

# Extracting Reusable Document Components for Variable Data Printing

Steven R. Bagley, David F. Brailsford and James A. Ollis

Document Engineering Laboratory

School of Computer Science

University of Nottingham

Nottingham NG8 1BB, UK

{srb,dfb,jao}@cs.nott.ac.uk

## ABSTRACT

Variable Data Printing (VDP) has brought new flexibility and dynamism to the printed page. Each printed instance of a specific class of document can now have different degrees of customized content within the document template.

This flexibility comes at a cost. If every printed page is potentially different from all others it must be rasterized separately, which is a time-consuming process. Technologies such as PPML (Personalized Print Markup Language) attempt to address this problem by dividing the bitmapped page into components that can be cached at the raster level, thereby speeding up the generation of page instances.

A large number of documents are stored in Page Description Languages at a higher level of abstraction than the bitmapped page. Much of this content could be reused within a VDP environment provided that separable document components can be identified and extracted. These components then need to be individually rasterisable so that each high-level component can be related to its low-level (bitmap) equivalent. Unfortunately, the unstructured nature of most Page Description Languages makes it difficult to extract content easily.

This paper outlines the problems encountered in extracting component-based content from existing page description formats, such as PostScript, PDF and SVG, and how the differences between the formats affects the ease with which content can be extracted. The techniques are illustrated with reference to a tool called COG Extractor, which extracts content from PDF and SVG and prepares it for reuse.

## 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

PostScript, PDF, SVG, graphic objects, Content Extraction, Variable Data Printing.

FINAL DRAFT of Full paper accepted for:  
*DocEng'07*, August 29, 2007, Winnipeg, Manitoba, Canada  
Copyright Bagley Brailsford and Ollis 2007.

## 1. INTRODUCTION

Many of the issues in the digital encoding of high-quality page layout were solved by Adobe in the mid-1980s with the introduction of Adobe PostScript [1]. This page description language provided users with a device-independent way of describing the appearance of a document. The same digital representation could just as easily be printed on a consumer-grade laser printer as on a professional image setter and, crucially, it could exploit both devices to their full capabilities. PostScript provided facilities for raster and vector graphics together with high-quality typographic support.

More recently, PostScript technology has been refined into Adobe PDF (Portable Document Format) [2] and has also been used as the graphical model behind SVG (Scalable Vector Graphics) [3] – an XML-based representation for digital documents.

A typical print workflow involves a PostScript document being converted to a raster image by a process known as Raster Image Processing (or RIPping, for short). It is this raster image that eventually drives the low-level printing engine. On smaller devices, such as a laser printer, the document is RIPped on the device itself but as computer power and network speeds have increased, commercial image setter machinery increasingly employs dedicated computers to do the RIPping, with the bitmapped page being transmitted over a network to the printing engine.

In the past five years faster raster processing has combined with new advances in paper handling and ink technology to create digital printing devices which are very nearly as fast as traditional plate-driven printing presses. This has enabled the print industry to shift from “single-document, many copies” to a variable data paradigm of “many documents, one copy”. This Variable Data Printing (VDP) allows each instance of a document to be customized so that a direct marketing campaign could use data collected from loyalty schemes, to suggest products that are attractive to each recipient. With so much direct-mail marketing going straight to the wastebasket, the hope is that personalized marketing will capture the recipient’s attention whereas a generic mailshot would not.

VDP is built on two main technologies. At the hardware level, the advent of digital offset presses (such as the HP Indigo [4]) enables the efficient production of VDP documents by removing the need to make plates for a print run. Before the digital press, VDP was limited to overprinting custom data, often using a laser printer, into ‘copy holes’ left in a static template previously printed on an offset press. With a digital press, there are no restrictions on the variability of a document. Each document instance can be radically different from the next, with changes including

repositioning of objects on the page or the wholesale replacement of one block of content with another.

The second major requirement for VDP is new software techniques for describing documents at a component level. Tools such as HP's Document Definition Format (DDF) [5,6] are providing new abstractions about VDP documents and how to design and program them. At a lower level, technologies such as PPML [7] (Personalized Print Markup Language) or COGs [8,9] (Component Object Graphics) provide new methods of describing documents that encourage reuse of common components by describing the whole page in terms of blocks.

The RIPPING of a VDP document starts with a VDP-friendly representation in a language such as PPML. This document is sent to the RIP device in the usual way but a VDP-aware piece of software — generically called the 'PPML consumer'— is responsible for interpretation. PPML's default page coordinates are in points (just as in PostScript and PDF) and these are used to place document component objects on the page. In addition to these placement coordinates PPML allows invariant objects to be tagged as being "reusable" and it also requires that the source language (e.g. PDF) of each object be specified.

For a traditional non-VDP document, rasterization generates a set of 1-bit-per-colour images, based on resolution-dependent device coordinates. These images are used to drive the output device (be it an imagesetter or a platesetter, ). Within a VDP system a set of 1-bit images is still produced but the procedure thereafter is rather different and though the details may vary for different VDP output devices the general principles can now be outlined.

Each document component is rasterized individually to a separate set of colour bitmaps. This set, for the given component, is then interpolated onto the page at the correct position using bitwise operations. Invariant parts of the raster can be left in the cache from the previous RIPPed page but areas of the page corresponding to replaced content are first cleared with a bitwise-AND mask, to remove any data that will now be covered by the new components, followed by a bitwise-OR to insert each new chunk of content.

