Laying out the future of final-form digital documents

David F. Brailsford
School of Computer Science and IT, University of Nottingham,
Jubilee Campus, NOTTINGHAM NG8 1BB UK

ABSTRACT

It is just over 20 years since Adobe's PostScript opened a new era in digital documents. PostScript allows most details of rendering to be hidden within the imaging device itself, while providing a rich set of primitives enabling document engineers to think of final-form rendering as being just a sophisticated exercise in computer graphics. The refinement of the PostScript model into PDF has been amazingly successful in creating a near-universal interchange format for complex and graphically rich digital documents but the PDF format itself is neither easy to create nor to amend. In the meantime a whole new world of digital documents has sprung up centred around XML-based technologies. The most widespread example is XHTML (with optional CSS styling) but more recently we have seen Scalable Vector Graphics (SVG) emerge as an XML-based, low-level, rendering language with PostScript-compatible rendering semantics. This paper surveys graphically-rich final-form rendering technologies and asks how flexible they can be in allowing adjustments to be made to final appearance without the need for regenerating a whole page or an entire document. Particular attention is focused on the relative merits of SVG and PDF in this regard and on the desirability, in any document layout language, of being able to manipulate the graphic properties of document components parametrically, and at a level of granularity smaller than an entire page.

Keywords: PostScript, PDF, SVG, PPML, DDF, COG model, variable data printing

1. INTRODUCTION

The present generation of imaging devices makes it possible to create textually and graphically rich pages in a way that would have been unimaginable 30 years ago. Formats such as PDF give us *page independence* , which is to say that the graphic state of each page and the resources (e.g. fonts) that it needs are known and are guaranteed not to suffer any side effects from previous pages that have already been rendered. However, the story is very different, within a page, when it comes to considering the graphical independence of sub-page components. It can be extraordinarily difficult, for example, to extract, cleanly, a diagram, or an illustration, or even a paragraph of text from a PDF page. Although Adobe Acrobat tries to provide just such a facility it is very common to extract not only the desired object but a collection of unwanted material as well.

It is frustrating to reflect that design packages such as Adobe's *Illustrator* and *InDesign* do have some internal knowledge of the 'objects' they are manipulating and yet, so often, all of this knowledge is lost in an amorphous final-form page once PostScript or PDF is exported. This means that any alterations to fonts, fill colours or page layout requires the regeneration of a whole new page.

A new generation of colour laser printers and presses (Hewlett-Packard's *Indigo* for example) make possible high-speed 'variable data' printing. Here a page may consist of advertising material that is specific to a particular circulation area, or even to each individual customer. The ability to cut and paste cleanly-specified 'objects' from near-final-form page descriptions would make it much easier to re-use document components. Another benefit would be that most objects on a page could be kept constant, and efficiently cached, during a long print run while also allowing the insertion of different late-bound, variable-data, components on each separate page.

In the remainder of this paper we survey the reasons for the lack of encapsulated graphic objects in the code that is typically output from page-design software. We then show how such notions can be accommodated in PDF[1] using a little-known data structure present in that format. In order to show that the ideas put forward will generalise beyond PDF we examine how they can be realised in the XML-based Scalable Vector Graphics (SVG) with, arguably, a rather better facility than in PDF for parameterising the objects. SVG is used, at the moment, for "vector graphic inserts" inside Web pages but it seems possible that it might eventually develop into a page description language in its own right.

1.1. Bitmaps, SVG and XML.

It is natural that bitmap images for photographs etc. should feature prominently in our considerations as possible page objects but we shall assume that such objects are embedded in a more abstract description of the page. In other words, we exclude from consideration those pages which have, in their entirety, already been fully rendered to bitmap form. Such a low-level format (useful though it is for rendering 'problem' pages externally before sending them to a printer) is so final and immutable that it will generally have no abstractions allowing the bitmap to be sub-divided into objects at the sub-page level.

Much of what follows will consist of a survey of PostScript, and then PDF, as vehicles for the model to be described. However, in later sections of the paper we survey some possible advantages that might accrue when SVG develops full page-description capabilities. It will also become apparent that languages such as SVG and PPML usher in a new era where XML-based notations are used not just for highly abstract views of a document but also at a very detailed level for controlling page appearance and layout.

The ability for a page representation, in an XML notation, to be re-shaped and transformed by an XSLT script, followed by re-interpretation in an XML-based page layout package, means that much can be done in terms of supplying late-bound data for pages. The final sections of the paper assesses the importance of these new possibilities for creating personalized documents and for 'variable data' printing.

2. HISTORICAL

If we look back to the era of the early 1980s, just 25 years ago, it is now hard to recall just how difficult it was to produce complex, graphically rich, pages. During the years from 1955 to 1985 hot metal typesetting had gradually given way to text produced from optical phototypesetters or from third-generation Cathode Ray Tube devices like the Linotronic 202. The graphic capabilities of this generation of machine for rendering anything other than text, were either limited or non-existent. When printing plates came to be made graphic elements had to be laboriously prepared with pens, knives, and adhesive toned film possibly followed by airbrushing before being 'stripped in'.

