A Framework for Structure, Layout and Function in Documents♦

John Lumley, Roger Gimson, Owen Rees
Digital Media Systems Laboratory
HP Laboratories Bristol
HPL-2005-95(R.1)
December 19, 2005

XML, XSLT, SVG, document construction, functional programming

The Document Description Framework (DDF) is a representation for variable-data documents. It supports very high flexibility in the type and extent of variation supported, considerably beyond the 'copy-hole' or flow-based mechanisms of existing formats and tools. DDF is based on holding application data, logical data structure and presentation as well as constructional 'programs' together within a single document. DDF documents can be merged with other documents, bound to variable values incrementally, combine several types of layout and styling in the same document and support final delivery to different devices and page-ready formats. The framework uses XML syntax and fragments of XSLT to describe 'programmatic construction' of a bound document. DDF is extensible, especially in the ability to add new types of layout and inter-operability between components in different formats. In this paper we describe the motivation for DDF, the major design choices and how we evaluate a DDF document with specific data values. We show through implemented examples how it can be used to construct high-complexity and variability presentations and how the framework complements and can use many existing XML-based documents formats, such as SVG and XSL-FO.

* Internal Accession Date Only
♦ ACM Symposium on Document Engineering 2005, 2-5 November 2005, Bristol, U.K.
Approved for External Publication

A Framework for Structure, Layout & Function in Documents

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

The Document Description Framework (DDF) is a representation for variable-data documents. It supports very high flexibility in the type and extent of variation supported, considerably beyond the 'copy-hole' or flow-based mechanisms of existing formats and tools. DDF is based on holding application data, logical data structure and presentation as well as constructional 'programs' together within a single document. DDF documents can be merged with other documents, bound to variable values incrementally, combine several types of layout and styling in the same document and support final delivery to different devices and page-ready formats. The framework uses XML syntax and fragments of XSLT to describe 'programmatic construction' of a bound document. DDF is extensible, especially in the ability to add new types of layout and inter-operability between components in different formats. In this paper we describe the motivation for DDF, the major design choices and how we evaluate a DDF document with specific data values. We show through implemented examples how it can be used to construct high-complexity and variability presentations and how the framework complements and can use many existing XML-based documents formats, such as SVG and XSL-FO.

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: XML, XSLT, SVG, Document construction, Functional programming

1. INTRODUCTION & MOTIVATION

This paper describes the motivation and design of the *Document Description Framework* (DDF), an experimental high-level framework for the construction of variable data documents. We'll outline some of the issues that appear with high-variability documents and discuss advantages and shortcomings of some of the current 'standards' in supporting such publications. We then review our research goals and posit our major decisions. The current design of DDF is described in three sections corresponding to documents as 'structures', 'functions' and 'layouts'. Examples of documents and their interpretation and processing are presented.

The motivation for this research is a desire to make the construction and processing of *variable data documents* significantly more robust and flexible. An ability to tailor a published document to a particular consumer is seen as a potentially valuable feature in many publication relationships, such as marketing collateral, personalized communications and advertising material in many fields from finance to retail, politics to internal business messages. (PODi [12] has many example cases in printing.) The extent to which customisation can be performed is to a great deal dependent upon three factors:

- an ability to decide what is the appropriate variation in the 'message' for the specific audience,
- the ability of the delivery device to support that customisation efficiently, and
- the means whereby the appearance of the effect of the customisation is defined and determined.

Our interest is in the third of these factors, especially for print situations. Developments in *Customer Relationship Management* systems (CRMs) and marketing tactics are beginning to show possible individually customised messages (e.g. Amazon.com.) Web-based systems, based on 'instantly' computable delivery platforms and extensible presentation spaces have pushed the ability to deliver such customisations very far, leading to standards for various forms of presentation, and extensive workflows.

In high-quality print the delivery devices are much more rigid and potentially either expensive to use, have high inertia or are incapable of high throughput. Recently intermediate *digital presses* have appeared which promise to provide some reasonable compromise and support the goal of 'every page different' at reasonable cost. However in all these cases the 'page' is still a significant boundary. In many cases customisation is limited to overprinting with low quality/high throughput devices, for cases such as name-and-address interpolation.

Constructing high-quality documents in paginated situations has historically been the province of the graphic artist supported with tools such as Quark XPress. These tools have been adapted to handle workflows with modest degrees of customisation, mostly based on the *copy-hole*, and with a very strong desire to preserve

DocEng'05, November 2–4, 2005, Bristol, United Kingdom.

the artist's control over almost-exact final appearance. This can be effective when the variation in 'size' of the substitutional elements is small - text pieces are reasonably the same length or whitespace tolerant, images have substantially similar aspect ratio and the *number and type of the elements substituted is constant.* Standards such as *PPML* [13] have emerged to support efficient description of large customised runs of such documents.
However, when the message must be tailored much more flexibly, such as varying the *number* of products advertised to a particular customer on a given page, such systems are unable to adjust appropriately. Often a series of separate templates has to be constructed or adjusted (by hand) to accomodate the variation, and the workflow modified to select an appropriate template for the given message instance.

Our motivation was to design a document representation that would be a suitable basis for the *complete* automation of the construction of such publications when the levels of flexibility and variation get very high and the styles of 'layout' can be extremely variable. Other longer-term requirements were that:

i) the representation should be extensible,

ii) it enables several types of document 'manipulation' including merging for reuse, partial binding of variable instance values and documents as 'data input' vehicles, and

iii) it should support using and combining many existing document formats describing partial document components.

Given recent developments in XML technologies for both representation and manipulation of documents, we focused on encapsulated representations of documents as 'computable functions'.

2. EXISTING DOCUMENT FORMATS

Several current document formats that support some degree of variability are intimately tied into a (WYSIWYG) editing environment, MSWord being a typical example. Without significant development of 'plug-ins' these tend to fail our requirement for extensibility, as well as the programmability being 'hidden' within the format.