The advantage of component reuse in the VDP arena is one of speed. Reused components can be cached at the raster level and merged directly with the page so that invariant parts of a page need not be rasterized more than once. Traditionally, the time taken to RIP a document has not mattered (being inconsequential compared to the time taken to move the plates from the image setter to the printing press, for example) but in a VDP-workflow, using digital presses that need to be fed continuously with page data, any delay is costly. A speed-up of just 0.1 seconds in page rasterization on a 30000-page VDP job can lead to an overall saving of almost an hour in the printing time.

New documents can, in principle, be created in a component form, with VDP in mind. But if it is desired to extract component content for a VDP workflow, from existing monolithic documents stored in traditional PDLs, then many challenges present themselves.

The remainder of this paper looks at the problem of extracting self-contained components from legacy digital documents. We outline the general principles using PDF and SVG as examples in the context of a tool called COG Extractor [10].

## 2. ENCODING THE PAGE APPEARANCE

Before analysing the task of extracting content from legacy document formats it is necessary to understand how different PDLs model document appearance. This review is limited firstly to the common formats of PostScript and PDF, because they exhibit linear and page-based inheritance of graphic properties, and secondly to SVG because it provides interesting sidelights on inheritance of graphic properties as a result of its XML-based tree-structured model.

### 2.1. PDL Basics

The various capabilities of output devices means that device-independent PDLs need to describe a document at a higher level of abstraction than those PDLs targeted at a specific typesetter. In particular, it is necessary to have a common graphics model with 'user space' coordinates that are transformed to device space coordinates at a late stage once the target output device is known.

If a document is considered to be a series of marks (e.g. glyphs, lines, raster images etc) imaged with certain properties onto a page (typeface, font size, line width, position, colour etc) then a PDL encodes all this information so that its interpretation by a computer causes the document to be imaged.

The document can be programmatically modelled in many ways and PDF, PostScript and SVG all use different language constructs for describing the document even though they all have a near-identical graphics model. For example, PDF and SVG<sup>1</sup> are both declarative languages (see sections 3.1 and 3.3 for more details). Their 'execution' consists of interpreting a fixed data structure and so they produce the same output every time they are displayed. In contrast, PostScript is a full imperative programming language and the displayed output is the result of executing a PostScript program which could, in principle, behave differently every time, perhaps by reading in external data or by generating random numbers.

A further difference between the three PDLs lies in the structure of the files. PDF and PostScript are stream based, with the document being described as a sequence of operators whose execution causes rendering to take place. However SVG, being specified in XML, uses a tree model with the nodes on the tree specifying either marks on the page (leaf nodes) or the grouping of marks together (interior nodes).

There are also differences in the way that the scope of graphical properties is treated. In PDF and PostScript a graphic state model is used. Once a graphical property has been set its effect persists until the property is altered, or the end of the stream is reached. SVG differs by having a hierarchical approach; the setting of any graphical property has an effect only on the current document node and on the sub-tree of all its child nodes

Broadly, then, our three PDLs, when used to render pages, can be classified as being either state-based or hierarchical, and either statically interpreted or dynamically executed. These properties are orthogonal to each other and so a PDL can be both state-based and dynamic (e.g. PostScript) or hierarchical and static (SVG).

## 3. CONTENT EXTRACTION PROBLEMS

We now consider the ease with which content can be extracted from a document. We assume, in each case, that the surface

---

<sup>1</sup> Ignoring the use of JavaScript and animation options in SVG.

syntax has been parsed leaving an internal sequence of tokens (or, in the case of SVG, a tree) for the later stages of the parser to handle. We begin by considering what the process of ‘Extracting Content’ actually involves.

Extracting Content can be defined as selecting the parts of the input document stream that are necessary for correctly imaging the selected piece of content. A more helpful approach is to turn the definition around and to describe it as “Removing the pieces of the input document that have no effect on the imaging of the selected section”, which more accurately describes what needs to be done when processing the document for content extraction.

This immediately raises two further problems. The first is the obvious one of deciding which pieces of the stream are involved with imaging the content to be extracted. If the user selects the content of interest in the usual ‘rubber-band’ style, via some user interface, then this leaves the program with a rectangle defining the extent of the area to be extracted. Next, this area can be tested against the bounding boxes of each object on the page. If the two bounding-boxes intersect, then the object is kept. If they do not then it is removed.

The second problem is trickier to solve. It involves finding out which pieces of the document’s program are responsible for the *graphical state* of the content seen within the delineated area of interest. The problem stems from the fact that an imaging operator is executed in a context constructed from the execution of *all* the preceding operators in the document stream, or the ancestor nodes in an SVG document tree, and not just from the graphical operations within the selected area. This can be illustrated by the following simple piece of PostScript which draws the output shown in Figure 3.1, i.e. the words ‘Hello World’ typeset in Helvetica and Times

```
1 0 0 setrgbcolor
300 300 moveto
/Helvetica 12 selectfont
(Hello ) show
/Times-Roman 12 selectfont
(World) show
showpage
```



**Figure 3.1 — Output of Example Postscript file**

Now suppose the user wishes to extract just the word ‘World’ from the displayed document. The version below highlights all the code involved in imaging the word ‘World’ at the correct position.

```
1 0 0 setrgbcolor
300 300 moveto
/Helvetica 12 selectfont
(Hello ) show
/Times-Roman 12 selectfont
(World) show
showpage
```

Some of the effects are obvious: the `setrgbcolor` and second `selectfont` both set visible properties of the image and their effect persists until an explicit change is made. What is perhaps more surprising is that the `selectfont` for the word

‘Hello’ also has a significant effect on the output of ‘World’, for reasons we shall now examine.