Once the new generation of laser-driven printing devices became available there was an understandable urge, for a time, to exploit the ability for manipulating all graphic elements — from half-tones to character outlines — at the bitmap level. But in late 1984 and early 1985 all of that changed once PostScript came on the scene. Here was a language that was not only a *tour de force* of two-dimensional computer graphics for character shapes, vectors and for greyscale and colour bitmaps, but also a sophisticated interpreted language of LISP-like flexibility which also provided associatively-indexed dictionaries and a stack-based procedural model

By late 1985 the take-up of PostScript by graphics-arts professionals was enthusiastic and near total. Let us now imagine that we are back in 1985 and that we want to design a logo† for the organisation sponsoring the current conference. This hypothetical logo is shown in Figure 1.

This design utilises no fewer than eight separate PostScript layers. The first two of these set up the white version of the word "Electronic" and then outline it in black. The third layer implements the preliminary laying down of

† adapted from material in reference 2,

the grey bar, which will initially obscure the type directly beneath it. Layer 4 uses the grey bar of the previous layer as a clipping box and within that clipping rectangle it re-sets “Electronic” in a pale shade of grey (0.7 on the PostScript scale). Layer 5 shows the pale grey type through the clipping rectangle boundary while layer 6 draws a black line around the grey box. In layer 7 the word “IMAGING” is force justified in black with bullets between each letter. Finally, in the topmost layer, the word “IMAGING” is set again but this time with an offset one point up and two points to the left of the previous black version of the text to act as a highlight.

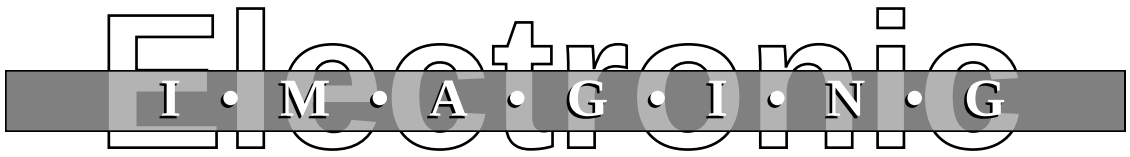

Figure 1: An 8-layer logo created using 774 bytes of PostScript

It is instructive to follow the analysis by Simon Fuller[2] of what would be required to create such a logo by traditional methods:

> “To get such an image traditionally would cost a lot in film stripping and headaches. For one thing, it would be extremely difficult to draw the bar with a technical pen and to get such sharp corners, Secondly, without film stripping it would be enormously difficult to butt the line to adhesive toned film, let alone get a clean join with the lighter portions of type cut into the bar — a join free of Moiré patterns, if you will. The only other alternative would be to airbrush the bar, risking inconsistencies in tonal depth, as well as running into as highly difficult masking session on the type. In either case, due to the halftone, you would be unable to reduce the artwork by fifty percent or any other percent, as all the dots would probably join together on the line shot. In PostScript, however, all you need to do to get a perfectly reduced image is to put one line at the head of the file : 0.5 0.5 scale

But even more telling is the fact that this logo is just a small PostScript program of some 774 bytes. It could be turned into a single compact PostScript procedure because just about every aspect of it is parameterisable: the texts and typefaces for the two words in the logo; the size and shade of the grey-filled clipping box; the overall placement of the logo and the precise shade of grey of the text, within the clip rectangle, which will optimise the translucency illusion. And yet, despite all of this, it remains the case that only hand-written and hand-tuned PostScript would ever deliver code that is similarly procedural in its approach. Indeed, Fuller laments that an EPS file for a logo very similar to the one shown (which, as we have shown, needs no more than 800 bytes at most) would occupy about 36K if produced from Adobe Illustrator.

In what follows we analyse the historical reasons why page design packages choose to output low-level PostScript with no attempt to create identifiable objects at the sub-page level. This, in turn, leads to an analysis of how faster computers, faster rendering devices, and new page description languages such as PDF and SVG (both of which are ‘interpreted data structures’ rather than full-blown programming languages) now make it possible to introduce a form of modularity and configurability for graphic components on a finally rendered page.

2.1. ‘Compiling’ PostScript

The Apple LaserWriter, launched in 1985, was the first ever PostScript imaging device. With its Motorola 68000 processor, its 1.5 Mbyte of RAM and with the PostScript interpreter present in 0.5 Mbyte of masked ROM it was far more powerful than the Macintosh computer that was intended to drive it. Even so, speed (or the lack of it) was an issue from the very moment that the device was launched. Steve Jobs (then, as now, the Apple CEO) wanted to have some documents that could be printed out, on stage, at the LaserWriter launch event. In those days, prior to the release of applications such as Pagemaker, Adobe had to create hand-coded PostScript to print

out a number of documents, the most visually and politically impressive of which was to be the IRS tax form. The co-inventor of PostScript, John Warnock, created elegant PostScript subroutines to print out these forms but the problem was that the interpretive burden for all the language constructs used caused it to take two and a half minutes for the LaserWriter to process the code and print the page.