PDF has concentrated on the page-description of high-quality print, and doesn't handle *internal* variability, relying instead on external generators and modifiers to create new PDF files. Its precursor, Postscript is a general programming language with in-built renderer but has lately been avoided as *too uncontrollable* for general use, and replaced by PDF for precision documents.

The Scribe-influenced formats (Scribe, Troff, TeX [8]) all pioneered the notion of explicit document layout markup with limited extension mechanisms such as macros, traps and diversions. Indeed the successful LaTeX [9] was principally a sizeable set of macros built atop TeX. In several of these cases some extension to supporting 'variable data' documents was possible, but mostly it was anticipated that external programs would generate document instances for given data bindings. These mechanisms also encouraged the separation of document structure from presentation style which we'll return to in a subsequent section.

Of the modern 'print-compatible' formats, XSL approaches the issue of variability by providing two separate entities: XSL-FO [16] describing a vocabulary for generic 'flow' style of layout (very suitable for paginated reports), and XSLT [17] being an 'XML-oriented' construction/transformation language. (The usual operation is for an XSLT program to generate a grounded set of XSL-FO instructions which is then interpreted during rendering.)
PPML [13] is a framework for the construction of pre-paginated documents by the geometric combination of 'pictures', coupled with a specific mechanism for the declaration of reusability from levels within a page up to multiple print jobs. The set of supported pictures (image formats such as JPEG, general mechanisms such as Postscript) can be extended. PPML/T [14] adds a binding of an XSLT program to support 'copy-hole' variability, or other computed layouts *that do not depend upon the results of rendering*. These both have demonstrated the possible benefits to be gained through a suitably-chosen level of 'framework'.

3. RESEARCH GOALS AND CHOICES

Earlier we listed our main requirements from documents and framework (complete automation, extensibility, multi-use and multi-format). We expanded these to seeking a framework for describing wide classes of variable data documents with the following requirements on the document:

- It should as far as possible be a *self-contained* object. That is all the information necessary for the construction of the final result when variable data is bound should be able to be held within the document itself. Whilst documents and document components could exist in well-managed repositories, it should still be able to treat a complete document as an 'object'. (PPML has a similar complete encapsulation of all the resources within a PPML 'job' and PPML/T extends this to variable content.)
- It should be *mergeable* with other documents in similar 'families' to encourage reuse. Such merging can occur both in terms of 'data' and 'construction program'.
- It can contain pieces in possibly *different formatting languages* . This is in the same spirit as being able to embed MathML within XHTML, or SVG within XSL-FO.
- A document may be bound to *partial variable data* yielding a result which is capable of processing further bindings of data.
- The document should, as much as possible be *declarative* internally, rather than contain imperative (and order-sensitive) algorithms.

A typical 'workflow' for a DDF document looks something like:

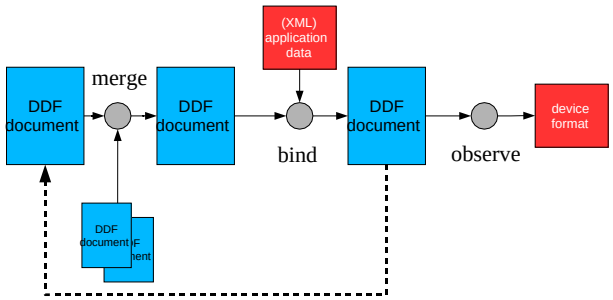

Figure 1. Typical workflow for a DDF document

where some main DDF document may be merged with others (acting as modular templates for example). The result is a valid DDF document. This can be bound to variable data, which may make new internal structures. DDF documents can be reprocessed

(merged, bound...) or 'observed' by extracting or generating some device format form (SVG, PDF, HTML etc.) from presentations buried in the document.
With these stated aims and experience of other formats, the main choices we made were:

- The format syntax would be XML-based
- A document could contain information in three partitioned spaces: (application) data, (logical) document structure and (graphical) presentation instructions.
- The document was to be considered a function of (possibly null) variable data - evaluation of this 'function' on an instance of the data would produce a grounded document
- Construction of new document parts as a result of variable data would be declared in embedded XSLT programs.
- Document 'layout' would be declared through extensible functions in the document's presentational space.
- Internal document functionality shall be *side-effect-free* .

Our research examines what a framework like this would look like, is it evaluable, is it extensible and can it support a wide variety of documents on the single framework. Like other representations it is not intended *per se* to be created/edited directly by the end-user, but support authoring, editing and conversion tools and the formation of libraries. Its usefulness will come from providing the base representation that other document tools may use.

4. BASIC DDF FORM

Our need to merge documents flexibly and reuse styles and programmatic generators, suggests that having an explicit description of the logical structure of the document, separate from possible application data or presentational form would help support desired flexibility. Hence we chose to represent a DDF document in the following XML form:

```
<ddf:doc>
<ddf:data>
application data
</ddf:data>
<ddf:struct>
logical structure + programs
</ddf:struct>
<ddf:pres>
presentation + programs
</ddf:pres>
</ddf:doc>
```

Figure 2. Basic structure of a DDF document

where:

- `<ddf:data/>` holds pure application data
- `<ddf:struct/>` describes logical structures in resulting documents
- `<ddf:pres/>` contains presentational instructions to create final visible forms, and
- 'program' components describe generating new content as a result of variable bindings.

More or 'fewer' layers could be employed but for our current design these three seem adequate. We'll explain how these work together in constructing a business card looking like this:

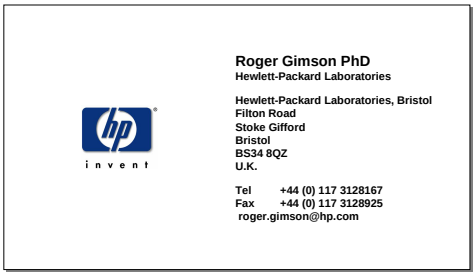

Figure 3. A variable-data business card

We start by making a DDF document that acts as a template and takes some application data as input (we'll describe these as function and argument later). The DDF document contains an application data space to hold instances of bound data.