While ‘Hello’ is given an absolute position on the page because of the `moveto` command, the word ‘World’ is not. According to the semantics of PostScript, the `showing` of ‘Hello’ advances the cursor by the width of the word. Therefore, ‘World’ is imaged immediately after the word ‘Hello’ (and the set widths of 12 pt Helvetica determine where that starting point will be). In other words, the position of ‘World’ is dependent on the context in which it is displayed.

This means that we need to be careful when removing seemingly unused content, because it might actually have an effect on the output. If the `show` for ‘Hello’ were removed from the document, we would not want the text string for ‘World’ to move. But, as things stand, it would do so unless we calculate the width of the deleted Helvetica ‘Hello’ text and add that displacement onto the `moveto` command earlier in the document.

The result of these side-effects is that operators can affect all other operators and imaging commands which follow them in the stream (or which are descendants of the current tree node in SVG). Indeed, the only place where it is guaranteed that removing an operator will not change the imaged appearance is when the removed operator is preceded by all the operators responsible for the appearance of content within the selected area.

### 3.1. SVG

The hierarchical nature of SVG makes it one of the simpler PDLs from which to extract content. This is because the tree structure automatically implies containment of the effect of various graphical properties. That is, any graphical property defined on a node applies only to itself and to any descendant nodes. Crucially, any properties specified on a node have no effect on sibling nodes within the tree.

Since the properties on a node affect only its children, it is possible to safely remove nodes from the tree if they do not image content on the page. Furthermore, it is possible to backtrack up the tree and remove grouping nodes if none of their descendants’ images intersects with the bounding box of the area to be extracted. By processing each node, and removing it if it does not intersect with the bounding box of the selection, it is possible to build up a tree that draws only the extracted content. It may still contain some superfluous graphical properties, relevant only to omitted content, but these operations can be optimized out as we shall see later.

Unfortunately, whilst the hierarchical structure of SVG makes extraction easier there is another problem that affects the ease of SVG content extraction, namely, how resources (such as fonts or images) are accessed. In SVG, fonts can be specified in two ways; they can either be included in the document file, or, more usually, can be assumed to be installed on the system and then referenced (by name) within the document. The first case is not a problem: the document has access to the font and so can copy it into the extracted file with ease (ensuring that identical rasterization will take place on any system).

In the second case the extractor tool must have access to all fonts present on the system. Without access to these fonts and their metrics it is not possible to correctly calculate the bounding boxes of rendered text. As we saw earlier this, in turn, might make it impossible to work out the exact position of certain parts of the

document. A related problem is to ensure that the extracted content correctly references these fonts so that the extracted component displays correctly.

Images in SVG are generally referenced via a URL to an external entity and while the extractor will most likely have access to this entity it needs to ensure that access continues to work for the extracted piece of content.

### 3.2. PostScript

Imperative PDLs such as PostScript are probably the most difficult form of PDL from which to extract content, largely because the exact effect of an operator is not known until it is executed. The presence of conditionals and loops in the PostScript programming language means that the output can literally be quite different each time the document is displayed.

The solution to this problem is simple: rewrite the PostScript document to remove any conditionals, loops, and so on, from the document. The easiest way to do this is to execute the document but, rather than imaging the content, write out all of the commands that have an effect on the imaging into a new file along with their called parameters. This gives a version of the document that is static (that is, every time it is executed the same output is generated) and which has unrolled all the loops, conditionals and procedure calls found in the document.

Once these processes have been completed one is left with a document that is equivalent to a PDF content stream (in fact, Adobe Distiller when converting from PostScript to PDF performs exactly this process). Therefore, all the considerations for parsing PDF (described in the next section) will also apply to an expanded PostScript document.

As with SVG, PostScript documents can refer to both internally and externally defined fonts. Thus the issues described for SVG resource handling apply here also.

### 3.3. PDF

Declarative stream-based document formats such as PDF (and compiled-out PostScript documents) would seem to create a host of difficulties for content extraction that are, on the face of it, more complex to solve than in a hierarchical PDL such as SVG. However, it turns out that this is not necessarily the case.

To all intents and purposes, the PDF format enforces a tight coupling of resources (fonts, images etc) to the objects that use them (the pages). This means that the issues raised with SVG and PostScript, about external references to resources, disappear. It is relatively simple to detect whether a resource is being used in the finally extracted component and to remove the links to the resources that are no longer used. While the PDF specification, does theoretically allow for certain resources (mainly fonts) to be called by references to external entities, this is extremely rare in PDF found out in the wild (generally it is only seen as the by-product of poorly written open-source or third-party products) and so the problem can effectively be ignored.

