect we require that the layout repertoire can be extended smoothly.

This paper describes the model of *layout* within DDF documents and how it can support many forms, such as simple flows, grids, tables, constrained geometry, pagination into a set of containers, resizing, packing, optional inclusion and choice and more esoteric methods like guillotine rearrangement, within a single framework. We'll start by discussing what we mean by layout, some of the overall requirements and some of the main assumptions. We'll then discuss briefly current alternative technologies and then proceed to describe the model used within DDF. This will be illustrated through simple examples and then extended through discussion of how more complex arrangements such as declared constraints and pagination can be supported. Finally we'll illustrate how re-use and rendering interdependency can be supported.

This document describes the layout of document - as such it is of course written in itself. The layout examples are themselves buried in the document source, and all other parts are laid out by the system described herein...

1.1. What Is Layout?

It's worthwhile trying to define what we meant by the term 'layout' as used in this paper. Sometimes it could be considered synonymous with 'rendering', but for the rest of this discussion we mean the modification, placement and combination of a series of rendered components to produce a composite presentation. The critical idea is that normally a set of 'parts' are combined geometrically to produce a composite section which will usually appear as a single visual group in the final presentation. (Other parts of DDF are concerned about supporting the generation of appropriate logical structures in the document which can be mapped into suitable visual 'scopes').

Usually when a designer builds a document he starts with a set of pieces and groups which he'll arrange and modify relative to each other to achieve the necessary effect. This can involve designing sets of containers and page layouts, sizing components like pictures, selecting from palettes of permitted styles, and crucially of course selecting which components should go where. This process can happen in two directions simultaneously - bottom up, where components are fitted together and top-down where pieces are packed within containers.

1.2. Automating Layout In High Variability Documents

A major problem in automating such processes is that in *variable content* documents, the 'size' of a component may change as a result of different data bindings: consequently the layout of this component and its siblings (or other relatives) may also vary, *but this cannot be determined until rendering has been carried out.* This is very much the case with text blocks, where normally the height of the block can depend crucially on the actual characters involved, not just their numbers. Similarly image aspect ratios can easily alter by a factor of two with landscape and portrait swaps. (Figure 1 shows extreme examples where the number of characters is constant in the text and the images have the same width.)

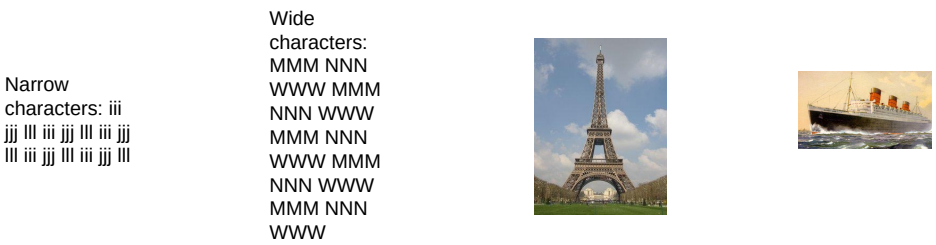

Figure 1. Extreme post-rendering size variation with constant input text length or image width

In documents with high levels of customisation, even the number of components required to be placed may be variable, or some pieces may be entirely optional. In these cases the use of a 'copy-hole' to accept substituted variable data would lead to excessive whitespace and poorly presented final documents.

These factors make it essential that a variable data document system needs a model for layout of sets of component parts, which can be processed automatically and produce effective presentational forms. The goal of course would be to have final presentations which are judged to be of good 'quality' by comparison with those that a professional designer could achieve. For some document types, such as reports and similar ones where layout is based primarily on flows, this is already possible, through the developments in typography, compositing and word-processing technology of the last 30 years. Our interest however is in that set of publications which are*not* flow-based.

1.3. Assumptions & Choices

Whilst a human designer can produce any arbitrary layout of parts, in practice layout involves mostly the following assumptions

- Few components overlap
- Most components are generally rectangular in nature and oriented parallel to the page edges
- Many group layouts involve orthogonal alignment of pieces

These assumptions are not hard and fast of course, but they do handle a majority of cases that designers use and which help clue the reader to intended meanings in the viewed presentation. Indeed many classes of layout, such as the table, rely on this. With this in mind our **principal** model of geometry is:

- The geometry is Euclidean x-y, subject to y +ve downwards of SVG [2]
- SVG is used as the canonical representation form, including geometric transformations.
- Pieces have explicit 'width' and 'height' attributes, or implict sizes dependent upon type - these are used to define the bounding rectangle for that part.
- Most layout occurs between pieces merely on the basis of their bounding rectangle. This means that: i) the type of a component doesn't usually affect its sibling-relative layout and ii) overlap will not occur unless requested intentionally.
- Most layout occurs in orthogonal directions - i.e. x-relative layout is usually separate from y-relative.