This space can contain any data that the application sees fit - DDF makes no restrictions and assumes no responsibility for processing this, beyond evaluating programs contained within DDF documents that will typically generate logical document structures as a result. *Variable data* , that is values of the 'arguments' to the entire DDF 'document as function', will appear in this space. At present DDF documents contain no schemas describing valid application data, but such mechanisms could be added.

So to start our process we take the DDF document and bind it to an instance of its variable data:

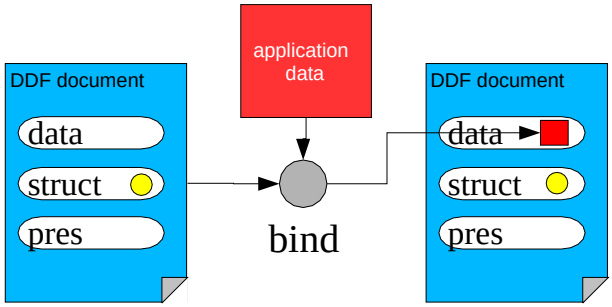

Figure 4. Binding to data

Now the document contains a copy of its binding which will be transported around with it. This stage is not very exciting. To proceed further we need to detemine if new document *structure* will be generated as a result of this data. To do this we evaluate appropriate program (data -> structure) which is embedded in the DDF document or has been merged in from another document. This program sits within the structural space (Figure 5).

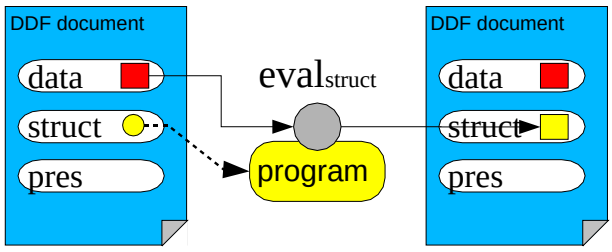

Figure 5. Generating new structure

Now we have a document with structure but no specific presentation. If however we merge this document with *another* DDF docu-

ment which contains program to make presentation from generic structure, we can then evaluate the new mappings and generate the appropriate presentation:

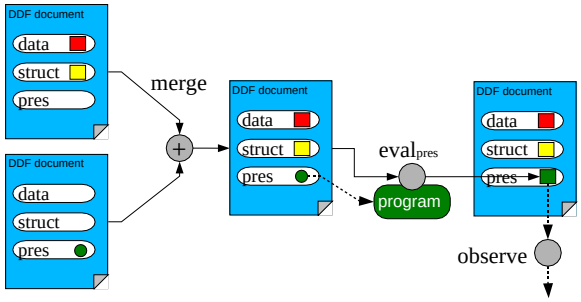

Figure 6. Generating new presentation

The result is still a legitimate DDF document, but one for which it is now meaningful to 'observe' the presentational part. A suitable observer function can extract this portion, possibly converting to a 'display format' (such as PDF) or directly rendering, to produce the resulting publication, shown at the start. Clearly different types of presentation can be generated from the same structure by merging a DDF document containing a different presentational mapping. Thus we can support an older style of HP business card:

Figure 7. An alternative presentation of the business card

4.1. Why an Encapsulated Form?

The DDF outlined is an encapsulation of data, structure, presentation and program as 'document'. Many *content management systems (CMS)* provide mechanisms where these are separated and controlled workflows apply necessary programs or services to generate resultant documents and components. These services can vary from arbitrary programs to XSLT transformations, which may even be buried within data files. These systems are often arranged as web-based service architectures, with a view to distribution and direct attachment to communications and consumers.

Whilst in these simple examples the differences in workflow and capability between DDF and simple use of XLST seem minor, as we'll discuss later, keeping program and data together will permit us to support advanced features such as incremental binding and merging. We particularly wanted to show that a document *could* be described and processed as a function, even with partial binding of data and evaluation of program. It still can be a component of a CMS, but it can also act as a standalone document.

5. DOCUMENT AS STRUCTURE

Developments in the late 1980s, especially on the Open Document Architecture (ODA) [7], emphasised benefits to be gained in flexibility and repurposing by separating the *logical structure* of a document and its *presentational style*. André et al.[1] give an excellent discussion on methods and benefits of such separation. (X)HTML and CSS have furthered this for Web-based use.

The logical structure is intended as the focus for most merging of documents and is constructed and modified from application data (which will of course include bindings of variable data). DDF does not fix what the syntax or semantics for this logical layer should be - we anticipate document engineers will choose a suitable 'standard' such as XHTML for principally report-type documents, or an application-slanted suitable format, like DocBook [18] for book material or GML [11] for documents with high geographical content, or of course a mix. The requirement is that for *that type of application simple generic merging at this layer is practical and useful.*

Structure acts as a buffer between application data and presentation, providing a canonical representation of the major groupings and sequences of the eventual message. By choosing to merge different documents providing mappings to and from the same structure, we gain flexibility. With an intermediate structure like this:

```
<doc>
  <h>Introduction</h>
  <p>An introductory paragraph which outlines the
  notion that structure can be used for different
  presentations</p>
  <h>Document as Structure</h>
  <p>Structure can act as the buffer between applica-
  tion data and presentation. Several different
  presentations can be created from the same struc-
  ture by use of an appropriate set of programs.</p>
</doc>
```

Figure 8. A fragment of document structure

we can merge different mappings to show either of the following presentation forms: a simple report or a high-level overview.

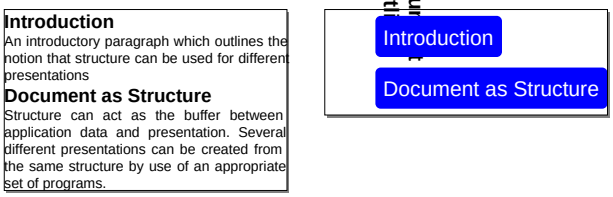

Figure 9. Two differing presentations from a single structure

Equally well, different types of application data can of course be mapped into the same structure. We could map portions of an XSLT source into this intermediate form to make documentation:

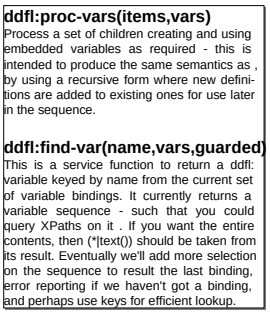

Figure 10. Reuse of the structure for documentation

Here we've selected `<xsl:function/>` elements, collected their names and the names of the embedded `<xsl:param/>` children and formed up a <h> structure with a function 'call' as its text value. A preceding documentation block is raided to make the description paragraph.