The biggest hurdle when extracting content from a PDF file is to identify which parts of the document are actually involved in imaging content on the page. Inside a PDF, an amorphous stream of tokens describes each page of the document. These tokens can

be divided into those that image content and those that set graphical properties. As illustrated earlier, it is difficult to determine which graphic-property tokens are involved in describing the appearance of the extracted content because there is no inherent containment. Therefore a piece of content imaged by tokens at the end of the stream could be influenced by graphical properties set by tokens at the start. It seems that the extractor tool needs to keep track of exactly which properties have an effect on all of the imaging tokens in the stream.

But if we step back and consider the bigger picture for a second, it is possible to devise a method that can simplify the task considerably. As noted with SVG, its implicit tree-structure seems to give a containment to the effect of graphical properties that is missing in stream-based languages such as PDFs. But is this containment actually missing? It has already been noted that each graphical property has effect from the point it is set until the end of the stream (or until it is reset). Does this not imply a sense of containment?

In fact, it turns out that the stream of tokens in a PDF content stream can be folded into a tree structure that is equivalent to that found in SVG documents, except that it tends to descend to the next level on the last sibling of any given node and so forms a tree that is continuously descending to the right. In what follows it is important to remember that an intermediate form of the document is being created within the parser, one that is more useful for content extraction, and that this will need to be converted back into PDF at a later stage by a simple code-generation engine.

The folding process is best illustrated with the help of diagrams. Figure 3.2 illustrates a PDF content stream schematically, with the imaging operators visualized as circles and the graphical property-setting operators as triangles.

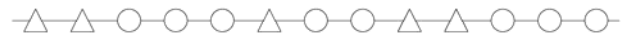


Figure 3.2 — Visualization of a PDF Content Stream

While at first there may appear to be no structure to the stream, the application of a simple rule quickly shows that this is not the case. If the stream is folded in under itself, before every imaging operator that immediately follows a property-setting operator, then, as Figure 3.3 demonstrates, the implicit tree structure hidden within the PDF content stream begins to appear.

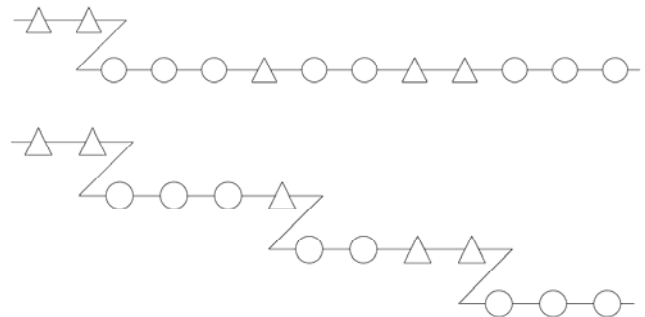
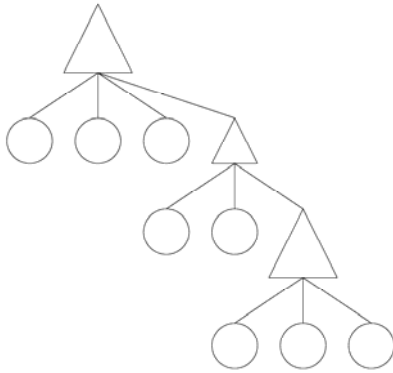


Figure 3.3 — Progressive folding of PDF content stream

The final stage is to coalesce consecutive runs of property-setting operators into interior nodes that represent the graphics state changes at that point and to allow the imaging operations to become leaf nodes, which gives the tree shown in Figure 3.4. It is

important that the PDF rules are followed while merging operators; that is, the later operators must override the effect of earlier ones. For example, if a run of graphical property-setting operators first sets the fill colour to red and then sets it later in the run to blue, only the ‘set to blue’ effect needs to be recorded in the interior node. As we shall see later, the PDF operators that allow for the preservation of the graphical state are also easily represented in this fashion, but as another graphics state interior node that is *not* the last child of its parent.



**Figure 3.4 — The fully-folded PDF Content Stream**

By using this method of PDF folding, we find that it becomes as simple to extract material from a PDF document, as from an SVG document.

At this stage one might be tempted to improve the hierarchical representation of the document, prior to component extraction, by promoting shared graphical properties up the tree. However, it is important to remember that the aim of the extraction process is simply to produce a component whose *appearance* accurately matches the original material. There is no point in expending effort on tree optimisation if the ultimate code generation process for the component can make little use of it.

In fact it turns out that many PDF documents encountered in practice have a good hierarchical structure already (and especially so if they have been produced by high-quality PDF engines such as *Adobe Distiller* or Apple’s *Quartz*). In some cases there will also be internal sub-hierarchies created by the use of save-restore operations. These existing hierarchies will generally carry through into the code produced for the extracted component but one has to recall that the extracted areas will tend to use only a small subset of the state-setting operators (the rest are removed by the optimization techniques described in section 5) and so opportunities for property sharing after extraction tend to be very limited. Once again, therefore, we see that attempts to produce a beautiful tree structure before extraction will generally be a waste of CPU cycles.

## 4. EXTRACTION

As seen in section 3, a hierarchical structure considerably helps the task of extracting content because it provides an easy method of identifying the graphical properties that each piece of imaged content relies on. For any node in the tree that images content (which by definition has to be a leaf node) then the graphical state in which it is imaged is defined as the combination of the graphical state of its ancestor nodes. Therefore, by identifying which pieces of imaged content are to be extracted (e.g. by seeing

if the content’s bounding box intersects with the selected area) and which pieces do not, the tree can be pruned to remove those nodes that do not contribute any graphical appearance to the selected area. The result is a tree which, when imaged, will display only the content selected for extraction.

