Enhancing composite Digital Documents Using XML-based Standoff Markup

Peter L. Thomas
Document Engineering Laboratory
School of Computer Science
University of Nottingham
Nottingham, NG8 1BB, UK
plt@cs.nott.ac.uk

David F. Brailsford
Document Engineering Laboratory
School of Computer Science
University of Nottingham
Nottingham, NG8 1BB, UK
dfb@cs.nott.ac.uk

ABSTRACT

Document representations can rapidly become unwieldy if they try to encapsulate all possible document properties, ranging from abstract structure to detailed rendering and layout.

We present a composite document approach wherein an XML-based document representation is linked via a 'shadow tree' of bi-directional pointers to a PDF representation of the same document. Using a two-window viewer any material selected in the PDF can be related back to the corresponding material in the XML, and vice versa. In this way the treatment of specialist material such as mathematics, music or chemistry (e.g. via 'read aloud' or 'play aloud') can be activated via standard tools working within the XML representation, rather than requiring that application-specific structures be embedded in the PDF itself.

The problems of textual recognition and tree pattern matching between the two representations are discussed in detail.

Comparisons are drawn between our use of a shadow tree of pointers to map between document representations and the use of a code-replacement shadow tree in technologies such as XBL.

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

XML, PDF, standoff markup, composite documents, MathML, MusicXML, XBL

FINAL DRAFT of paper accepted for:

DocEng'05 November 2–4, 2005, Bristol, UK
Copyright 2005 Thomas and Brailsford

1. INTRODUCTION

In two previous papers [1, 2] we have set out techniques for using XML templates as a guide for embedding customised structure into PDF files. A Structured PDF file usually marks up its PDF content with the Adobe Standard Structured Tagset (SST). The SST Tags (see [3]) are roughly equivalent to those in HTML in that they denote document components such as paragraphs, titles, headings etc. Unlike XML, it is not possible to directly intermix SST tags and PDF content. Instead, the tags comprise the nodes of a separate structure tree and each of these nodes contains a pointer to a linked list of *marked content* within the PDF itself. Some basic facilities exist for creating PDF Structure trees where the nodes are custom tags, rather than tags from the SST. By using a system of *role mapping* one can indicate within the structure tree that a custom tag such as `PARAGRAPH` equates, say, to the `P` tag in the SST.

The addition of either the SST, or customised structure, to a PDF file does, of course, involve creating a new version of that file. In some circumstances this is not allowable — the file may have been saved in such a way that alterations to it are not permitted. Such a 'read only' property might, for example, be associated with the document having been digitally signed for authentication purposes.

It had always been our intention to develop structure insertion methods for 'read only' PDF documents by externalising the PDF structure tree and using it to point into the PDF content. This technique is often called *standoff markup* .and we hoped to enhance our Acrobat plugin to read in an externalised representation of the PDF structure tree and have it be displayed in the Acrobat document bookmarks, exactly as if it were a conventional internal PDF Structure tree.

It soon became apparent that much of what would be needed in the external structure tree was already present in the XML representation of the document as used in our two-window viewer and so, rather than replicating this structure in yet another tree, it seemed logical to investigate whether a hidden 'shadow tree' of pointers could be used as an intermediate between the XML and PDF representations of the same document. In effect this leads to a composite document format, where XML and PDF versions of the same document co-exist, and where the shadow tree of pointers, in conjunction with the XML, acts as standoff markup into the PDF.

One of the principal aims of our research has always been to invest Acrobat documents with structure so that 'added value' could be achieved in terms of enhanced accessibility for visually-impaired users and also in the areas of intelligent structural searching of PDF files and the flexible reuse of PDF documents. All of these aims are aided by knowing exactly what each PDF object represents in structural terms. However, placing customised structure trees inside PDF files is an awkward process and, even when it has been completed, further work might still be necessary if document accessibility is to be enhanced. To take a specific example let us consider the reading aloud of mathematical material rendered via PDF. Recent releases of Acrobat incorporate a read aloud facility but it comes badly unstuck when faced with 'difficult' material such as mathematics, chemical formulae or music. In all of these cases the placement of symbols requires many horizontal and vertical movements, interspersed with line drawing commands to render staves, fraction bars, chemical bonds etc. The process of reading aloud may then degenerate into the almost random recitation of the few isolated characters that can be recognised. Moreover, there is no guarantee that these characters will be emitted in anything remotely corresponding to a correct reading order.