This use of structure increases flexibility dramatically: for example much of the DDF documentation takes XSLT or DDF source documents as application data and generates equivalent XHTML. The code examples in this paper are extracted automatically from various stages of processing the DDF documents, pretty-printed and converted to XHTML sections - the resulting SVG components are extracted similarly from the DDF results.

6. DOCUMENT AS FUNCTION

The crucial design choice with DDF was to consider a document as a *function* of some (possibly null) variable application data. The result of evaluating this function on a given instance of (application) data will itself be another DDF document, with modifications to various sections as a result of evaluation of appropriate program fragments.

The key is that 'program' sections for the creation of new content (logical structure or presentation) are embedded in the target spaces. We anticipate that such program is XSLT, though other languages might be possible (though there are *very strong* advantages to such languages being functional or at least *side-effect-free*).

Once we have some binding of the variable data for a document, we can evaluate it as a function. This might be performed by some interpreter, or via a compiler, but the end result should be the same. After the evaluation of `ddf:doc(variable-data)` the document will have internally copies of bound data, new pieces of structure and new presentation instructions arising from the data. With XSLT as the exmple programming language the next section will give examples.

6.1. XSLT Construction

We use XSLT to describe the functional construction of the document as a result of variability. Each of the structure and presentation spaces is considered a separate space as far as XSLT scope is concerned, with a default 'source' context, (data for structure and structure for presentation.) A very simplistic example might be:

```
<ddf:doc>
  <ddf:data/>
  <ddf:struct>
    <xsl:template match="name">
      <p>
        <xsl:value-of
        select="(first,last,qualification)"/>
      </p>
    </xsl:template>
  </ddf:struct>
  <ddf:pres>
    <xsl:template match="p">
      <fo:block   width="45"   font-family="Helvetica"
      font-size="4">
        <xsl:value-of select="."/>
      </fo:block>
    </xsl:template>
  </ddf:pres>
</ddf:doc>
```

Figure 11. A simple DDF document

which uses a simple <p> element as structure and maps a person's name and qualifications into it. If we bind a simple data record then evaluate both the data to structure and structure to presentation programs, the result is Figure 12, where a copy of the data record (for Owen Rees) has appeared in the data space, a paragraph has appeared in structure and a simple formatting instruction in the presentation.

```
<ddf:doc>
  <ddf:data>
    <name>
      <first>Owen</first>
      <last>Rees</last>
      <qualification>MA (Cantab)</qualification>
    </name>
  </ddf:data>
  <ddf:struct>
    <p>Owen Rees MA (Cantab)</p>
  </ddf:struct>
  <ddf:pres>
    <fo:block   width="45"   font-family="Helvetica"
    font-size="4">Owen Rees MA (Cantab)</fo:block>
  </ddf:pres>
</ddf:doc>
```

Figure 12. Bound DDF document

The final result is effectively an instruction for creating a text block, in this case using XSL-FO semantics. Further resolution of the document requires some *layout processor* which has appropriate knowledge of the semantics. We'll describe such a processor in the next section, but for the curious this is what the result looks like:

Owen Rees MA (Cantab)

Figure 13. The presentation from instructions of Figure 12

Currently DDF supports most of the more common XSLT 2.0 functional instructions, specifically those within templates and functions and global parameters and variables. There is some refinement needed on scoping of the program fragment: templates and applications in the default (unnamed) mode are local to the DDF space they exist within (i.e. `ddf:struct` ,`ddf:pres`) - named modes and all functions are currently global. (This avoids extensive re-writing or analysis of embedded XPaths for example.) Note that as the document is a function, then the 'correct' result will be produced regardless of the evaluation being carried out lazily or eagerly - which will be appropriate may depend upon circumstances - documents with significant conditionality may benefit from lazy evaluation, whereas ones used over a very large set of variable instances merit eager computation of invariant pieces.

6.2. Merging Documents

One of the original goals was to make it possible to merge DDF documents relatively easily and robustly to support both libraries and complex document workflows. The structural level is intended to be the main area for merging *what the document is trying to say* - as such schema-based checking could be used to avoid a 'report' being combined with a 'brochure' at this level. By choosing the 'programs' in DDF to be free of side-effects (i.e. evaluation of program components does not alter system state nor do they have internal state), we can begin to check the effect of combining two DDF documents in terms of *function* . This could be by comparing

function and template signatures (what they match, type of arguments and results)

At present, DDF documents have flat merging semantics for programs, exploiting order insensitivity of a set of XSL templates and functions - name clashes could occur and there is no defined precedence mechanism other than that inherent in any final processing engine. Simplistic 'include' directives can be embedded within a DDF document to indicate a required merge and there is rudimentary detection of possible clashes. With current examples these systems are adequate to support simple (well-controlled) libraries, such as those used for making this document. Future work will need to define fuller merging semantics, such as those of XSLT which define importation precedence.

7. DOCUMENT AS LAYOUT

The presentational layer is where final published forms (SVG, XSL-FO, PDF etc.) are built and stored from fragments, instructions and functions. 'Observers' extract from this layer.

As we've already shown, XSLT syntax and semantics are used as the principal description of the *construction* of the document with respect to data variability, in much the same way that an XSLT phase may be used to generate XSL-FO 'instructions'. But to make meaningful and useful documents we must describe what atomic pieces and compound structures will appear in final presentation, when the type and number of such components may be highly variable.