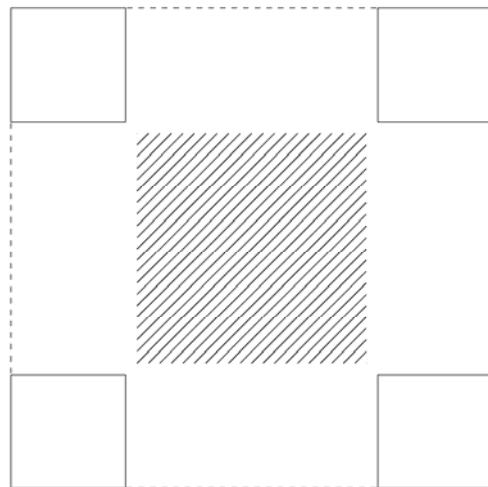
Section 3.3 also demonstrated that it was a simple task to convert a stream-based format such as PDF into a hierarchical format equivalent to that found in SVG. The graphical state nodes created by coalescing graphics state property tokens are effectively equivalent to the grouping nodes in SVG.

However, while the basics of the process are simple, there are a number of things that must be remembered, when processing the internal tree, to ensure that the highest quality output is generated.

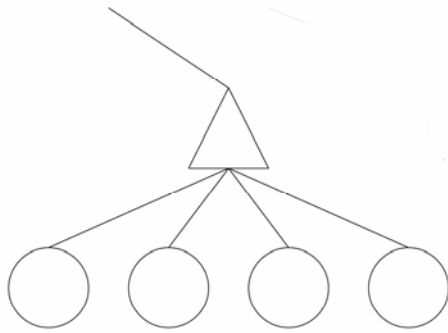
### 4.1. Topiary for Beginners

When pruning the hierarchical representation of the document it is important that the process begins at the leaf nodes and works back up the tree, removing the grouping interior nodes only if all their children have been pruned. While it may seem more sensible to perform a recursive descent of the tree and test whether the bounding box of each node intersects with the area to be extracted and then remove the grouping node (along with its children) if the bounding box intersects the bounding box of all its children, this approach can lead to content that is not visible within the extracted area being left in the tree.

This is best illustrated by Figure 4.1, which shows four squares all of which lie outside the area to be extracted (represented by the shaded area). Figure 4.2 shows an extract from the document tree responsible for drawing this document. It is clear from the graphical representation of Figure 4.1 that none of the squares overlaps the extracted area and so they can safely be pruned from the document tree. However, if you were to do a recursive descent and compare the intersection of the bounding box of the first grouping node (shown in Fig 4.1 by the dotted line and in Fig. 4.2 by the triangle) with the area to be extracted you would find that they do indeed intersect. This intersection would cause the redundant grouping node to be left in the document even though, as the iteration over the tree continues, the four squares would be removed from the document.



**Figure 4.1 — A problematic extraction scenario**



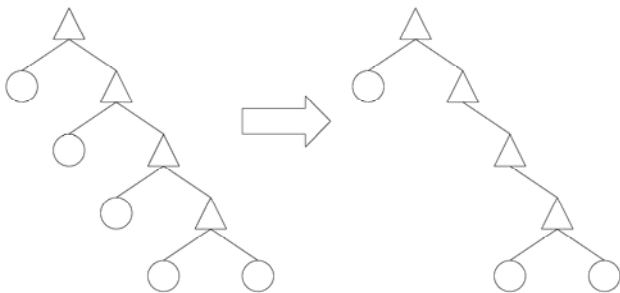
**Figure 4.2 — Tree representation of Fig 4.1**

However, if the process works bottom-up, from the leaf nodes back up the tree, and compares only the bounding boxes of the imaging operator nodes with the area to be extracted, then the four squares will be removed as before. By adding another rule that all grouping nodes with zero children are also removed from the tree it is possible to tidy up any redundant grouping nodes.

Both of these processes can be combined into a single depth-first recursive descent of the tree, where each node is deleted based on the two rules described above: leaf nodes are deleted if they do not intersect the area to be extracted and interior nodes are removed if all their children are deleted as they are processed.

## 4.2. Fine-tuning

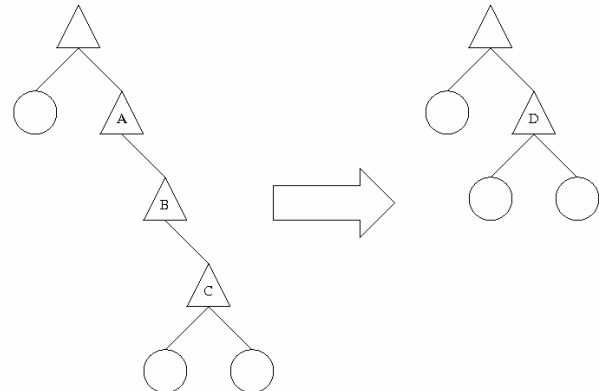
After any redundant nodes have been deleted from the tree, it is left in a state that correctly images the extracted content but it is not necessarily an optimal representation of that content. In the case of our PDF intermediate tree, but also with SVG, it is common to find that the extracted content tree displays a run of nodes which all have only one child — another single grouping node, as illustrated in Figure 4.3. The next step in extraction is to optimise the tree to remove unnecessary grouping nodes.