Steve Jobs protested that keeping the audience's attention for two and a half minutes, however wonderful the final printed result, was going to stretch even his legendary presentation skills to the utmost; a dramatic speed-up was essential. Warnock then revisited the code and using PostScript's ability to re-define system operators, managed to get the program to 'compile itself'. In doing so it replaced procedure calls with replicated in-line procedure-body code (i.e. with all arguments fully bound into the program text) and all the for loops were unrolled and linearised. The net result was a much longer, static, final-form page description of the IRS tax form but one which now printed in 12.5 seconds!

In essence the above tricks are used today by Adobe Distiller when converting arbitrary PostScript into PDF. Although clever compression techniques mean that a PDF file is usually rather smaller than its PostScript counterpart in cases such as this — where a tiny, procedural, PostScript program is distilled — one can easily find that the file size increases dramatically, for all the reasons just outlined. Thus the PDF version of the 774 byte PDF logo is ten times larger at 7755 bytes.

2.2. PostScript generated from front-end applications

Shortly after the release of PostScript, page layout programs such as Pagemaker and the Adobe Illustrator vector graphics package were able to generate PostScript from interactively created page designs. However, for ease of code generation and for reasons of speed when interpreting the PostScript output, such generated code tends to use PostScript as a kind of macro-assembler language rather than exploiting its powerful procedural capabilities. That is to say the generated code is a static page description very close to a fully-bound and final form. Apart from a few housekeeping procedures there is no attempt to generate procedural code with parameters that could be varied independently of the front-end application.

2.3. Page independence in page description languages

From the very outset PostScript did not have page independence although, interestingly, Xerox's Interpress (in many ways the forerunner of PostScript) did have. The graphic state of the PostScript interpreter can be saved and restored at any point with explicit gsave and grestore operations but these were not performed automatically at page boundaries. For this reason if a PostScript previewer is to show you just page 80 (say) of a document, then it needs to interpret all of the previous 79 pages in order to be sure that the full graphic state for page 80 has been set up correctly.

When PDF was being designed it was clear that users would wish to insert, delete and extract pages from a PDF file. Clearly, then, page independence needed to be established within PDF and so, when the Distiller program compiles PostScript into PDF it has to ensure that each page has the resources it needs and is independent of all others

3. PAGE CREATION USING INDEPENDENT GRAPHIC OBJECTS

What we now need to develop is a model for graphic objects on a page that is general enough to be applicable in any sufficiently powerful rendering language. We shall call this the **C**omponent **O**bject **G**raphic (COG) model.

The ideal situation would be a world in which front-end applications generated all pages as sequences of such COG objects, if only because subsequent extraction of a chosen COG (e.g. a paragraph, a photograph or a line diagram) from a finished page, for re-use or replacement, then becomes much easier than is presently the case.

Some of the desirable features of such a COG model can be listed as follows:

- Each COG to be independent of all others on the page
- Each COG must declare the resources it uses
- COGs must not presume any pre-existing graphic state
- All rendering within a COG must use co-ordinates relative to a local (0,0) origin
- COGs will need external positioning information to place them on a page

Figure 2 shows a notional set of software tools for creating and manipulating pages within the COG model and the purpose of the various tools can be summarised as follows: *COG Manipulator* is an interactive tool that reads in pre-prepared COGs (e.g. from Encapsulator, Extractor or Creator) and allows them to be placed and fitted on a page, prior to saving the completed COG page.