In addition we make the following design choices:

- Most of the layout will be defined with 'bottom-up' semantics - the combination of principally unmodified pieces to build larger assemblies. Top-down actions (like pagination) will be handled within the scheme by specialist functions.
- It should not normally be an error for sequences to be empty

As we will see later, our design against these choices involves the declaration of a *constructional tree*, coded in XML syntax, which describes all the pieces involved in the document and how they should be combined or otherwise modified to put this together. We use the tree to contain both primitive pieces, combinators, constraints and guides, larger scale document constructs (like paginations) and as importantly, use the tree scope to control the extent of effect.

2. OTHER RELEVANT TECHNOLOGIES

Most document processing systems and languages have a defined model of the layout of combination of pieces, the majority of which are concerned with 'flow' layouts of sequences of text words. The Scribe-influenced formats (Scribe, Troff, TeX[3]) all pioneered the notion of explicit document layout markup with limited extension mechanisms such as macros, traps and diversions, some with extensive libraries built on top of these base facilities. Some aspects of these models for text flow have influenced significantly commerical products such as MSWord and Framemaker.

2.1. Recent XML Formats

In more recent times there have been a number of XML document formats which take a different approach. Scalable Vector Graphics[2] (SVG) defines a canonical model for a set of graphical objects, with appropriate fixed transformations such as translation, scaling, rotation, clipping as well as colour combination models. It also uses an encapsulation mechanism defining new 'molecules' with its own relative co-ordinate system, akin to those of Postscript or PDF. As all pieces are of defined size and position, layout of parts relative to each other can be 'calculated' prior to rendering.

XSL-FO[4] is a model developed primarily for paginated, flowed documents, though like most others it can be 'pushed' to generate other forms through suitable programming. Its layout model is principally that of a 2D flow of areas (usually text/words) into page containers. For our purposes it has an attractive model for text definition.

Personalized Print Markup Language [5] (PPML) is a model for the construction of paginated documents by the superposition of *pictures* of defined size. A PPML document contains a descriptions of pages where a set of leaf elements (the 'pictures') are combined with simple geometric transforms of translation, scaling, rotation and cropping. The leaf elements contain instructions for making the 'picture' of itself, as well as an indicator of the type of agent required to interpret those pictures. PPML defines the semantics of how those resulting picture elements are to be combined to make the completed page. Additionally PPML defines an orthogonal scoped mechanism for defining and using *Reusable Objects*, including some constraints to improve implementation efficiency over extremely large variable print jobs. But crucially the relative layout of PPML pieces is not defined programmatically by PPML and hence it doesn't support variable content layout in itself.

2.2. Automated Non-flow Layout

Several recent research projects have explored automatic layout of non-flow documents, for both print and electronic form. One approach declares constraints between component parts and couples the form of these constraints with suitable runtime solvers, either in a publishing stage or dynamically within a browser. Constrained SVG [6] is major example, which adds declared constraints between various values (usually dimensions) to a fragment of SVG. This constraint network can connect to external parameters if required, or other variables such as animation trajectories. Solution of the

constraints dynamically within a renderer can reposition or size various components in response to changes. The constraints are limited to linear inequalities: using an effective Simplex solver means quite large problems can be addressed. CSVG doesn't have a scoping mechanism - rather a solution to the whole SVG is computed at once. Later work[7] added a library of common 'lookup functions' within the renderer to attach to component properties such as position and size, making building constraint equations easier and supporting dependency analysis.
Hurst[8] shows using 'one-way' constraints within a web browser where relationships between top-level display components are declared and solved. Recently Hurst et al[9] have shown an approach to describing the layout of textual tables as the solution of a large convex constraint set - this is an example of composite layout of a whole set of independent flows, adjusting the text blocks to meet global optimisation. Jacobs[10] describes laying-out grid-based documents which adapt to variable device formats. This is acheived through inferred constraints from examples and automated selection from a suite of potential templates, this search being capable of working over a paginated document.

These presented so far are examples where the topological arrangement of the parts is preserved, but there are important classes of documents (brochures for example) where this need not be preserved, at least globally, and geometric reordering is permitted to get more pleasing layout. Purvis[11] explores generating layouts by searching a space of guillotined layouts with genetic algorithms.

3. HIERARCHICAL LAYOUT

Throughout this paper our main focus is on viewing the presentation of a document as a hierarchical assembly of subcomponents. This hierarchy typically descends from pagesets through sub-page assemblies, such as columns and frames to groups of primary components like images, text blocks and graphics. As one aspect of successful message communication is a visual correspondence between the intended document *structure* and the presentation, using the hierarchy as the *primary* description will make preserving that correspondence easier.