These varieties of 'difficult' material cry out for a read aloud facility that can revert back to some more abstract, and yet more recognisable, representation. At the end of a previous paper [2] we showed a mathematics example in which we embedded customised structure — roughly corresponding to a MathML description of a typeset formula — into the PDF file. Unfortunately there are no standardised facilities in PDF to divert read aloud into reading out the tags within the structure tree whenever difficult material is encountered. Instead a subject-specific "alternate reading" has to be provided and our previous work forced us to embed that also.

These examples we have just cited made us increasingly convinced, as our work progressed, that instead of building customised structure into a PDF file it might be a much better strategy to have an accurate method of two-way cross correlation between XML and PDF versions of a document and to then rely on the ever-increasing number of XML tools to add value to the PDF. Thus, for example if our piece of mathematics were highlighted in the PDF and if this could be matched to equivalent MathML markup *in the XML source file* then it would be possible to deploy a MathML 'read aloud' tool to interpret the mathematics rather than relying on the embedding of structure-based 'alternate readings' in the PDF itself.

2. COMPOSITE DOCUMENT FORMATS

The idea of having two or more representations of a document (XML/DocBook and PDF in our case) co-existing in a single notional container, harks back to the days of Apple's OpenDoc framework [4]. OpenDoc envisaged a document being composed of material contributed from a variety of sources such as MacWrite, Adobe Photoshop, Adobe Illustrator and so on. Each piece of material in the OpenDoc document would be rendered by calling on the appropriate application at the appropriate time. If the document were sent to a remote machine, not having all of the required application programs, then a system of lower-quality rendering via bitmap approximations came into play to ensure that the document could at least be read. In many ways OpenDoc was well ahead of its time but it floundered because of the need to have a wide variety of authoring applications available and the effort needed to make each of these applications be 'OpenDoc aware' in order for them to fully participate in the framework. Even so, it is worth noting that OpenDoc documents were composite in an 'intra-document' sense of a sequence of fragments — there was never any idea of having two or more complete and consistent representations of the same document, but in different formats, cross-linked to one another.

Another application of composite document formats was ODA (Office/Open Document Architecture). The aim of ODA was to allows for blind exchange of office documents, containing both structural and layout information within the same file. ODA proved to be limited since the all layout and logical structures were explicitly specified by the standard, which limited the number of possible valid document structures. The logical stucture was limited to a 'numbered sections' approach, which included footnotes but excluded common elements such as lists and tables. Layouts were also limited, for example overlapping frames were not permitted. Finally, it was implemented at the syntactic level in ASN.1, a binary notation which was widely regarded as being difficult to read, which further hindered ODA's uptake [5].

In many ways the work that is closer in spirit to what we want to achieve is seen in Phelps and Wilensky's ingenious Multivalent Browser [6] This single browser allows a wide variety of document formats to be interpreted and it achieves this by extracting and characterising a set of 'behaviours' which are independent of any particular document format. The software is driven by extracting the essence of an abstract document tree from a supplied document in any of the 10 or so supported formats. Once this has been done a range of extra generic annotations and links can be added and displayed. Although the multivalent browser does not directly support cross-representational linking in quite the manner we are about to describe there seems little doubt that such support could be retro-fitted without too much effort.

3. STANDOFF MARKUP

Our approach will be to use the XML source of our document in conjunction with a shadow tree of pointers, as a form of standoff markup into the PDF content. The principle behind standoff markup is the maintenance of a rigid separation between markup and content instead of mixing the two together as in a conventional XML document. The idea is certainly not new because its origins can be traced back to the separation of program code from data that is imposed by modern operating systems in conjunction with the computer hardware. In the world of hypertext and hyperlinking, systems such as XLink provide a means of separating hyperlinks into linkbases, which are separate from the material to which they refer [7].

More generally, standoff markup is finding favour within the XML community [8] because:

1. it enables multiple different markups to be applied to the same base material, and
2. (related to 1.) it enables 'overlapping hierarchies' to be handled.

