Page Composition using PPML as a Link-editing Script

Steven R. Bagley and David F. Brailsford
Electronic Publishing Research Group
School of Computer Science & IT
University of Nottingham
Nottingham NG8 1BB, UK
{srb , dfb}@cs.nott.ac.uk

ABSTRACT
The advantages of a COG (Component Object Graphic) approach to the composition of PDF pages have been set out in a previous paper [1]. However, if pages are to be composed in this way then the individual graphic objects must have known bounding boxes and must be correctly placed on the page in a process that resembles the link editing of a multi-module computer program. Ideally the linker should be able to utilize all declared resource information attached to each COG.

We have investigated the use of an XML application called Personalized Print Markup Language (PPML) to control the link editing process for PDF COGs. Our experiments, though successful, have shown up the shortcomings of PPML's resource handling capabilities which are currently active at the document and page levels but which cannot be elegantly applied to individual graphic objects at a sub-page level. Proposals are put forward for modifications to PPML that would make easier any COG-based approach to page composition.

Categories and Subject Descriptors
E.1 [Data]: Data Structures — *Trees*; I.7.2 [Document and Text Processing]: Document Preparation — *Markup languages*; I.7.4 [Document and Text Processing]: Electronic Publishing.

General Terms
Algorithms, Documentation.

Keywords
PDF, PPML, graphic objects, Form Xobjects, link editing .

1. INTRODUCTION
The Component Object Graphic (COG) model for PDF, introduced in [1], creates each page of a document from separate graphical blocks. Taking this present paper as an example, each heading, paragraph, etc., in the COG approach, would be a separate graphical object. In what follows our strategy would apply to COGs expressed in any language, but for the purposes of our tests all COGs were expressed in PDF.

FINAL DRAFT of paper accepted for:
DocEng'04, October 28–30, 2004, Milwaukee, Wisconsin, USA.

Each COG is required to be a self-contained entity, independent of any previous graphical operations. The COG sets up the state it requires before it draws anything and resets the state once rendering complete. In this way the appearance of a page is independent of the order in which COGs are placed on it. If each COG represents a unit such as a paragraph, table, diagram etc. then this effects a good balance between positional flexibility and potential re-usability of components.

COGs are drawn on the PDF page by the use of a 'Spacer' object, which causes the COG to be rendered on the page at a particular position. Since the final page position of each COG is not known when the COG is defined, all COGs are drawn relative to a local origin. It is the spacer that applies a specific translation in order to locate the COG on the page. Internally, PDF-COGs are implemented using PDF's FormXObject structure[2].

To demonstrate COG-PDF technology, an Acrobat plug-in was created to allow users to directly drag and drop COGs onto a page. A *ditroff* [3] post-processor, *pdfdit*, was written to create COGs, with all internal spacings and displacements expressed relative to a local COG origin (rather than the page origin).

2. COG LINK-EDITING
The flexibility provided by COGs can only be fully exploited if COGs from multiple sources can be combined to form a composite document. To bring this about we need to borrow, from compiler technology, the idea of a *Linker* [4,5] to take COGs from a variety of sources and link them together to form a single document.

In a program language linker, object modules are created using a local base address of zero (exactly as our COGs all have a local (0,0) origin). But unlike this traditional linker, where the relative location of the object code within the final executable is unimportant, the placement of COGs on the visible page is vital for achieving the desired appearance. Therefore a COG linker needs to know which COGs appear, on which page, and at which positions. Our search for a suitable link-editing language (preferably XML based) led us to *Personalized Print Markup Language* (PPML).

2.1. Introduction to PPML
PPML was developed by a consortium of digital press manufacturers (under the name PODi [6]) to meet the needs of personalized printing (where each document is a custom variation of some master template). Essentially it is a standardized method for describing personalized documents [7]. The impetus behind PPML is to allow a page to be composed from blocks of material that can be Raster Image Processed (RIPped) just once and then cached in their RIPped form for reuse.

Figure 1. The logo and Web address of a church

The following listing shows a sample PPML document corresponding to the diagram of Figure 1.

```
<PPML><DOCUMENT_SET><DOCUMENT>
<REUSABLE_OBJECT>
 <OBJECT Position="0 0">
  <SOURCE Dimensions="723 227"
Format="application/postscript">
   <EXTERNAL_DATA Src="logo.eps" />
  </SOURCE></OBJECT>
 <OCCURRENCE_LIST>
  <OCCURRENCE Name="churchlogo"/>
 </OCCURRENCE_LIST>
</REUSABLE_OBJECT>
<PAGE Dimensions="595 842">
 <MARK Position="72 300">
  <OBJECT Position="0 0">
          <SOURCE Dimensions="450 100"
       Format="application/postscript">
    <INTERNAL_DATA>
     /Times-Italic 18 selectfont 0 12 moveto
        (http://www.beeston-free.org) show
    </INTERNAL_DATA></SOURCE></OBJECT>
   </MARK><MARK Position="72 200">
    <OCCURRENCE_REF Name="churchlogo" />
   </MARK></PAGE></DOCUMENT>
</DOCUMENT_SET></PPML>
```