Our basic model thus treats layout mainly as a tree of components and assemblies, and declares the assembly instructions as a parent node over a set of child subcomponents. Thus in the following tree:

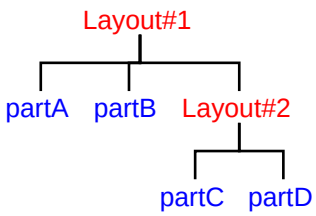

Figure 2. A simple compound layout tree

there are four primitive parts A, B, C and D. Of these C and D are combined, through whatever semantics Layout#2 has, to give some composite molecule which is then treated in a similar manner to A and B. Presumably parts C and D are considered to be more tightly bound (or less separable) than A and B, and this is expected to be clued by the layout. [Whether this is actually the case, and what the originator intended is entirely a matter for the document engineer.] Note that this approach doesn't require the leaf parts to be *immutable* - whilst Layout#1 might be a simple translator of its children, Layout#2 might be a packing layout which may resize *its* children (i.e. partC may be resized in the final result) or even reform them if necessary (e.g. it might change colours in children to increase contrast...)

The tree nodes are the points at which the combinators are declared and to which all necessary parameters for control are attached. Most of these tend to be attributes in the XML sense, though it is possible to attach identifiable control structures as children (we'll illustrate this more when we describe how a constrained layout may be described.) The main point is that the bulk of the children are considered to be graphical components which will be adjusted (translated, scaled, modified) according to the semantics of the given layout, to produce a new graphical view. And of course some of these children parts themselves might be layouts, as shown in Figure 2.

This is basically how we *declare* the layout requirement but doesn't say how it should be evaluated. For some of the constraint-based technologies outlined earlier the intention is to attempt to 'flatten' large sections of the document into a single canonical form and solve for all unknowns (e.g. piece positions) in a single operation, then back-applying results

to drive final rendering. Indeed in CSVG the underlying solver can be driven incrementally so this can be very effective for derivative-modification of a document. Our approach to implementation is different: we use the tree to act as a guide to locality of effect, solving pieces mostly separately and combining part results. Both approaches have their own merits, but we believe ours can support extensibility more easily.
With such a scheme the evaluation of layout, that is the construction of the final result, is primarily one of tree-walking, visiting a node and taking action dependent upon the node requirements. In Figure 3 we have a case of a simple combination of text and graphics through vertical and horizontal flows. A simple recursive descent process evaluates children successively, generating a graphical atom or molecule corresponding to whether the child is a primitive or an embedded layout. Figure 3 shows successive stages of this evaluation:

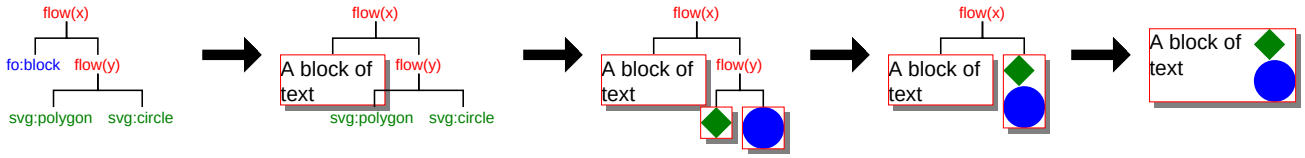

Figure 3. Successive evaluation of a composite layout

(The shadowed portion shows the limits of the evaluated piece returned.) This method can work and be extensible if there is a canonical form for the *results* of a layout. In our case, as given in the assumptions earlier, components are generally 'rectangular' for purposes of layout: most layouts need no knowledge of the component beyond its position and /or size. We'll show later on how specialist requirements which require type-specific knowledge can be handled by appropriate *wrapping functions* .

3.1. Simple Layouts

This canonical form of 'bottom-up' layout (evaluate all the children to graphical molecules, treat them as atoms, arrange them relative to each other and work out composite width and height) is suprisingly powerful. Flows can be defined simply over different types of children as shown in Figure 4

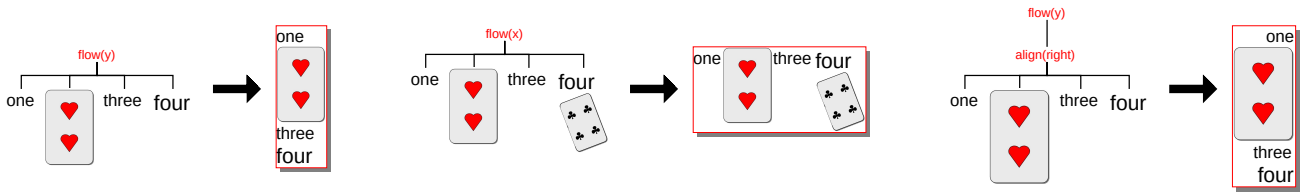

Figure 4. Simple 'bottom-up' layouts; a) vertical flow, b) horizontal flow, c) vertical flow after right alignment

Other more complex computed layouts can be defined, such as that shown in Figure 5 where for some reason we wish to lay pieces out around the circumference of a circle. Again this layout is agnostic to the type of its component children, so we can use different primitives and compounds and still get a robust result.