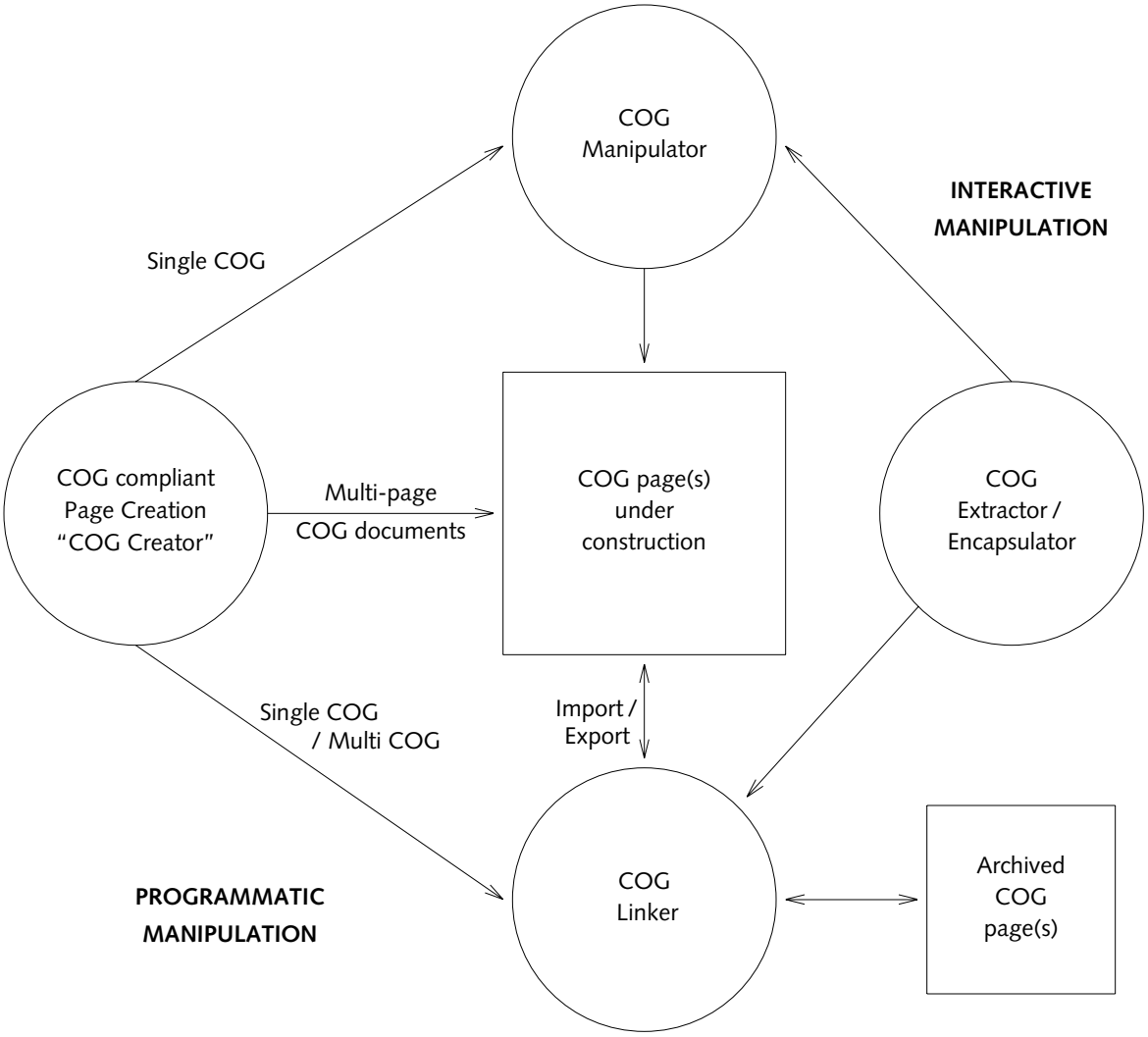

Figure 2: Assembling a page from COG components

The *COG Encapsulator* puts a COG PDF wrapper around free-standing images (e.g TIFF or JPEG) or a COG wrapper around 'clean' PDF objects exported from Illustrator, Photoshop etc.

The *COG Extractor* allows for cropping out of material from arbitrary final-form pages and for saving that material as a COG. Great care needs to be taken in determining the graphic state of this extracted material and discovering the resources (e.g. fonts) that it uses.

COG Creator is a generic term for any page layout software capable of creating individual COGs, or multi-page sequences of COGs.

COG Linker is software capable of saving a whole page of COGs in a standardised form suitable for archiving or for re-loading into a COG page-makeup application at some later time. The idea here is that the COGs themselves may be in any of the popular formats such as PostScript, TIFF, JPEG, PDF or SVG but the declarative superstructure specifying the resources needed by the various COGs, and their placement on the page, will be XML based. A later section describes how an adapted form of PPML is suitable for this purpose.

3.1. Implementation of COGs

The implementation of COGs in a PDF framework is fully described in the paper by Bagley Brailsford and Hardy[3] . The COG PDF implementation was made possible by the existence of two little-known data structures within PDF itself. The first of these is the option of having PDF page content be described as an array of content streams, rather than a single amorphous content stream for the whole page. The second is the existence of data structures known as *Form XObject* s which can be components in just such an array of streams.

A Form XObject is defined in PDF as just one example of a so-called COS Stream[1]. Its header dictionary contains information of the sort shown below

```
<< /Type /XObject
/Subtype /Form
/FormType 1
/BBox [0 0 1000 1000]
/Matrix [1 0 0 1 0 0]
/Resources <<
/ProcSet [/PDF]
>>
…
>>
```

Some of the dictionary keys are obvious: the `Type` and Subtype keys define it as a Form XObject, the `BBox` sets the bounding box, and the embedded `Resources` dictionary specifies the resources needed. Form XObjects have the useful property that when they are executed they make no change to the graphic state; their execution is implicitly wrapped up between graphical save and restore operations.

We now see that a COG PDF page consists of nothing more than an array of Form Xobject COGs, interleaved with spacer objects that position the COGs on the page.

3.2. PDF COG tools

All the tools shown in Figure 2 are now complete, or nearly so, for PDF COGs. Extractor, Encapsulator and Manipulator are all implemented as plugins for Adobe Acrobat 7.0; COG Linker is a standalone XML application to be described later. A proof of concept has been completed for COG Creator via the development of a post-processor for output from the UNIX device-independent version of *troff*. This post-processor is capable of creating an entire document such as a report, or an academic paper, as a sequence of COGs and a sample page from just such a paper, created in PDF-COG form, is shown in Figure 3. By viewing such a document with the COG Manipulator plugin it is very clear that individual items such as the table, the mathematical formula or the line diagram could readily be extracted for use elsewhere.

However, this happy state of affairs does not apply if COG Manipulator is asked to extract material from a PDF page that has not been prepared with COG-compatible software. The difficulties encountered prompted the development of COG Extractor and they merit a sub-section of their own because they highlight the problems faced when a page layout language fails to encourage an 'object oriented' view of page components.