In this example the logo for the church has been declared as a <REUSABLE_OBJECT> so that it can potentially be used on more than one occasion. Note that PPML does not define its own drawing operators. Instead, content is imported in whatever format the PPML consumer understands (the example here uses PostScript). The content can either be embedded in the PPML stream as <INTERNAL_DATA> or it can be called up (e.g. as logo.eps in the present example) from an external file as <EXTERNAL_DATA>. A PPML document's root node is <PPML> and this can contain one or more <DOCUMENT_SET>s each of which contains at least one <DOCUMENT>. In what follows we confine our discussion to PPML scripts that represent just a single document. Content defined by a <REUSABLE_OBJECT> at the <DOCUMENT> level is available for the rest of the document, and the same rules apply to content defined at the <PPML>, <DOCUMENT_SET> and <PAGE> level. Note the use of the <OBJECT> and <SOURCE> elements to place pieces of content within the space of the <REUSABLE_OBJECT>. The <OCCURRENCE_LIST> allows many <OCCURRENCE>s of the content to be defined, each with a different transformation applied by an optional <VIEW> tag.

In a typical document most of the content will take the form of non-reusable objects, rather like the Web address part of the current example. All content is imaged on the page by <MARK> elements, which have a Position attribute, to define where the content appears. An <OCCURRENCE_REF> enables one to invoke a particular <OCCURRENCE> of the content. For one-off content, as here, it can be defined inside the <MARK> itself.

3. MAPPING COGS USING PPML

Because PPML is content-agnostic the example we have just shown could equally well be applied to PDF versions of the church logo and its Web address caption. In particular, each of these components might be turned into a PDF COG. On the whole PPML works well as a link-edit scripting language for COGs, but an analysis of its strengths and shortcomings proves very illuminating.

Let us begin with the COG object features that map nicely into PPML. Firstly, the Spacers in COG PDF map directly to PPML <MARK>s with the Position attribute placing the COG on the page. Next, the COG's unique identifier can be used as a name for a single <OCCURRENCE>. This identifier can also be used as the Name attribute of <OCCURRENCE_REF>, to refer to a COG that has been made into a <REUSABLE_OBJECT>. The FormXObject that constitutes the PDF COG maps directly to PPML's <REUSABLE_OBJECT>. The size of the COG is stored in the <SOURCE>'s Dimensions attribute (the Format being given as "application/cog") and the PDF content stream is copied into the <INTERNAL_DATA> section. The order of Spacers on the PDF page object is identical to the order of <MARK>s in the PPML <PAGE>, and the order of <PAGE> elements is identical to a depth-first search of the PDF pages tree.

Within PDF, each page contains a list of *references* to the FormXObjects that are used on that page (in its **Resources** dictionary) and it would appear sensible at first to map this across to PPML, placing the <REUSABLE_OBJECT> definitions at the <PAGE> level. However, PPML does not provide a general method of referring to objects, except those on the same page or those that have been made truly global at the topmost levels of PPML. This in turn, means that for COGs to be shared they must be placed at the <DOCUMENT> level. Currently we promote all our COG objects to the <DOCUMENT> level to make it easier to add or remove them from the final document.

4. THE RESOURCE PROBLEM

The major area where problems were encountered in linking COG-PDF via PPML, lay in the handling of resources (e.g. fonts) used by the COGs. In PDF, COGs are tightly coupled to the resources they use by means of dictionary entries in the COG's FormXObject, which point to the actual resources held at either the page or document level. The names used for these resources have a scope that is local to the FormXObject dictionary. Thus the same Times Roman font might be referred to as R in one COG and as T in another. Equally, and very usefully, two different COGs might use the same name, H, to refer, perhaps, to two subtly different versions of a Helvetica font, with different metrics, one for use in a company logo letterhead and the other for use in body text.

In PPML there is indeed a <SUPPLIED_RESOURCES> element, which can be used to name resources at the <PPML>, <DOCUMENT_SET>, <DOCUMENT> or <PAGE> levels and these resources are then visible for the rest of that level (although it is possible to alter their scope; for example, they can be made *global* via a Scope="global" attribute declaration, in which case they are cached in the consumer application to be available to other PPML streams). PPML also supplies a <REQUIRED_RESOURCES> element that can, in turn, contain a <SUPPLIED_RESOURCE_REF> to indicate that a particular

<PAGE>, <DOCUMENT> etc. needs this resource. The PPML specification makes it clear that this element is optional, but hints that it might be useful in pre-flight checking of a print job, or in subsetting pages from a composite document.