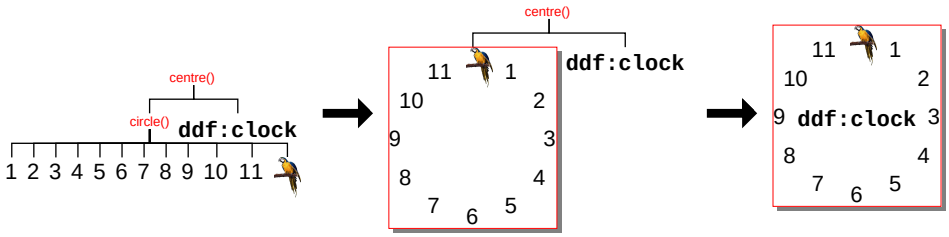

Figure 5. A computed layout of pieces around a circle

3.2. Modifying the Results.

In the previous examples component parts have been translated and agglomerated into higher-level components: such use tends to be very common in layout. But many other useful operations can be described in this 'tree-functional' form, with results that are *sequences* of components. For example if we wish to have a flow of pieces that is ordered in size (Piagetian serration) then we can define a reordering function and use it as a preprocessor for the set of children of the layout as shown in Figure 6:

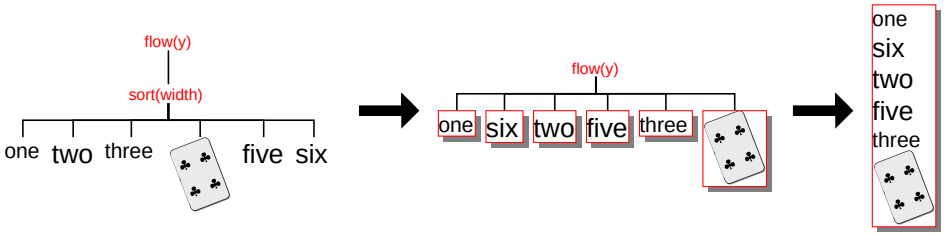

Figure 6. Sorting a set of pieces by width

The point to note is that this manipulation occurs on the graphical components *after* they have been evaluated and thus gives logical capability within the 'visible' presentation space. Similarly a suite of appropriate filters and manipulators can be used in any particular function. In Figure 7 we sort the pictures by height, rotate each one and then lay out only those that are smaller than a given size:

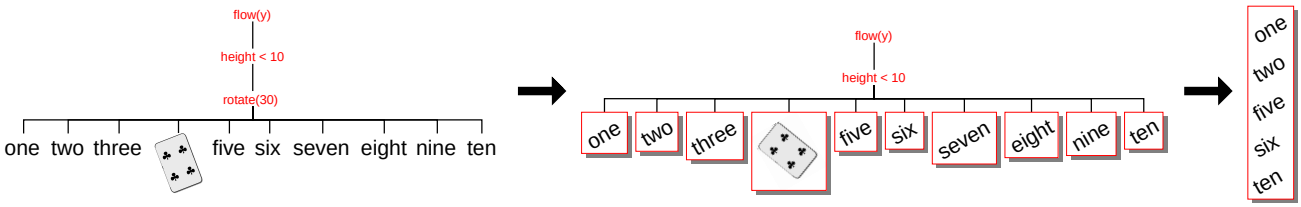

Figure 7. A post-rendering filter, removing 'tall' components

4. COMPLEX LAYOUTS

The layouts we have shown so far are comparatively simple and 'bottom up', though by suitable nesting of layouts fairly complex forms can be created. However an approach to arbitrary document layout should be capable of processing much more complex forms. In this section we consider in some detail how two types of complex layout, constrained geometries and pagination may be declared and supported and then briefly summarise approaches to other types such as grids and tables. In the following section we'll show how information can be shared *between* layouts to achieve more complex effects.

4.1. Geometrically Constrained

As mentioned earlier, there have been several systems that describe layout in terms of geometric constraints between components. Useful concepts such as alignment, spacing and order can be described as constraints between co-ordinates of component boundaries. (This is illustrated well in Juno-2[12] and has been part of CAD systems since Sketchpad[13].) Our implementation within DDF also includes an ability to layout components to satisfy constraints between them. As with others (like CSVG) we restrict ourselves to sets of *linear* inequalities between pieces, exploiting existing powerful

solvers to produce results on relatively large problems. In the general case the layout is a function:

```
constrained-layout(constraints*,contents*) -> composite+
```

where:

- constraints* is a set of constraints between the pieces to be laid out.
- contents* is the set of parts to be laid out. Each of these parts may need to be constructed to reveal both its presentational form, and as a size measure. (It is not an error for this set to be empty)
- composite+ is the resulting graphical molecule.