**Figure 4.3—Removing nodes leaves interior nodes**

The solution to this problem is the same as that described in section 3.3, where multiple runs of graphic state tokens in the PDF content stream are merged together to form the interior nodes of the parser's intermediate tree. In this case, however it is two interior nodes (or, in the case of SVG input, grouping nodes) that are joined together but the same rules still apply. Any properties set in the lower descendants replace the properties in their parents. The result of applying this process to the tree shown in Figure 4.3

is shown in Figure 4.4, where the three interior nodes become a single node.



**Figure 4.4—Merging interior nodes**

If the three nodes being merged had the properties shown in the table below (again illustrated using PostScript notation) then the final row would show the properties set in the single output node.

Node	Properties
A	1 0 0 setrgbcolor 1 setlinewidth
B	0.5 setgray
C	0 1 0 setrgbcolor
D (Output)	0 1 0 setrgbcolor 1 setlinewidth

As is clearly seen, only the last of the three colour setting operators is preserved in the new node, alongside the setlinewidth operator.

In most cases the order in which graphic state operators are executed does not have an effect but in certain circumstances it does and in these cases care must be taken to ensure that the order is preserved. A particularly important example of this are those graphics operators that manipulate the many transformation matrices used in modern PDLs. Since matrix multiplication is generally non-commutative, the order must be preserved so that the same final matrix is generated.

## 4.3. Clipping

So far, it has been assumed that content can be kept or removed from the document depending on whether its bounding box intersects with the area to be extracted but relying on this method exclusively will lead to a jagged boundary to the extracted area. Any material that is not totally outside the extracted area will be kept in the output and be allowed to draw all of its output, including material that is half-in and half-out of the extracted area.

There are two options for dealing with this. The first option is to break down each imaging operation to remove the segments that cross the area to be extracted. This is an easy process for simple primitives such as paths but for complex marks (text, raster graphics, etc) it becomes considerably more complex.

The second option, which is to insert a clipping path into the tree equal to the area to be extracted, is the more desirable. While it will lead to slightly less optimal generated output it will produce the desired imaging effect.

#### 4.4. Positioning

Given that the position of certain objects depends on the actions of previous operators (see the example at the beginning of section 3), it is important that the extractor tool is able to calculate the exact position of each piece of imaged content.

Fortunately, it is relatively simple to keep track of where things are being positioned while building up the intermediate tree, and thus the nodes can be tagged with their absolute position (along with their bounding box). Later, when this intermediate stage is converted back into PDF or SVG, the code can be generated using this absolute position. This will produce functionally equivalent output code to the input, but the operators in the output will no longer rely on relative position based on the execution of previous operators.

### 5. OUTPUT CODE GENERATION

With the redundant nodes deleted from the intermediate (or SVG) tree, the final thing that remains is to convert this representation back into PDF or SVG for use elsewhere. While it may seem perverse that any work should be needed in code-generating SVG (given that all the tasks described earlier could be performed on the DOM tree) there are still some tasks that can be undertaken to improve the quality of the generated output. Broadly speaking, these fall into two categories; code generation and code optimization.

#### 5.1. Code generation

In the cases of PDF and PostScript, code generation is the task of taking the intermediate tree representation of the document and converting it back into the correct syntax. The difference between the output for PDF and PostScript will be entirely syntactical but the task of inferring the best use of PostScript's imperative language operators is a large problem in its own right. Therefore, as before, the discussion will limit itself to considering only the static representation found in PDF.

The code-generation of a PDF content stream from the intermediate tree can be considered as the reverse of the process used to create the tree. Instead of the stream of tokens being folded in on itself, to create a tree, here the tree is being linearized.

Fortunately, this unfolding of the tree can be performed by a depth-first recursive descent of the intermediate internal tree within the parser. As each node is visited, it is serialized to the output stream. For interior nodes, the correct tokens are issued to set all the graphical properties defined by this node and for leaf nodes, the correct tokens are emitted to draw the content at the absolute position calculated when building the tree.

The result is a PDF content stream that can be dressed in the appropriate structures to turn it into a full PDF document, and any resources in the original PDF file can be copied over.

The easiest way to ensure that the resources are correctly transmitted is to copy them across as the intermediate tree is being traversed. But this means that the correct PDF document structure

needs to be established within the tree before it is traversed for code generation.

The approach described so far, of a right-descending tree based on final-sibling interior nodes, will only work providing the original source document did not use the graphics state preservation operators (`gsave/grestore` in PostScript; `q/Q` in PDF). If these operators are used then the intermediate node will look similar to that shown in figure 5.1, where an interior node is *not* the last sibling. In this case, a depth-first traverse will produce the serialization shown in figure 5.2.