A major goal of DDF is to support an extensible set of types of layout with minimal disturbance to the rest of the processing machinery. Most of our experimentation has been with geometric styles of layout (catalogues, flyers, posters etc.) so we've chosen SVG as the canonical representation of grounded graphical pieces and assemblies.

PPML has shown the benefit in decoupling the definition of the graphical 'atom' from the compound assembly. We take a similar approach using a mainly top-down approach, describing layout as a nested series of functions until 'leaf nodes' are reached, with an extensible agent to interpret these functions. For example:

```
<ddfl:layout function="flow" direction="x">
  <svg:circle cx="3" cy="3" r="3" fill="red"/>
  <fo:block  max-width="45"  font-family="Helvetica"
  font-size="4">A little piece of text</fo:block>
</ddfl:layout>
```

Figure 14. A call for simple flow layout

is an instruction for a simple function that will flow its children in sequence in a given direction. The children in this case are a grounded SVG primitive (the circle) and an `<fo:block/>` with appropriate styling and the addition of a maximum width attribute. The default layout processor starts evaluating this 'function' by evaluating the children arguments into the space of SVG. The circle obviously stands for itself, but the processor has an extension (a mixture of XSLT and Java-based extensions) that can perform text-wrap roughly to the semantics of XSL-FO and return an SVG fragment, sized to the actual width and depth of the text.

With these children in a canonical SVG form, each with a determined width and height, the flow function can be evaluated to shift the text block right and then encapsulate the modified children to produce the compound SVG element of Figure 15, show in Figure 16 (the line boundary marks the edge of the compound result.)

```
<svg:svg x="0" y="0" width="40.0" height="6.0">
  <svg:circle  cx="3"  cy="3"  r="3"  fill="red"
  aspect="1" width="6" height="6" x="0"/>
  <svg:svg  max-width="45"  font-family="Helvetica"
  font-size="4" width="34" height="4.8" x="6.0">
    <svg:text x="0" y="3.97" width="34" height="4.8"
    line-offset="3.972">A  little  piece  of
    text</svg:text>
  </svg:svg>
</svg:svg>
```

Figure 15. Resulting SVG source after layout resolution

Figure 16. Compound graphic component of Figure 15

This principle operates hierarchically, exploiting SVG as the canonical form with principally rectangular 'molecules' and laying out based on these rectangular boundaries. Non-rectangular shapes may be supportable for denser packing, but in this case orthogonality of 'x' and 'y' disappears and in some cases layout becomes non-monotonic. As an example of hierarchical composition:

```
<ddfl:layout  function="flow"  direction="y"  spa-
cing="1">
  <ddfl:layout function="flow" direction="x">
    <svg:circle cx="3" cy="3" r="3" fill="red"/>
    <fo:block  max-width="45"  font-family="Helvetica"
    font-size="4">A little piece of text</fo:block>
  </ddfl:layout>
  <fo:block width="45" font-family="Helvetica" font-
  style="italic" font-size="2" hyphenate="yes">A very
  much longer piece of text that should be expected
  to wrap around showing that text blocks and columns
  are supported</fo:block>
</ddfl:layout>
```

Figure 17. Nested layout instructions

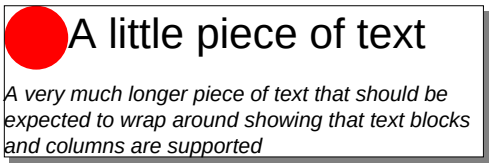

Figure 18. Nested compound layout

The tree of instructions and the resulting SVG tree show what's happening - usually a layout instruction makes an SVG encapsulation of its children, and a leaf node expands into an SVG child:

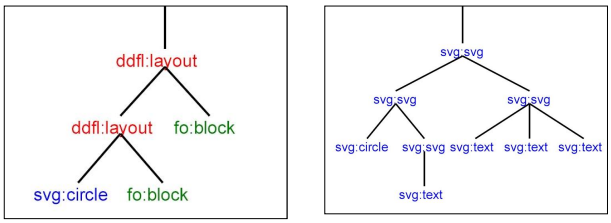

Figure 19. Composite layout tree and resulting SVG

This principal of nested functions connected through a canonical form (rectangular SVG molecules) can support a wide variety of

simple linear and non-linear layout forms, such as distribution and pagination. For example:

```
<xsl:variable                          name="colours"
select="('red','blue','yellow','green','magenta')"/>
<ddfl:layout   function="flow"   direction="x"   spa-
cing="3">
  <ddfl:layout function="paginate" height="18" encap-
  sulate="line">
    <xsl:for-each select="1 to 15">
      <ddfl:layout function="flow" direction="x" spa-
      cing="1">
        <fo:block    font-family="Helvetica"    font-
        size="2">
          <xsl:value-of select="."/>
        </fo:block>
        <svg:rect    width="9"    height="4"    rx="1"
        fill="{$colours[((current()  -  1)  mod  5)  +
        1]}"/>
      </ddfl:layout>
    </xsl:for-each>
  </ddfl:layout>
</ddfl:layout>
```

Figure 20. Pagination instruction over other instructions

surrounds a set of numbered rounded rectangles with a pagination function, assumed to flow vertically into a series of containers 18 units high - this function produces a set of 'pages' which can then be manipulated separately - in this case flowing them horizontally.

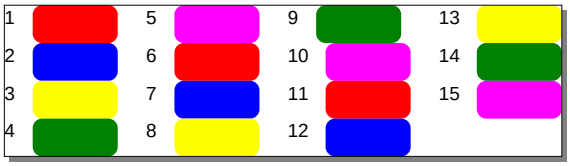

Figure 21. The non-linear paginated flow from Figure 20

7.1. Benefits & Limitations

This approach works well to build up compound graphical objects, using the tree's scoping to support naming of parts, scope of effect and so forth. By using a canonical form (SVG in this case) we can combine many different types of layout together simply, such as text-blocks inside pie-charts, without having to program these combinations especially. As this paper itself shows, a wide variety of layouts can be combined in a tree to produce complex results.