Effectively the argument to the function is a set of graphs where the nodes are the pieces and the edges describe constraint relationships. We need to represent this within the tree syntax we use. Usually this involves giving nodes names attached to the relevant component (as an XML attribute) and the edges represented as special-type children which name their end-nodes. For example in Figure 8 we define abutting relationships between two pieces, using a simple set of declarative tags, which are expanded to constraints between relative edges of the pieces. (`ddfl:` is a namespace prefix used to identify DDF layout elements).

```
<ddfl:layout function="linear-constrained">
    <svg:circle name="red" fill="red" r="4"/>
    <svg:ellipse name="yellow" fill="yellow" stroke="black"
                 rx="4"
                 ry="2"/>
    <ddfl:constraints layout="abut(below) abut(left)" parts="red yellow"/>
</ddfl:layout>
```

Figure 8. Components and constraints

Figure 9. The resulting layout from the constraints of Figure 8

The tag forms in Figure 9 between *red* and *yellow* have the following equivalences (SVG has y +ve downwards):

- *abut(left)* => `red.x + red.width = yellow.x`
- *abut(below)* => `red.y + red.height = yellow.y`

In many cases all the component parts have defined size and the constraints really act as a 'one-way' network setting positions from sizes, and the dependency order to solve could be derived statically. But constraints can also be used to determine size. For example in Figure 10 we declare the circle to have the same height as the text block:

```
<ddfl:layout function="linear-constrained">
    <svg:circle name="red" fill="red"/>
    <svg:rect name="green" fill="green" width="12"
              height="8"/>
    <fo:block name="text" font-family="Helvetica"
              font-size="2"
              width="20">A block of text that will probably indulge in a spot of line-wrapping</fo:block>
    <ddfl:constraints layout="same(height)" parts="text red"/>
    <ddfl:constraints layout="align(middle) abut(left)" parts="red green"/>
</ddfl:layout>
```

Figure 10. Constraints on the size of a component

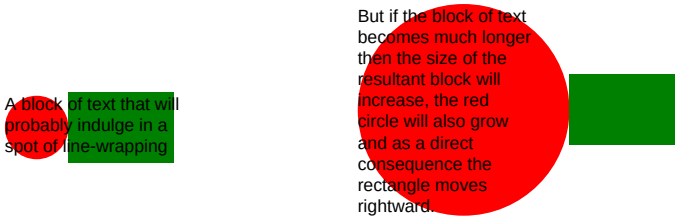

Figure 11. Resulting presentation from Figure 10, with two different bindings of the text

This technique (where the unknowns in the simultaneous equations declared by the constraints include component sizes) is very useful for simple components like circles and images or rectangles. Here the 'self-constraints', those imposed between width and height by the component itself, are either non-existent or fixed and linear (e.g. aspect ratio for images). Unfortunately the important class of text-block does not obey this (width-height is non-linear, potentially discrete and even non-monotonic in the extreme) and other measures are required to solve arbitrary constraint systems involving 'free-width' text. This is a subject that has started to receive some early attention[9] .

In our example we've needed to give the components specific names in order to label the constraint graph edges. However for many situations, especially in deeply nested and scoped designs, the graph is implicit, really being a constant set of edges between successive children. Thus in Figure 12(a) we offset the top-left corners of successive children - this can be described as a set of edges between *child1* and *child2* and the same set between *child2* and *child3*. Similarly in Figure 12(b) we have defined a constant size proportion between pieces.

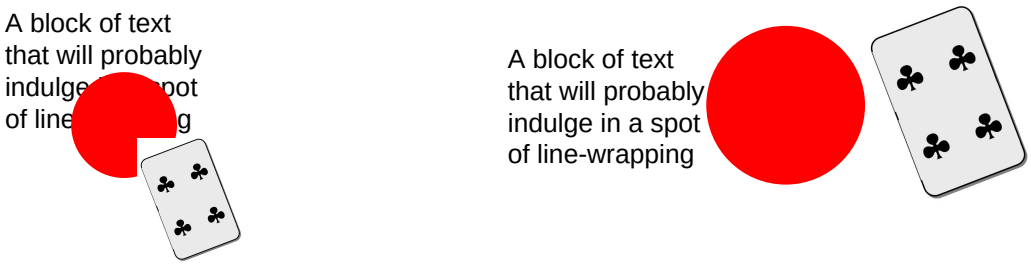

Figure 12. Constant constraints between successive children: a) top-left alignment with an offset of 5; b) centred vertically, abutting horizontally and successive heights in ratio 1:1.25

4.2. Pagination

Printed documents (unlike scrolls) are usually divided into specific sections called 'pages', often a sequence of constant sized physical entities, for reasons of production and reading practicality. Such page sets can of course be filled piece by piece, identifying all components for page 1, then for page 2 and so forth. (Later versions of SVG contain such a model). However an important class of documents include linear 'flows' of material that continue from page to page, such as this document. Almost all document creation tools and formats support such flows - indeed for most the flow model is actually at the core of layout. In XSL-FO for example the basic ideal is a sequences of blocks that flow into a set of containers defined by a page-sequence generator. (It is possible to constrain XSL-FO to finite sets of pages and control placement into a specific page by appropriate manipulation and decoration of the block sequence, but the page set is still the core description.)