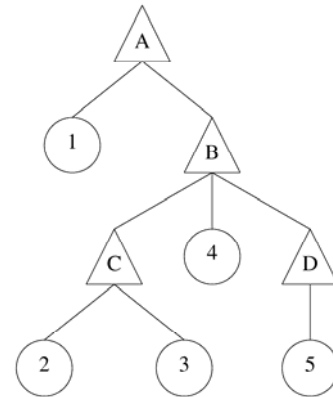


Figure 5.1 — Tree with potential serialization problems



Figure 5.2 — An incorrect serialization of Figure 5.1

As can be seen from these figures, the mark labelled 4, after serialization, is imaged in the context of nodes A-B-C instead of the correct context A-B, and mark 5 is imaged in the context of A-B-C-D, a context that is never described by the document tree.

This can easily be solved by ensuring that when an interior node is serialized it emits a token that saves the graphics state, and immediately after all its children have been serialized emits a token to restore the graphics state. These save and restore tokens will correctly bracket the graphical properties and ensure that the correctly rendered output is produced.

The disadvantage of this approach is that it may lead to a large number of unnecessary save and restore operations taking place that can lead to a PDF rasterizer's storage space being exhausted. However, it should be noted that any properties set by an interior node which is the *last* child of its parent does not need to have its effects contained since they are already being contained by an either an explicit save/restore operation in the parent or by an implicit save/restore at the end of the document stream. By adding this simple rule into the depth-first search it is possible to generate a usable PDF content stream.

#### 5.2. Optimization

While the code generated by the process described above, or by serializing the DOM representation of a pruned SVG document, will correctly display the extracted content, the code produced is likely to be less than optimal. This tends to be more of a problem

for PDF/PostScript extraction but the criticism can also apply to SVG.

Consider the case of a document containing a vector line drawing and some text, where it is the text that is selected for extraction. If, in the original source document, the line diagram is drawn first, then even though the imaging operators for the line diagram are removed, the graphics state setting operators will be copied across since they still exist on the interior nodes at the top of the tree. Note that if the text occurred first then these nodes would be removed since they would be left with no children.

To produce a properly optimized document, it is necessary to establish whether a property that is being set is used by any imaging operator. One solution is to use an algorithm on the tree similar to the following. For each property that is set on a node, check through each of its children to see their imaging operations would make use of that property. If any of them do then, that property can be marked as required and the search aborted.

For any interior nodes reached as children of the given node, first check to see whether the node resets the property under consideration. If it does, then ignore that node and its children. If it does not reset the property, then test each of its children in the same manner.

If the search is not aborted at any point, then the property can be safely removed because no imaging operators were found which relied on that property. If the search was aborted, then at least one imaging operator below it relies on the property and so it needs to remain set. Note that properties that have cumulative effects (such as transformation matrices) should not be candidates for removal if another instance of that property is found further down the tree since the resultant effect is the combination of the two.

### 5.3. Transcoding

A by-product of the way this extraction algorithm works is that is possible to use it as a transcoder between document formats. If the algorithm to remove redundant nodes is bypassed, then the document will be translated and reproduced as a whole. For example, if a different code-generation module were written it would be possible to load a PDF document and output it as SVG.

## 6. COG EXTRACTOR

Component Object Graphics [8,9] (COGs) provide a method of encoding a component document within a PDF or SVG framework. Previous work has shown how documents of this type can be created, and composite documents formed, from pre-existing components. However, the toolset has been lacking a method to repurpose content from existing PDF documents. COG Extractor is a tool that uses the methods described in this paper to allow users to mouse-over a selected part of a document and to extract it as a COG for use elsewhere.

The PDF tool works as a plug-in for Adobe Acrobat, providing a new menu option to select an area for extraction as shown in Figure 6.1. The selected area can then be extracted as a COG and placed inside another COG document (Figure 6.2).

Internally, as soon as the tool is selected the content stream describing the current page is parsed by COG Extractor to build the intermediate document tree. This process uses a custom parser to parse the document streams directly; it does not rely on Acrobat's built-in API for accessing marks on the page. This is because previous work [11] has shown that these routines can lead

to rounding errors creeping into the position of objects on a page. Another drawback to using the API is the need to convert the API data structures into a form that COG extractor could use.

The intermediate document tree is stored in memory while the user selects the area of interest. The UI part of the plug-in interrogates this tree to highlight the bounding boxes of the graphical objects selected (visible in Figure 6.1).