3.3. Problems encountered in PDF-COG Extractor

To understand the problems in extracting material from amorphous and unstructured pages, let us take PostScript and PDF as an example. We have already noted the distinct performance advantages if page creation software generates PostScript which is close to final form and which has no procedural abstractions corresponding to objects on the page. Furthermore, there is nothing at all in PostScript to require that material on a page be laid down in anything like a logical order — the approach has always been that if the final page looks correct then the rendering order is irrelevant.

Once the PostScript is distilled to PDF, optimisations are effected to keep file size down using various compression techniques and by subsetting embedded fonts to contain only the characters that are actually used. However, it is not generally safe for Adobe Distiller to alter the rendering order of the text and graphic items it encounters from that established in the original PostScript.

The quality of the PDF, therefore, with regard to the clean extraction of material is determined very much by the quality of the original PostScript From Acrobat 4.0 onwards the *Touch Up* facility has been enhanced to allow graphic objects, as well as text objects, to be selected and moved around the page. The aim is to allow adjustments to the detailed layout of a page without having to revert to the software that initially created the PDF. The problem is that the underlying lack of a proper object model means that Touch Up tends to select what it knows it can safely manipulate rather than what the user might actually want.

COG Extractor works very hard to try and identify which objects lie within the selected area and to select the minimum possible material from the page.

3.4. COG Linker

The flexibility provided by COGs can only be fully exploited if COGs from multiple, disparate, sources can be combined programmatically to form a composite document. Although the COGs themselves have to be in some specific format, such as PDF or SVG, it helps greatly if the script that places them, and which specifies their properties, can be in a format that is amenable to processing by a variety of other programs. In effect we want is something similar to a program language *linker* that takes COGs from a variety of sources and links them together to form a single document. In a program language linker object modules are created using a local base address of zero; the linker places these modules and alters their addresses so that they sit within a new, merged, address space. In a similar manner our COGs all have a local (0,0) origin but the COG linker needs to know which COGs appear on which page, and at which positions.

Rather than developing our own linking language we investigated the XML-based *Personalized Print Markup Language* (PPML) developed by a consortium of digital press manufacturers (PODi)[4] . PPML seemed promising not the least because it is a standardized method for describing personalized or 'variable data' documents [5] . in such a way that blocks of material can be Raster Image Processed (RIPped) just once and then cached in a RIPped form for reuse. PPML has XML elements such as `<OBJECT>`, `<REUSABLE_OBJECT>` and `<OCCURRENCE_LIST>` to allow for object specification and re-use, together with optional `<SUPPLIED_RESOURCES>`, `<REQUIRED_RESOURCES>` and `<REQUIRED_RESOURCE_REF>` elements. It was disappointing to find that PPML's inheritance and scoping rules for resources could be restricted no further than the entire page (though, to be fair, this is a very plausible unit of interest for digital press manufacturers). Fortunately, it has proved relatively easy to adapt PPML into a form we have called PaC (**P**PML **A**dapted for **C**OGS) so that we now have a language which fully satisfies our need for a COG link-editing language[6] .

Figure 1: Processing (LA)TEX to PDF

Turning to the question of the size of the Department of Computer Science in a hypothetical UK university, in 1989, the following table will be of interest:

Course	UG or PG	Full Time Equivalent Students							
		Home + EC				Overseas			
		On Course		Graduated 1988		On Course		Graduated 1988	
		M	F	M	F	M	F	M	F
Computer Science	UG	63	3	15	3	10	1	1	0
Computer Science/Statistics	UG	1	0.5	0.5	0.5	0	0	0	0
Mathematics/Computer Science	UG	3.5	0	1	0	0.5	1	0.5	0.5
Computer Science/Cybernetics	UG	10.5	0.5	4.5	0.5	0	0	0	0

Sceptics may conjecture that the above table does not show real student numbers but that they have, instead, been generated from the formula:

$$u(x,y) = \sum_{n=1}^{\infty} \left\{ \left[\frac{2}{a} \int_0^a f(x') \sin \frac{n\pi x'}{a} dx' \right] \frac{\sin \frac{n\pi x}{a} \sinh \frac{n\pi (b-y)}{a}}{\sinh \frac{n\pi b}{a}} \right\}$$

References

1. N. Abramson, *Introduction to Information Theory and Coding*, McGraw Hill, 1963.
2. Arnold Aardvark, "Mutual Channel Capacity for Avaricious Aardvarks," *Journal of Irreproducible Results*, vol. 3, no. 2, pp. 61–63, May 1978.
3. Bertie Bearskin and Carole Coypu, *Our unpublished theories on Mutual Information*, January 2001. http://www.bearskin.com/~bertie/paper.pdf.
4. Steve G. Probets and David F. Brailsford, "Substituting outline fonts for bitmap fonts in archived PDF files," *Software — Practice and Experience*, 2003. (to appear)

Figure 3: A sample page from an academic paper rendered as COGs. COG boundaries are highlighted

Figure 4 shows an extract from a PaC script corresponding to a page of a scientific paper. The extract shows the declaration of one of the resources (the Times Roman font), a reusable COG object for the page header "INTERNATIONAL JOURNAL OF QUANTUM CHEMISTRY" which, in turn, contains a statement of the font(s) it needs followed by the PDF for this COG and, in an occurrence list, the unique COG identifier by which this object will be referenced. Finally, there is a page layout section where this particular COG is placed via a `<MARK>` operation and called out via an `<OCCURRENCE_REF>`.