However using a tree and functional evaluation does have potential limitations that may require special features to ameliorate their effects. Reuse of components and linkage between components ('this text-block should be the same width as that image on the preceding page') requires relationships that cut across the tree - we've added an explicit variable and reference scheme to assist (see section 7.3). Layout functions that can 'overflow' such as flows and paginations may need 'fallback' options. So far this has been limited to processing based on explicit indicators of priority (e.g. in the case of overflow a paginator may remove pieces marked with 'low' prorities), but more general solutions are needed.

7.2. Constrained Layouts

For many situations, particularly in brochures and catalogues with high graphic content, layout can often be declared in terms of constraints between pieces. For conjunctive linear inequality constraints ('aligned left', 'above' etc.) we have several very powerful solvers available to process large sets of constraints relatively quickly. Such a system fits easily into the architecture of nested layout functions, with suitable means of defining the constraints and the pieces to which they attach.

In Figure 22 we define constraints on layout between six 'named' pieces (the five colours and 'title') by containing special form children that describe the edges of the constraint graph:

- 'red' is set above 'blue' and 'blue' above 'green' and all are aligned by their right edges.
- 'magenta' is placed above 'yellow' with right edges aligned.
- 'red' is set just left of 'magenta' and both have their tops aligned.
- The title is placed above and centred on 'magenta' ('magenta' is both the box and the text to its left)

The outcome is shown in Figure 23. (The children themselves are compounds which could of course be constrained layouts.)

```
<ddfl:layout function="linear-constrained">
  <fo:block    name="title"    font-family="Helvetica"
  font-size="3">A selection of COLOURS</fo:block>
  <xsl:for-each
  select="('red','blue','yellow','green','magenta')">
    <ddfl:layout  name="{.}"  function="flow"  direc-
    tion="x">
      <fo:block     font-family="Helvetica"     font-
      size="2">
        <xsl:value-of select="."/>
      </fo:block>
      <svg:rect    width="9"    height="4"    rx="1"
      fill="{.}"/>
    </ddfl:layout>
  </xsl:for-each>
  <ddfl:constraints layout="align(right) abut(above)"
  parts="red blue green"/>
  <ddfl:constraints layout="align(right) abut(below)"
  parts="yellow magenta"/>
  <ddfl:constraints             layout="align(centre)
  abut(above)" parts="title magenta"/>
  <ddfl:constraints   layout="align(top)   abut(left)"
  parts="red magenta"/>
</ddfl:layout>
```

Figure 22. Instructions for layout of a set of components based on linear constraints

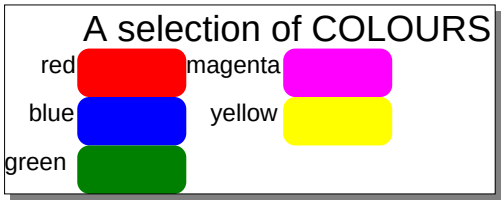

Figure 23. Resolution of the positional constraints between components of Figure 22

This approach has proved very powerful, combining scoped layout with flexibility of constraint solution. Pieces can be used as construction lines and discarded (circles for example can relate vertical and horizontal constraints.) It's possible to evaluate some constant constraint between successive members of a sequence by creating a constraint graph dynamically such as shown in Figure 24 and Figure 25.

```
<ddfl:layout    function="linear-constrained"    lay-
out="align(left,offset=off) align(top,offset=3)">
  <fo:block   name="caption"   font-family="Helvetica"
  font-size="4">A selection of COLOURS</fo:block>
  <xsl:for-each
  select="('red','blue','yellow','green','magenta')">
    <ddfl:layout          name="{.}"          func-
    tion="linear-constrained"   layout="align(center)
    align(middle)">
      <svg:rect    width="9"    height="4"    rx="1"
      fill="{.}"/>
      <fo:block font-family="Helvetica" font-size="2"
      fill="white" font-weight="bold">
        <xsl:value-of select="."/>
      </fo:block>
    </ddfl:layout>
  </xsl:for-each>
  <ddfl:constraints             layout="align(right)"
  parts="caption magenta"/>
</ddfl:layout>
```

Figure 24. Automatically generated constraints and additional unknown values

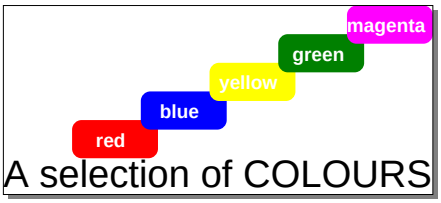

Figure 25. Constant constraints between successive members of a sequence - the result of Figure 24.

where an unknown constant offset (*off*) has been added between alignments of all the pieces (the title and all the colours) to create a cascade, and the last element ('magenta') has been constrained to align with the right of the caption, whose width of course depends upon the text and the font. ('red' is offset from the title by exactly the same distance as each pair of colours is separated.) Simple extensions generating constraint graphs automatically can be used to generate grids and tabular forms.

Similarly components such as rectangles, circles and images can be defined to be of indeterminate size and the size determined at layout time. (Images of course benefit from having a known aspect ratio and in most practical situations upper and lower limits on size based on pixellation.) Here is an example where without knowing the specific size of pictures we can still use a set of soluble constraints to complete layout:

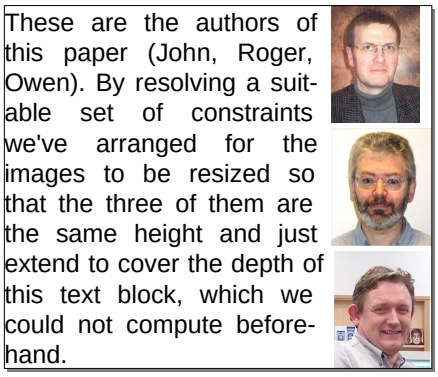

Figure 26. Variable-sized pictures in constrained layout.

7.3. Computed Graphical Components

The layout functions we've shown so far are of type*graphics*X *graphics->graphics.* But we can add 'leaf processors' for other types of computed graphics, constructing suitable instruction trees and making components like Figure 27.