As pagination is likely to be an important requirement, how can we define and implement such functionality within our overall model, whilst still preserving the extensibility and generality of our approach to layout? This is made more complex by expectations of whole-document pagination supporting richer functionality such as floating, orphan/widow and page cross-reference, which seems to conflict with our 'combine components' usual approach. However, as we'll show, with suitable programming 'meta-layers', many of these features, which are the exception rather than the rule, can be supported whilst not compromising the general layout philosophy.

First we must realise that 'pagination' is, to first order, a one-dimensional, sequence preserving, bin packing problem, and one for which very simple linear algorithms exist. But in our case we won't know the size of the parts to be packed

until they have been 'rendered'.
We can think of a pagination 'function':

```
paginate(container*,contents*) -> packed-containers*
```

where:

- container* is a sequence (or lazy sequence generator) of one-dimensional containers. What these are is perhaps immaterial at least to first order - but they should provide at least a 'length' (in practice height) to denote their space budget. There is of course no requirement that they are all the same size, nor a finite sequence.
- contents* is the set of parts to flow into these containers. Each of these parts may need to be constructed to reveal both its presentational form, and as far as this pagination is concerned, a length measure (again in practice its height).
- packed-containers* is the set of resulting containers, in sequence, each packed with the pieces that will fit.

So in this representation, using our tree-based description, we have to distinguish between the container* and contents* arguments. There are a number of ways we can do this. The simplest is to identify specially the containers as they are rarer. We can do this with special child class, for example `<ddfl:container/>`, whose children are assumed to be graphical pieces which act as containers. In Figure 13 the grey and white rectangles act as containers into which the sequence of other children are flowed after they have been themselves evaluated. In its purest form the pagination function only requires these to provide an appropriate length measure, but with suitable embellishment the function could use them to act as 'backgrounds' to the filled container.

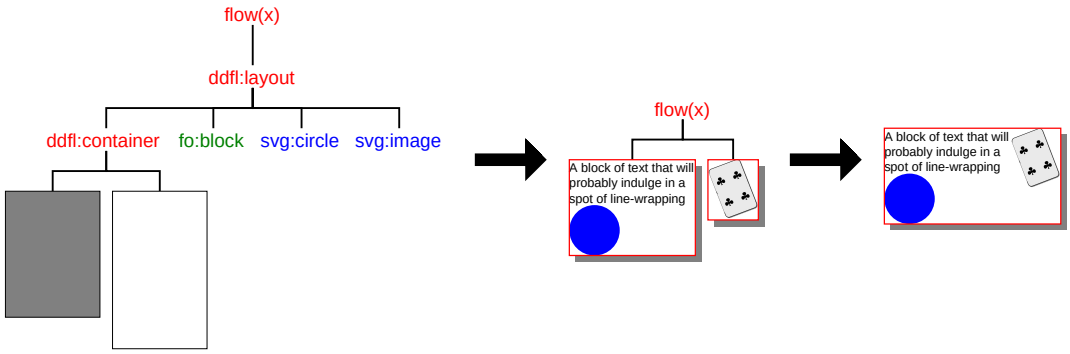

Figure 13. A paginated layout

The result of this pagination is itself a *sequence* of filled containers, with no defined geometrical relationship between them - normally a higher-level mechanism will place these 'children' as required, perhaps in (horizontally-) parallel rows, or spread across separate pages. Note that it is possible to have the filled containers returned in two different views of size: either the original container or a 'shrink-wrap' of the actual contents that fill it.

The implementation of this paginator is relatively simple. The first stage is to identify non-containers and evaluate them to a sequence of graphical molecules. Then the sequence of containers (or more accurately their lengths) is determined and a simple linear packing operation takes place, allocating and translating successive children into the target container until no more will fit. The partitions are then encapsulated and the sequence returned for some high-level processing.

For long sequences of text, described as paragraphs, this approach isn't entirely satisfactory, as our mechanism would focus on a paragraph as an atomic child component of a paginated flow; ordinarily it would be evaluated into a graphical molecule through line-wrap semantics before the pagination. The consequence is that no paragraph would flow across a page boundary. However, this can be solved by a fairly simple and rather general technique. Graphical molecules can be marked with breakability properties (we use `@ddfl:can-break`) to indicate cases where, if required, the piece can be broken into an individual sequence of parts. In this case the implementation of the pagination algorithm, when faced with a piece that 'won't fit', checks for such marking and if found breaks into a sequence of constituent parts and proceeds to reprocess. Of course if the sequence of constituent graphical atoms didn't constitute a flow this will produce nonsense, and the piece shouldn't have been so marked.