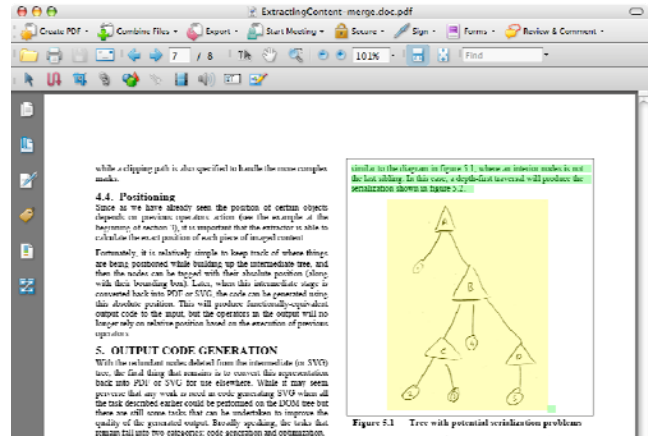


Figure 6.1 — Selected Content in COG Extractor

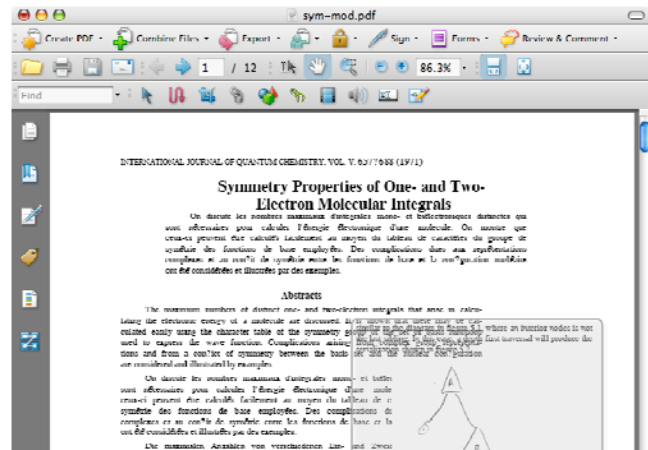


Figure 6.2 — Content inserted into new document

Once the correct area is selected, the user can push the extract button, which then copies the selected content into a new document, as a COG, in either PDF or SVG format.

Internally, the data structure storing the intermediate tree accepts a Visitor object that each node calls in the process of passing details of its data. This arrangement allows for export modules to be written for a number of different formats (currently both PDF and SVG) and for the output format to be switched simply by passing a pointer to the object implementing that particular export module. This method is also used to implement the UI, that is, it is also implemented as a Visitor object passed to the tree but in this case, rather than exporting the tree, it highlights the content on screen.

## 7. CONCLUSIONS

Content Extraction from layout-oriented documents is an important tool that allows content from legacy documents to be used in a VDP environment. However, as the discussions above



show it is not entirely straightforward and care has to be taken at every stage to produce high-quality output code and to ensure that correct appearance is maintained. But once the appropriate steps have been established the methodology used can be applied to other problems (such as format transcoding).

Understanding the nature of the content extraction problem has also highlighted the strengths and weaknesses of the different types of PDL. Imperative PDLs, such as PostScript, are able to use the power of parameterized function calls to create compact documents that produce high-quality output. However, this dynamic nature means that the output of the document is uncertain until the document is viewed and ideally it needs to be processed to a static final form before any document processing can safely be carried out. By contrast, PDF, being declarative, is already in final-form and so can be processed immediately. However, the linear nature of PDF makes certain document processing tasks (including content extraction) complicated and an explicitly hierarchical structure, as found in SVG, is a bonus.

There is a perception in some quarters that XML notation makes everything easier to work with but we need to take a balanced view. A hierarchical representation is helpful for many document processing tasks, but we have shown that it is very easy to map PDF into a hierarchical structure, at which point SVG's hierarchical advantage is matched and its shortcomings (in resource handling) become apparent. Indeed, as always with XML-based notations, the reason SVG seems simpler is twofold: firstly, the format forces the user, or the generating application, to explicitly express the containment of properties; secondly, the ready availability of parsers for XML prevents users having to write their own. But then, once the initial working parser has been created this can also be the case with PDF. The authors have already reused major chunks of COG Extractor (and its intermediate document tree) in other programs, as an easy way of creating PDF documents.

Functionally both SVG and PDF are capable of modelling the graphic-inheritance and resource-handling characteristics of documents well enough for content extraction. That being said, SVG still has some way to go before it develops into a full PDL with the capabilities of PDF. However, the choice of which PDL to use may also be influenced by the context in which it is being employed. As a part of an all-XML document-processing pipeline, it makes sense to use an XML-based language such as SVG, while if the documents are going to be distributed widely then PDF makes sense due to its ubiquity. Certainly, there are no programmatic reasons to favour one format over another as the problem of content extraction has shown.

## 8. ACKNOWLEDGEMENTS

The authors would like to thank Hewlett-Packard (UK) for their support in this research via funding for Steven Bagley. Thanks are also due to EPSRC (in collaboration with HP-UK) for the award of an Industrial CASE PhD studentship to James Ollis.

## 9. REFERENCES

- [1] Adobe Systems Incorporated, *PostScript Language Reference Manual*, Addison-Wesley, February 1999. Third edition.
- [2] Adobe Systems Inc, *PDF Reference (Third Edition; PDF 1.4)*, Addison Wesley
- [3] SVG 1.2 – Multiple Pages. <http://www.w3.org/TR/2004/WD-SVG1220041027/multipage.html>
- [4] HP Indigo. <http://www.hpl.hp.com/news/2006/janmar/indigo.html>
- [5] 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 (DocEng05)*, pp. 32–41, ACM Press, November 2005.
- [6] John Lumley, Roger Gimson, and Owen Rees, "Extensible Layout in Functional Documents," in *SPIE/EI 2006 Digital Publishing Conference*, January 2006.
- [7] PODi, *Print markup language functional specification version 2.1*, June 23 2003. <http://www.podi.org>
- [8] 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
- [9] Alexander J. Macdonald, David F. Brailsford, and Steven R. 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.
- [10] Steven R. Bagley, "COG Extractor," in *Proceedings of the ACM Symposium on Document Engineering (DocEng'06)*, p. 31, ACM Press, 10–13 October 2006.
- [11] S. G. Proberts and D. F. Brailsford, "Substituting outline fonts for bitmap fonts in archived PDF files," *Software — Practice and Experience*, vol. 33, no. 9, p. 885–899, July 2003.