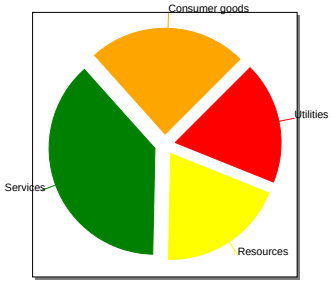

Figure 27. A computed chart as a graphical component.

7.4. Presentational Interdependency and Reuse: Variables.

Whilst many problems can be solved completely top-down, sometimes the result of layout of one part is required before another can be processed. A typical case is where a text-column has a width that is dependent upon how large some other pieces turn out to be - we need some information*post-rendering*of one component to control rendering of another. Such interdependency might be analysable and evaluation order determined, but we forsee circumstances where more explicit control is needed.

In many practical documents there are common substructures that are reused often, such as page backgrounds. We should be able to construct these only once and then copy as required.

To support both these needs (reuse and interdependency), the layout processor has a simple model for*presentational variables*, roughly akin to the XLST scoped variable model. (SVG has 'def' and 'use' which can perform similar function, but we wanted something that was generally applicable within our layout space, as well as being able to be processed through XPath expressions).

```
<ddfl:variable name="magenta">
  <ddfl:layout   function="linear-constrained"   lay-
  out="align(centre)    align(middle)    same(width)
  same(height)">
    <svg:rect rx="1" fill="magenta"/>
    <fo:block  font-family="Helvetica"  font-size="3"
    fill="white">a magenta block whose size depends
    on font and text</fo:block>
  </ddfl:layout>
</ddfl:variable>
<ddfl:layout    function="linear-constrained"    lay-
out="align(centre) abut(below)">
  <ddfl:copy-of select="$magenta"/>
  <fo:block   font-family="Helvetica"   font-size="2"
  text-align="justify">
    <ddfl:attribute                      name="width"
    select="$magenta/svg:svg/@width div 2"/>A portion
    of text that has been flowed into a block whose
    width should be half that of the 'magenta'
    object. This can only be determined after that
    object has been constructed</fo:block>
  <ddfl:copy-of select="$magenta"/>
</ddfl:layout>
```

Figure 28. Declaration and use of a presentational variable.

In Figure 28 we generate the item 'magenta' and assign it to a variable of the same name. We can extract its width to be applied to the text block (via an XPath expression which also performs some arithmetic), and then interpolate two copies of 'magenta' into the final result, laying out the three parts as a compound object:

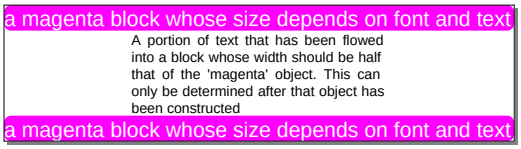

Figure 29. The result of evaluating Figure 28.

Use of such a mechanism both allows finer control by designers and also can be generated dynamically for higher-order forms of layout. In practice we've used it often to determine a width of text-blocks, such as in this article.

By bringing all these techniques together we can produce quite complex output. In the following multi-page retail flyer document (Figure 30), mappings for individual products are held in one DDF file, flyer background in another and the composition co-ordinated by a third. Two levels of customisation have been imposed: to the particular store and to a specific mix of products:

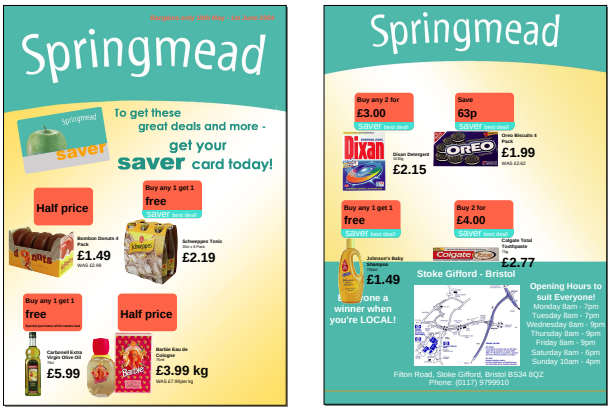

Figure 30. A complex customised document

8. COMPARABLE TECHNOLOGIES

There are a number of relevant and comparable recent technologies arising from developments in XML and its processing, XML-based presentation formats and extensive document *workflows* .

Aßmann [2] outlines an architectural style for active documents that accomodates merging, (invasive) composition, dynamic updating (transconsistency) and *staged* processing, though specific examples and implementations aren't apparent.

Villard and Layaïda [15] present an approach (*incXSLT*) to evaluating XSLT programs in an incremental fashion, specifically to carry out minimal rework on dynamic change in source data. It involves internal dependency analysis of the program to determine sets of re-evaluation rules, rather than trace logging of execution in some other schemes. Such analysis may be helpful in supporting incremental binding of variables in DDF.

Bes et al.[5][6] have explored coupling formatting systems together through sets of presentation operators allowing closer control of preferences between alternatives, fallback in case of failure and declaration of global policies. These are described in a *control document* which defines extraction, translation and combination of components between different formatters. In contrast, current DDF layout works with a canonical SVG form and strict hierarchical instruction scoping and explicit variable declarations, relying on naming discipline to avoid clashes.

Constrained SVG [4] is an extended form of SVG where various numerical attributes (positions, sizes etc.) on SVG components and assemblies can be declared as named variables and a series of linear constraints added relating these variables. An SVG renderer solves the network of constraints dynamically to determine the actual bindings for rendering the components. This can also occur repeatedly within animations for example. Later work [10] has 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. DDF's constrained layout concentrates on how to declare these relationships between *components* without having to construct the equations explicitly, though we use the same underlying constraint solver. We think that DDF layout could in some cases *generate* constrained SVG as output.

9. CURRENT STATUS; FUTURE PLANS

An early form of DDF and an associated set of layout semantics exists and is supported by an experimental implementation which is adequate to generate all the examples shown in this paper *and this paper itself*. [The paper 'source' is ~XHTML - the DDF document acts as a template mapping to modified XHTML as structure (references and figures interpolated, XML fragments pretty-printed) and thence to ACM style for layout and pagination.]