3.5. Recent developments

The development of the COG tools already described has led to the start of the *COG Scrapbook* [7] project in which visitors to museums, or other buildings of interest, can download pre-existing COGs, or create their own, for subsequent placement in a montage that records their visit. This project makes heavy use of MS-Windows versions of COG Encapsulator (for wrapping JPEG photographs from digital cameras as PDF COGs) together with COG manipulator for creating the montage.

4. SVG AS A PAGE DESCRIPTION LANGUAGE AND A FORMAT FOR COGS

In 1998 the World Wide Web Consortium (W3C), set up a working group to draw up draft proposals for Scalable Vector Graphics (SVG). The perceived need at the time was for improved rendering, in Web browsers, of material such as line diagrams, schematics, and maps. For the most part popular browsers such as Internet Explorer need to install an SVG plug-in supplied by Adobe Systems Inc. but recently an Open Source SVG browser called *Squiggle* has been released by the Apache project[8] . SVG is an application of XML and this allows it to participate, via namespaces, in general XML documents and to be transformed by XML parsers, by ECMAscript inside Web browsers or by scripts written in **X**ML **S**tylesheet **L**anguage **T**ransformations (XSLT).

The semantics of SVG's graphics model are similar to those of PostScript and PDF. In common with these two languages, SVG combines graphical sophistication with the ability to place text strings accurately. It is tempting to imagine, therefore, that an extension of SVG to take on board the notion of pages, via the proposed *pagesets* [9] in SVG 1.2, would be relatively straightforward to bring about. Sadly, this is not the case. A paper by Danilo and Fujiisawa [10] surveys just some of the features that are missing such as filters, ICC colour spaces and sophisticated font handling and embedding. These, and a host of other necessary features, are already present in PDF.

At the time of writing the release of SVG 1.2 does not seem imminent and, for the moment at least, Acrobat's increasing ability to render SVG (used in *Photoshop Album*) coupled with an ever-expanding use of XML notation inside PDF itself might seem the best way forward for creating an SVG-friendly Page Description Language (PDL).

However, it seems certain, one way or another, that SVG will eventually emerge as a viable PDL and it is interesting, therefore, to see what it might offer in the area of variable data documents. Much of SVG's potential arises from the fact that it is an XML application and, like all XML applications, it is therefore a tree structure that can be suitably interpreted. This is a sufficiently important point to warrant an explanatory sub-section.

4.1. The nature of XML

One of the most common misconceptions about XML is the failure to realise that it is a *metasyntax* in which a potentially infinite number of XML 'languages' can be expressed. The standardised features of XML are such things as the 'pointy bracket' notation for the defined tags (more properly called *elements*) , the specification of the various character sets that can be used, the syntax of the *attributes* that can be attached to elements and so on. Perhaps most important of all is that the specification of a set of XML elements, via a Document Type Definition or an XML Schema, will set out the allowed ways in which the elements may form sequences and the ways in which the elements can nest inside one another. All in all the most general data structure that an XML document defines is a tree, and one whose nodes are the various XML elements which occur in a particular document. These nodes themselves can optionally be decorated with attributes.

```
<?xml version="1.0" encoding="utf-8" ?>
<PPML xmlns="http://www.podi.org/ppml/ppml210.xsd"><DOCUMENT_SET><DOCUMENT>
<SUPPLIED_RESOURCES>
<SUPPLIED_RESOURCE Name="840fed1a-70b3-11da-9d9f-c82689080901"
                   Format="pdf/font" Type="Font">
<INTERNAL_DATA><PDF><DICT DEFINE="1/0">
<NAME KEY="Subtype" VAL="Type1"></NAME>
<NAME KEY="Name" VAL="R"></NAME>
<NAME KEY="BaseFont" VAL="Times-Roman"></NAME>
<NAME KEY="Encoding" VAL="MacRomanEncoding"></NAME>
<NAME KEY="Type" VAL="Font"></NAME>
<NAME KEY="UUID" VAL="840fed1a-70b3-11da-9d9f-c82689080901"></NAME>
</SUPPLIED_RESOURCE>
...
</SUPPLIED_RESOURCES>
<REUSABLE_OBJECT>
<REQUIRED_RESOURCES>
<FONT FontName="840fed1a-70b3-11da-9d9f-c82689080901" Format="pdf/font"
                   ResourceName="R"></FONT>
</REQUIRED_RESOURCES>
<OBJECT Position="0 0">
<SOURCE Dimensions="306.579987 8.649994" Format="application/cog">
<INTERNAL_DATA>
q
0.125000 0 0 0.125000 0 0 cm
BT
/R 1 Tf
64.000000 0 0 64.000000 0.000000 14.160000 Tm
(INTERNATIONAL JOURNAL OF QUANTUM CHEMISTRY, VOL. V,) Tj
80.000000 0 0 80.000000 1907.000000 14.160000 Tm ( 657) Tj
80.000000 0 0 80.000000 2053.000000 14.160000 Tm (?) Tj
80.000000 0 0 80.000000 2099.000000 14.160000 Tm (688  71) ) Tj
ET
Q
</INTERNAL_DATA>
</SOURCE>
</OBJECT>
<OCCURRENCE_LIST>
<OCCURRENCE Name="Cogc38dc814-707b-11da-9fdd-bce2766de247"></OCCURRENCE>
</OCCURRENCE_LIST>
</REUSABLE_OBJECT>
...
...
<PAGE Dimensions="595.000000 841.000000">
<MARK Position="72.000000 786.119995">
<OCCURRENCE_REF Ref="Cogc38dc814-707b-11da-9fdd-bce2766de247">
</OCCURRENCE_REF>
</MARK>
...
</PAGE> </DOCUMENT> </DOCUMENT_SET>
</PPML>
```