These characteristics are vitally important in areas such as biomedicine where different structures need to be imposed on highly complex basic data [9] and also in biblical text studies where items such as quoted speech (which is normally contained entirely within a single verse) can sometimes break the hierarchical rules completely and stretch over several verses [10]. Multiple standoff markup trees can handle both of these situations and we foresee a similar advantage, in our work, of eventually being able to impose multiple interpretations onto the same given PDF material.

4. RELATIONSHIP TO XBL

It is useful at this stage to point out also the resemblance that our shadow tree of pointers and 'landmarks' (see later sections) bears to the XML Binding Language (XBL) [11,12]. The idea of XBL is that a given XML source document can have alongside it a shadow code-generation tree which represents the details of the translation of the source tree into some other format (e.g. HTML or SVG). Now, it might validly be objected that an XSLT script can achieve exactly this sort of transformation but the problem is that the emitted code appears as a monolitihic stream with no indication of which part of the XSLT internal DOM generated which part of the output code. By externalising the code-generation templates one can adjust details of the code generation process without needing to re-run an entire XSLT transformation The drawback, unfortunately, is that the XBL sub-tree is just a simplistic set of code generation templates and it is not possible to invoke these templates in a context-sensitive way. If the source tree is traversed in the normal depth-first, left-to-right way, then its nodes can invoke code bindings in the code templates which create a *shadow tree* representing the result of translating the source tree into some output format such as HTML or SVG. A variant of XBL has been used to control rendering in the Mozilla/Firefox browsers and an SVG-specific version of XBL (s-XBL) is under active development [13].

We now recognise that our shadow tree of landmarks and pointers, to be described in detail in subsequent sections, is in some ways similar to the result of an XBL mapping. The difference is that instead of the shadow tree containing templates to generate PDF it now contains *pointers* into a PDF file that has been produced and rendered independently of the XML source document. But with an appropriate choice of pointer representation a flexible two-way correlation becomes possible between major corresponding features of two entirely different representations of the same document.

5. PREPARING THE TEST DOCUMENTS

In what follows we assume that a document in some popular authoring application such as MS-Word or $\mathrm{L_AT_EX}$, can be processed in two distinct ways: firstly to produce an equivalent version of the document in an XML-based markup (this may often be XHTML but for our examples here we shall be using DocBook tags) and secondly, via tools such as PDFMaker or PDFTeX, to produce an 'appearance based' paginated version in PDF. The existence of the two representations in a single compound container allows us to use the XML version of the document, where the textual content will almost always be in the correct 'reading order', as a source of information in helping to determine the reading order in the corresponding PDF. (The reasons why reading order will generally differ from rendering order, in both PostScript and PDF, are set out in section 1 of reference [2]). Once reading order has been established this is a major step along the way of setting up our system of landmarks and pointers.

6. TREE MATCHING AND STANDOFF MARKUP POINTERS

To demonstrate stand-off markup for PDF documents we developed a new Acrobat plugin. Some of the principles used are similar to those in the structure insertion plugin described in [2] but it was soon found that the more integrated composite document we were developing —with its two-way shadow tree of pointers —demanded a thorough re-engineering of our approach. An Acrobat plug-in gives access to the currently loaded PDF documents via an API [14]. This allows the plug-in to manipulate the contents of the PDF and to provide enhanced functionality of some sort. To a limited extent it is also possible to modify the Acrobat interface itself.

6.1.1. Referencing content in an unstructured PDF

By definition we wish to correlate a relatively abstract XML document with an equivalent unstructured PDF document, with the constraint that the PDF document may not be modified in any way. Perversely, if our read-only document just happened to be a Structured PDF we could simplify the correlation process quite considerably. Structured PDF documents do not explicitly embed their tags within the PDF content but, instead, they use a system of embedded Marked Content Identifiers (MCIDs) [3] and a separate structure tree. The structure tree references these MCIDs in order to demarcate content in the PDF. In some ways this could be regarded as being similar to standoff markup since the structure tree is separate from the content itself; it does however require appropriate MCIDs to be inserted so that 'back pointers' can be created and this contravenes the spirit of standoff markup i.e. that it should leave the content it points to unaltered. Since a well-formed structured PDF tree is guaranteed to give a correct reading order for the PDF file, it follows that these MCIDs could be utilised by a stand-off markup engine as a foundation for referencing content within the PDF. Unfortunately the real strength of standoff markup is precisely for those documents which are already unstructured and to which internal structure *cannot* be added.