Construction semantics based on XSLT are reasonably stable. Layout is based principally on SVG as the canonical form of a set of hierarchical geometric components. Primitive components can be direct SVG fragments or wrapped text blocks based on slightly extended semantics of the XSL-FO `<fo:block/>` (which is converted to equivalent SVG.) A layout library (instructions in the XSLT sense, i.e. attributed nodes with children acting as arguments) exists covering simple functions such as distribution, encapsulation, simple flows and pagination. The processor also attaches to a linear constraint solver (Cassowary [3]) providing a richer set of possible layouts, and there is a simple Java connection to support other more esoteric forms of layout such as reordering packers. Output formats include subsets of SVG (1.2), HTML and PDF, generated from an SVG intermediate.

Apart from refinement of the DDF design and its implementations, there are other areas for development, including managing external references and supporting incremental binding. We will discuss these subjects briefly to show the possibilities within DDF.

9.1. Incremental Binding

Our examples shown in this paper have used fully bound data: the arguments to the document-as-function have been fully instantiated. In some cases however the data may be only *partly* bound. For example a sequence of daily records might end in a continuation (Mon, Tues,...). In this case it would be attractive to be able to consume and process this section of the data, making a new DDF document, whilst still able to consume more data (Wed,Thurs) at a subsequent stage using that result document. In such cases we want the evaluation of 'document-as-function' to

yield something of similar type - a function that can be reapplied on some other variable data. The workflow of our earlier diagram now should look like Figure 31:

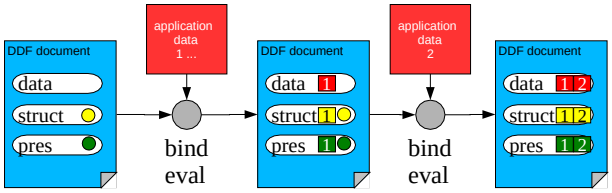

Figure 31. Generating new presentation

where the indication of a continuation ('...') on the original data is detected and results in *program elements* being generated in the resulting DDF document as well as structure and presentation derived from the presented data. What has been evaluated is

```
ddf:doc(variable-data...)
```

whose type may still be a function.

Then at a subsequent point the DDF document (which contains all necessary program) can be subjected to additional binding and re-evaluation, effectively determining:

```
(ddf:doc(variable-data...))(more-data)
```

Similarly a set of alternatives within a document may be dependent upon a data value or other control parameter (page sizing, internationalisation etc.) which is known not to be yet defined. Program constructions can be passed through into the results such that correct choice may be made at a later choice point.

The advantage of exploring such partial binding and 'eager' evaluation, rather than using a lazy evaluation scheme when results are needed in final printed form, is that with suitable implementations re-use can be optimised. For example a partial binding of a promotional flyer with products to be promoted can be processed once to produce a *template* that could be further customised, by addition of simple customer details (name, address ..) over many thousands of customers but with the the template construction only performed once. Of course it is a research issue on what operations in construction, and especially layout, can be performed partially and incrementally re-performed. We have started some very preliminary experimentation in supporting these types of partial binding, though we expect considerable use of designer-specified 'hints' for early examples.

9.2. External References

One issue that appeared in early prototypes was pieces of application data acting as references to external objects, most notably images. Final documents and intermediates may be in different places from sections of the original application data, making relative references exceptionally fragile. We needed to add some mechanism that would track such references effectively through arbitrary XSLT program (especially forming temporary trees where ancestry paths back to source routes are normally broken). An early attempt used XML's namespace system to achieve this, but a more robust and explict system of context maps has been developed for future implementations.

10. ACKNOWLEDGEMENTS

We're grateful to our colleagues Fabio Giannetti, Royston Sellman and Tony Wiley for robust discussion on the purpose and direction of DDF, to Xiaofan Lin and Brian Atkins for exercising the extensibility of layout and to Peter Woods for help in developing the DDF build environment. The reviewers of this paper made helpful suggestions which have been incorporated in this version.

11. REFERENCES

[1] André, J., Furuta, R. and Quint, V. *Structured Documents* . Cambridge University Press. 1989.

[2] Aßmann, U. Architectural styles for active documents . *Science of Computer Programming* . Vol56, 79-98. Elsevier. 2005.

[3] Badros, G., Borning, A. and Stuckey, P. The Cassowary linear arithmetic constraint solving algorithm .*ACM Transactions on Computer-Human Interaction (TOCHI)* . Vol8 (4), 267-306. 2001.

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

[5] Bes, F. and Roisin, C. A Presentation Language for Controlling the Formatting Process in Multimedia Documents . In *Proceedings of the 2002 ACM symposium on Document engineering* . 2002.

[6] Boulmaiz, F., Roisin, C. and Bes, F. Improved Formatting Documents by Coupling Formatting Systems . In *Proceedings of the 2003 ACM symposium on Document engineering* . 2003.

[7] ISO, International Standards Organisation *Open Document Architecture* . http://www.iso.org/. 1994.

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

[9] Lamport, L. *LaTeX User's Guide and Document Reference Manual* . Addison-Wesley, Reading, MA. 1986.

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

[11] Open Geospatial Consortium *Geography Markup Language (GML 3.0)* . http://www.opengeospatial.org/. 2001.

[12] PODi, Print On Demand Initiative *Print On Demand Initiative* .http://www.podi.org. 2005.

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

[14] PODi, Print On Demand Initiative *Personalized Print Markup Language - Templates (PPML/T)* . http://www.podi.org. 2002.

[15] Villard, L. and Layaïda, N. An Incremental XSLT Transformation Processor for XML Document Manipulation . In *Proc. 11th World Wide Web Conference, Honolulu* . 2002.

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

[17] W3C, World Wide Web Consortium *XSL Transformations (XSLT) Version 2.0* . http://www.w3.org/TR/xslt20/. 2005.

[18] Walsh, N. and Muellner, L. *DocBook: The Definitive Guide* . O'Reilly & Associates. 1999.