Figure 4: Portion of a PaC script for a scientific paper containing embedded PDF COGs

4.2. SVG COGs

The ability of SVG to implement the COG model has been investigated recently by Macdonald, Brailsford and Bagley[11] . It turns out that COG model can indeed be implemented in SVG but the manner in which this is achieved is rather different than for PDF. The first point to note, in achieving the encapsulation that COGs require, is that properties at a given point in the SVG tree tend to be inherited from higher layers in the tree. To some extent this can be overcome by specifying graphic state properties explicitly in a grouping element <g>. Thus if we typeset a simple 'Hello World' example, in Times Roman, in SVG, the coding might be:

```
<text x="100" y="100" font-family="Times"
font-size="12pt">
Hello World
</text>
```

but if we enclose this within a grouping we can enforce arbitrary scalings and translations:

```
<g transform="scale(2)">
<text x="100" y="100" font-family="Times" font-size="12pt">
Hello World
</text>
</g>
```

Here the text will be appear twice as big as before and the positioning coordinates will also double, so that the text is now placed at (200,200).

By careful use of groupings one can gather together all the graphical content for an SVG COG inside a <g> element labelled with a unique identifier (in a similar spirit to the unique ID of a PDF COG). We can then display this COG later on by means of a <use> element. Now, if this COG exists as an immediate child of the SVG tree's root node then it will be rendered even if it is never referred to, because the SVG renderer traverses the SVG document tree in a depth-first left-to-right fashion. To stop the COG definition from being rendered too early it can be placed inside a <defs> element. The SVG renderer knows that any SVG located within such an element should not be rendered until it is explicitly called out and used.

To complete the picture we note that <svg> elements can be nested inside each other and this gives us an extra layer of encapsulation and protection. By additionally wrapping the COGs inside <svg> elements they can then exist as a separate document fragments. This also means that SVG COGs become explicitly clipped to their bounding box, and so more closely match the behaviour of PDF COGs. Another benefit is that resources such as gradients and embedded fonts can be stored in the <svg> wrapper.

5. VARIABLE DATA DOCUMENTS USING DDF

There is a widespread perception, in some parts of the digital printing community, that XML-based tag sets are either used on the Web, in the form of XHTML, or else are likely to be employed in metadata or in other rarefied areas of document description totally unrelated to hard-copy printing. If this ever was true then, increasingly, it is no longer the case. The fact that XML notation is such a popular standard means that there are plenty of software tools for developing new XML tag sets and for manipulating documents marked up in them. This, in turn, means that there is every incentive to standardise on XML notation for as many aspects of digital documents as possible.

Arguably the single most important transformational tool is XSLT[12] , which is currently at the level of version 2.0 . XSLT is itself an XML application and it is much used for transforming an input XML tree into some other form. This output format may be yet another XML tree or it could be some other format with no resemblance to XML and with no vestiges of a tree structure. The very flexibility of XSLT is what makes it so attractive and its behaviour bears many resemblances to that of a functional programming language. Under the guidance of the XSLT script the entire XML input tree can be traversed in breadth-first or depth-first ways; its nodes and their

attributes can be examined and code can be output dependent on what is found in the input tree.

For all of these reasons it becomes appealing to tackle the problem of variable data printing by inventing an XML-based language designed with this very application in mind. Just such a language is DDF, developed by Lumley, Gimson and Rees[13] [14] . DDF holds application data, logical document structure and presentational information all together in a single unified XML container. By exploiting XSLT's 'functional' capabilities it is possible to create objects corresponding to blocks of material in a catalogue (say) which can be parameterised with arguments such as prices, fill colours, fonts and so on. Moreover, these function arguments can be fully bound or partly bound with some values provided very late indeed – if at all.

By combining DDF with a constraint-based layout system it is possible to generate flexible, dynamic catalogues and other configured advertising material. And it is certainly a bonus that the recent availability of SVG as an output format means that DDF can now offer an 'all XML' solution to variable data printing. Although the transformation of a DDF document may initially appear complex (and it can require more than one stage of XSLT processing) there is a certain transparency in having everything in XML when compared to more 'closed' formats such as PDF. Work is under way to investigate whether some aspects of the COG model could usefully be exploited by DDF.