The nub of the problem with PPML's scoping rules is as follows. Objects and resources can have a global scope, at the level of the whole document or document set (analogous to the **extern** storage class in the C language) or they can have a scope restricted to the current page (analogous to the **static** storage class in C, which restricts scope to the current C source file). But PPML lacks a mechanism for *local* scope, as might be found for variables inside C functions, or member names in a C++ class, or indeed resource names in a PDF FormXObject dictionary. More precisely, there is no way in PPML to attach resources to a <MARK>, <OBJECT> or <REUSABLE_OBJECT>. This omission means that any PPML-based linker for PDF COGs would need to interpret the PDF content stream to find out which resources that COG uses. This would slow down the linker (considerably) and would also require it to understand PDF graphical operators.

The lack of locally scoped names in PPML means that each <SUPPLIED_RESOURCE> declares a unique identifier (the ResourceName attribute) for the resource itself using the name by which the resource is called out within the graphical objects themselves. In addition, it declares a label (using the Name attribute) by which the resource can be referred to within the current scope. This, in turn, means that it is quite impossible to use the same Name to refer to different resources. Equally, any attempt to use different ResourceNames for the same resource (by supplying two <SUPPLIED_RESOURCE> sections with different ResourceNames) would simply result in multiple copies of the same resource, rather than the sharing of one copy.

To overcome this problem we have adapted PPML to allow a local binding of resources to objects. The resulting language we have called PaC (*P*PML *A*dapted for *C*OGs). If a PaC script is now used for link editing COGs then the link editor can identify and merge all the resources needed for each page of the output document. Moreover, the extra information in PaC allows it to unify resource requests and resolve name clashes.

Let us assume that the Web address and caption COGs for Figure 1 both use the local name, F, to refer to the Times Italic and Times Bold fonts, respectively. If the linker's output is to be a unified COG PDF, then there is no problem. The FormXObjects of the output COGs already support two different local bindings for F. However, to produce a conventional PDF file, with resources located at the Page or Document level then a PaC-based linker has to identify the clash and rename one of the uses.

One final and subtle resource problem remains. If resources are to be shared then we require some means of being sure that identically named resources really do represent exactly the same thing. The solution to this latter problem within PaC is two-fold. Firstly, the problem of correctly identifying a resource is solved by assigning each resource a Name, which is a globally unique identifier (again, as with COGs, a standard UUID is used). Furthermore this ResourceName attribute is moved from the resource definition to where the resource is referenced. This mimics COG-PDF's behaviour and so removes the problem of namespace collision in resources. Secondly, to tightly couple resources to COGs, the <REQUIRED_RESOURCES> section is now allowed only inside a <REUSABLE_OBJECT> definition and, moreover, it *must* occur there. In this way the ResourceName is used to declare information on the resources used by a COG, rather like a formal parameter to a procedure.

By adopting the PaC extensions to PPML our linker can now detect whether particular <REUSABLE_OBJECT>s or uniquely named resources are shared between several <PAGE>s and if so they can be promoted to the <DOCUMENT> level and kept in the cache, with all name clashes resolved. This sort of optimization would be quite impossible if conventional PPML were used, short of requiring the linker to indulge in parsing of the content-stream.

5. CONCLUSIONS

From the point of view of traditional workflow, the existing PPML definition seems adequate. The PPML consumer (usually a printing press) will find its resources in the same manner as when processing a PostScript document (the resource models are virtually identical). If a resource is absent then the job will abort.

However, as we have seen, PPML lends itself to being processed in other ways. For example, a tool could be written that subsets the different documents in a PPML stream to separate files, or which tries to optimize the use of <REUSABLE_OBJECTS> [8]. In these cases, where the structure of the PPML file is altered, the limited information supplied by PPML on resource usage, becomes a problem. We believe that the linking of graphic objects to their resources should be explicit, visible and mandatory.

By altering PPML to handle tight coupling of resources to objects (as we have done in PaC), our linker can now solve the twin problems of multiple names for the same resource and the same name being used for different resources. Equally, this mandatory naming of resources, on a per object basis, is invaluable for extracting reusable objects from documents.

6. REFERENCES

1. 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)*, p. 58–67, ACM Press, 20–22 November 2003.

2. Adobe Systems Incorporated, *PDF Reference (Third Edition) version 1.4,* Addison-Wesley, December 2001.

3. B. W. Kernighan, "A Typesetter Independent TROFF," Computing Science Technical Report No. 97, Bell Laboratories, Murray Hill, New Jersey 07974, March 1982.

4. D. W. Barron, *Assemblers and Loaders,* Macdonald , 1978 .

5. John Levine, *Linkers and Loaders,* Morgan Kaufmann , 1999 .

6. PODi, *Print markup language functional specification version 2.1*, June 23 2003. http://www.podi.org

7. D. DeBronkart and P. Davis, "PPML (Personalized Print Markup Language): a new XML-based industry standard print language.," in *XML Europe 2000*, p. 1–14. Paris, France

8. Felipe R. Meneguzzi, Leonardo L. Meirelles, Fernado 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.