This approach to paragraphs flowing across page boundaries illustrates the power of considering layout at a more general level. We could have assumed all flows are primarily text (as in XSL-FO) and other graphical components are treated as special 'characters' and base the implementation around the text-processing loop. But we take the other approach - text pieces are just paragraphs expanded into long columns which are then treated like any other graphical part, and 'fix' the rarer occasions where we need to break the molecule. Examining the pagination of this document should reveal cases where the current semantics and implementation aren't completely adequate.

4.3. Other Complex Forms

In discussing constrained layout and pagination we've tried to show how, with the addition of various special purpose children, structures and attributes, we can declare reasonably complex layouts which generally can combine a lot of arbitrary pieces of content. We can use similar techniques to support other mechanisms, such as tables and grids and other forms of packer. Tables are often used when items being displayed have 'record' characteristics, that is a series of items which each have similar flat internal structure and for which it is intended that the reader makes *comparision* between items based on subcomponents. They can also be used for denser display of a series of items (the table is often used in HTML for this purpose). To support these forms we would need to decide whether we wish to declare the two-dimensionality within the children (i.e. table rows and cells as in HTML) or enumerate the number of columns or rows and partition the child sequence accordingly. In practice we can easily support both forms, albeit with different function 'names' or a smart agent which inspects the form used. As before the normal practice would be to evaluate children as individual atoms before combination in suitable groupings. Some methods can use inequality constraints to solve the result, which would permit additional constraints to be added. In this way we could, for example leave a square 'hole' in the middle of a table to act as a future copy-hole.

5. INTERDEPENDENCY - REUSE AND REFERENCE

Thus far we have dealt with cases where each child component can be evaluated in isolation from other pieces, under parental control. Some parents (such as constrained layout with size-variable pieces like images) can modify one child dependent upon its siblings. However there are cases where the*evaluation* of one piece cannot proceed until another part has been 'rendered'. This is often the case where a text 'container' has a width that depends upon some other piece, such as an accompanying picture.

The ideal of course would be for such dependencies to be detected and suitably processed automatically. This is clearly a 'hard' problem. However with suitable tools the document engineer can achieve results when the dependencies aren't circular. The key is to be able to evaluate (and store) part of a layout, examine properties of it to determine some parameter of the layout of another component, perform that layout and then re-combine partial results. This can be achieved explicitly by a system of single-assignment variables and references, modelled on that of XSLT, but acting in the space of geometric 'rendering'. Figure 14 shows a simple example of the technique.

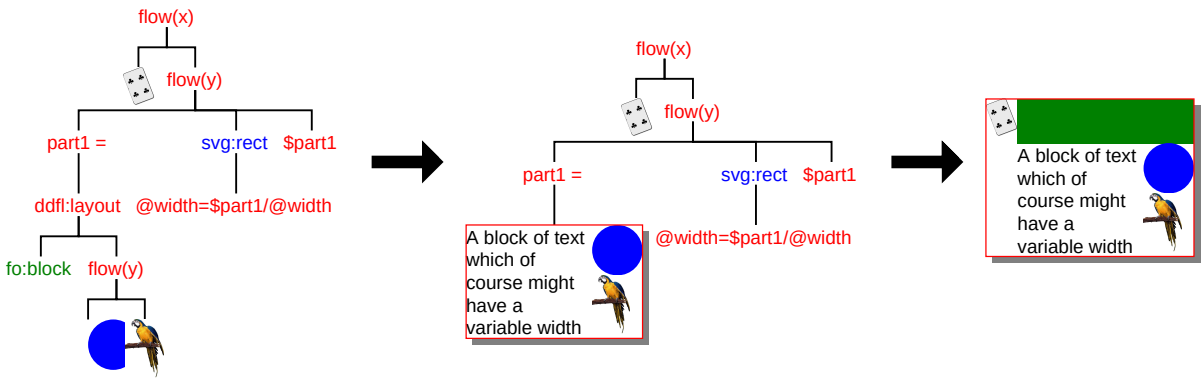

Figure 14. A composite layout using variables

Those familiar with XSLT's variable system will recognise the constructs we use:`<ddfl:variable/>`, `<ddfl:copy-of select="expression"/>`, `<ddfl:attribute/>` etc. which follow the XSLT scoping rules for visibility and usual XPath expression semantics though with layout variables substituting for XSLT variables. For ease of implementation, reference is supported in reverse document order only (c.f. XLST where 'global' variables

can be in mixed order, provided they don't involve circularity). Test and choices on such variable expressions can also be supported, permitting layout variations to be declared based on post-rendered results. Of course a suitable macro processor, or other form of compiler, could be used to project dependencies declared explicitly or implied within other forms, into variables and references.

This system of variables and references can also of course be used to support reuse of part-results in situations such as common sub-assemblies, page templates and the like. No alteration to the underlying mechanism is required to do so - the common component is assigned to a variable in evaluated form and this variable interpolated as required within the appropriate nesting scope. If the final 'rendered' form already has facilities for encoding reuse (such as `REUSABLE_OBJECT` in PPML or `def` in SVG) then in many circumstances variables which are multiply interpolated may be projected into these forms.