6. CONCLUSION

Document preparation systems have evolved in such a way that most of them are still inflexible when it comes to variable data printing. As we have seen PostScript has much power but is not easy to control, particularly in variable data situations where entire functions, or some of the arguments to them, would need to be ignored without causing calamitous crashes. Moreover, for reasons of interpreter speed in the early days, and for ease of code generation, page creation applications tend to produce PostScript which is essentially final form and with everything bound in place.

This strategy has persisted in PDF, where one has a well-defined data structure that is sequentially interpreted. The problem is that PDF's structures were not generally designed to have clearly visible parameter slots thereby enabling late-bound data to be inserted. Furthermore its structures are emphatically *not* designed to be internally reformatted at run time, via some form of script, prior to re-interpretation to achieve a rather different page appearance. One of the few PDF features to attain true object status and re-usability is the Form XObject and without its existence COG PDF would have been impossible.

The advantage of XML notation is that it reflects tree data structures, which are almost universal in computer science. XML attributes resemble procedural parameters and scripting systems, such as ECMAscript and XSLT, enable the tree to be reshaped and re-interpreted. These features alone give SVG some important advantages over PDF in terms of flexibility: the attributes on its nodes are clearly visible and it is commonplace for ECMAscript or XSLT to be able to create dynamic SVG documents. SVG's clear interpretation semantics make it possible to envisage genuinely parameterisable, variable-data, SVG COGs as opposed to the static encapsulation of PDF COGs.

The message seems to be (and this is particularly apparent in the COG and DDF work) that we need our modern digital documents to remain dynamic and malleable until very close to the moment that they are rendered. This kind of flexibility is certainly provided (albeit with some loss of performance at times) by adopting XML-based solutions. As XML-based solutions propagate ever closer to final-form document layout it seems clear that the world of digital documents and digital printing is set to change out of all recognition over the next few years.

ACKNOWLEDGEMENTS

Thanks are due to my two colleagues at Nottingham, Steven Bagley and Alex Macdonald, for many animated discussions and for help in providing COG-based material for this paper. Conrad Taylor's helpful report of John Warnock's Lovelace Medal talk (`http://www.epsg.org.uk/pub/warnock/warnock_01.html`)

filled in many historical details. It's a pleasure also to record our collaborative work with John Lumley and his colleagues at HP (UK) resulting from an initial interest in COGs by Hui Chao at HP (Palo Alto).

REFERENCES

1. Adobe Systems Incorporated, *PDF Reference (Third Edition) version 1.4*, ISBN 0-201-75839-3, Addison-Wesley, December 2001.
2. Simon Fuller, ''PostScript as a Design Tool,'' in *Real World PostScript*, ed. Stephen F. Roth, Addison-Wesley, 1988.
3. Steven Bagley, David Brailsford, and Matthew Hardy, ''Creating reusable well-structured PDF as a sequence of Component Object Graphic (COG) elements,'' in *Proceedings of the ACM Symposium on Document Engineering (DocEng'03)*, pp. 58–67, ACM Press, 20–22 November 2003. ISBN 1-58113-724-9
4. PODi, *Print markup language functional specification version 2.1*, June 23 2003. `http://www.podi.org`
5. Felipe R. Meneguzzi, Leonardo L. Meirelles, Fernando T. M. Mano, Ana Cristina B. da Silva, and João B. S. de Oliveira, ''Strategies for Document Optimization in Digital Publishing,'' in *Proceedings of the ACM Symposium on Document Engineering (DocEng04)*, ACM Press, October 2004. Milwaukee, Wisconsin
6. Steven Bagley and David Brailsford, ''Page Composition using PPML as a link-editing script,'' in *Proceedings of the ACM Symposium on Document Engineering (DocEng'04)*, pp. 134–136, ACM Press, 27–31 October 2004. ISBN: 1-58113-938-1
7. Steven R, Bagley and David F. Brailsford, ''The COG Scrapbook,'' in *Proceedings of the ACM Symposium on Document Engineering (DocEng'05)*, p. 31, ACM Press, 2–4 November 2005. Bristol, UK
8. *Squiggle – the SVG browser*. `http://xml.apache.org/batik/svgviewer.html`
9. *SVG 1.2 – Multiple Page*. `http://www.w3.org/TR/2004/WDSVG12-20041027/multipage.html`
10. Alex Danilo and Jun Fujisawa, *SVG as a Page Description Language*. See: `http://www.svgopen.org/2002/papers`
11. Alexander J. Macdonald, David Brailsford, and Steven Bagley, ''Encapsulating and Manipulating Component Object Graphics (COGs) using SVG,'' in *Proceedings of the ACM Symposium on Document Engineering (DocEng'05)*, pp. 61–63, ACM Press, 2–4 November 2005. Bristol, UK
12. W3C, World Wide Web Consortium, *XSL Transformations (XSLT) Version 2.0*. `http://www.w3.org/TR/xslt20/`
13. John Lumley, Roger Gimson, and Owen Rees, ''A Framework for Structure, Layout and Function in Documents,'' in *Proceedings of the ACM Symposium on Document Engineering (DocEng'05)*, ACM Press, 2–4 November 2005. Bristol, UK
14. John Lumley, Roger Gimson, and Owen Rees, ''Extensible Layout in Functional Documents,'' in *EI 2006 Digital Publishing Conference*, 16–17 January 2006.