6. IMPLEMENTATION AND EXTENSIBILITY

The model for layout we have used so far is reasonably independent of how its evaluation should be implemented. In the context of DDF we tend to use XSLT2.0 as much as possible, given its suitability for push- and pull- driven transformation of XML trees. This is also the case with processing layout. The program is written as a set of `<xsl:templates/>` which match the given 'function' nodes. Figure 15 shows a simple example for a flow, declared by a node `<ddfl:flow-y/>` which illustrates the most common 'bottom-up' methodology - evaluating all the children, manipulating the sequence of results and then encapsulating the result as an SVG 'molecule':

```
<xsl:template match="ddfl:flow-y"
              mode="ddfl:layout">
    <xsl:variable name="children">
        <xsl:apply-templates mode="#current"/>
    </xsl:variable>
    <svg:svg width="{max($children/*/@width)}"
             height="{sum($children/*/@height)}">
        <xsl:for-each select="$children/*">
            <xsl:copy>
                <xsl:sequence select="@*"/>
                <xsl:attribute name="y" select="sum(preceding-sibling::*/@height)"/>
                <xsl:sequence select="*|text()"/>
            </xsl:copy>
        </xsl:for-each>
    </svg:svg>
</xsl:template>
```

Figure 15. Template for simple flow function

Extensibility is then a matter of writing and including appropriate templates for new functions. By suitable use of priorities it is possible to overload and add new functionality based for example on additional control attributes, or the presence of specially identified child components. Careful discipline is required of course, but in practice this hasn't proved a problem, provided all produce a canonical result form (i.e. SVG components each presenting a 'rectangular' size) and as is common practice in XML, 'unknown' attributes on a 'call' are copied onto the result. (This latter point is crucial as a control parameter for specialist treament of the component by a higher layout must be preserved.)

Most leaf types are similarly described - a template matches the description and returns the primitive perhaps further decorated with size information. Some primitive types, such as text blocks, or layout functions, such as constraint declarations, need processing via other algorithms or libraries in other languages. In this case typically an XSLT2.0 template wraps a call to an extension function, pre- and post-processing the argument trees and results to make the function appear to be *geometry components -> geometry composite* .

When it comes to implementation performance, we have not yet carried out specific measurement studies. However the layout of this document itself (which includes a wide variety of examples that are *evaluated within this document*) takes just a few seconds on our unoptimised implementations. However, as in many of these techniques, the implementation engineer must take care not to introduce high-order scaling algorithms inadvertently - whilst most methods used have *O(n)* or *O(n log n)* complexity, very extensive use of deep XPath constructs can introduce higher-order scalings.

7. SUMMARY

We've demonstrated that with a suitably disciplined approach to describing layout as a nested set of functions, we can combine a wide variety of layout types in a single document, and provide a framework for robust extensible document construction tools. Such a system can work well with an XML-based variable data interpolation mechanism. Extension of this idea with a variable-and-reference system akin to that of functional programs means that interdependency and reuse can be supported. This approach to layout is adequate to lay out the entirety of this paper itself, including examples evaluated within the paper construction, as the whole document is written as a variable-content input to a set of DDF documents that constitute a template for SPIE report format.

8. ACKNOWLEGMENTS

The authors are grateful to Xiaofan Lin and Greg Nelson for layout techniques that have stretched our model, and Giordano Beretta and Hui Chao for some specific challenging application examples.

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

3 Knuth, D. *TEX the program*. Addison-Wesley Pub. Co., Reading, Mass. 1986.

4 W3C, World Wide Web Consortium *Extensible Stylesheet Language (XSL)* . http://www.w3.org/TR/xsl/. 2001.

5 PODi, Print On Demand Initiative *Personalized Print Markup Language (PPML) Version 2.0* .http://www.podi.org. 2002.

6 Badros, G. et al. A constraint extension to scalable vector graphics . In *Proc. 10th World Wide Web Conference, Hong Kong* . 2001.

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

8 Hurst, N., Mariott, K. and Moulder, P. Cobweb: A COnstraint-Based WEB browser . In *Twenty-Sixth Australasian Computer Society Conference* . 2003.

9 Hurst, N., Mariott, K. and Moulder, P. Towards Tighter Tables . In *Proceedings of the 2005 ACM symposium on Document engineering*. 2005.

10 Jacobs, C. et al. *Adaptive Grid-Based Document Layout* . 2003.

11 Purvis, L. et al. Document formatting: Creating personalized documents: an optimization approach . In *Proceedings of the 2003 ACM symposium on Document engineering* . 2003.

12 Heydon, A. and Nelson, G. *The Juno-2 constraint-based drawing editor* .DEC SRC Technical Report 131a. 1994.

13 Sutherland, I. *Sketchpad: A Man-Machine Graphical Communication System* . PhD Thesis, Massachusetts Institute of Technology. 1963.