6.2. Nature of shadow tree pointers

An immediately obvious solution to the nature of shadow tree pointers would be to refer to the material via a content stream and a series of byte offsets into that stream which encompass the desired material. This would have the advantage of being totally accurate when applied to the exactly correct PDF document. Unfortunately, this detailed exactitude leads to a loss of flexibility because it ties us down to precisely one PDF document that produces the given appearance in a certain way. Now, the visual appearance of PDF documents can be considered as being the result of a graphics state which is built up by the operators and operands contained within the content stream. There are a multitude of ways in PDF for generating any one particular graphics state but provided this final state truly is identical then all of the PDF documents, produced in all these different ways, should deliver an identically rendered appearance

If we now imagine generating a set of byte offsets to bind an XML document to a particular PDF document then it is clear that modifying the way in which the graphics state was achieved would result in these byte offsets being completely invalid. Worse still, there would be no way to even partially recover from this calamity because the byte offsets would be all that one had to work with. There are many operations in Acrobat which can cause the content-streams to be re-written, the most fundamental of which is the automatic optimisation that takes place on a PDF file when the 'Save As' operation is performed by the user. Likewise, any use of the Acrobat TouchUp tool would cause the content streams to be re-written, thereby invalidating the offsets.

6.3. Document Landmarks

In what follows we shall investigate the use of 'document landmarks' as points of reference within the PDF document. These landmarks attempt to reflect the human perception of the significant features in the document layout and our system attempts to recognise precisely these features. Once a set of landmarks has been computed for a particular document the rest of the content can be described in relation to them, thereby creating a layered method of addressing content in an arbitrary PDF document (Figure 1). One of the most critical properties of these landmarks is that for any document with a given appearance the same landmarks should always be calculated regardless of how that appearance was actually achieved.

6.4. Landmark Calculation

As a pre-requisite to the binding process, a set of 'document landmarks' has to be computed. In order to determine these landmarks, we need to perform document analysis on the PDF file.

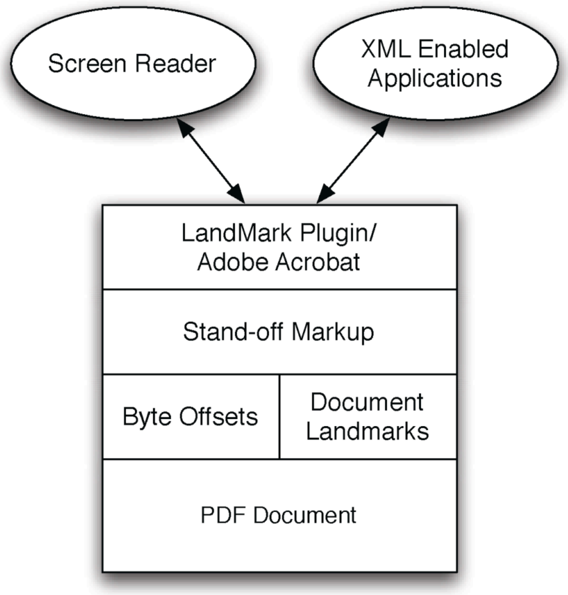


Figure 1: The layering model of document landmarks

The application of document analysis and understanding techniques to PDF files was begun by Lovegrove and Brailsford [15] in 1995. References 4 – 9 in that paper refer to previous research, in which many useful techniques were developed as a result of performing document recognition on bitmap files.

It might seem bizarre that document analysis is needed for the understanding of properly typeset, PDF Normal, files given that the entirety of the text will be present in the file. The problem is that PDF (often as a result of the PostScript from which it is generated) will often have a rendering order that is very different from the desired reading order. This makes extraction of material very difficult from PDF document sets such as newspaper archives, where complex layouts will probably have been used [16].

So, given that an unstructured PDF document may have no explicit reading order, except for that perceived by the user, a reading order has to be imposed on the document. This process can be roughly broken down into three sub-processes: blocking, block classification and block ordering. For our present purposes a block is defined as a visually contiguous piece of semantically similar material, e.g. a paragraph or a title.

The blocking process is a form of connected component analysis that attempts to group adjacent, semantically identical material in the document. Previously [2] we used a relatively simple, clustered X-Y sort algorithm, over each page, to reduce the page content to a set of lines, which in turn were a set of words. This was highly effective on relatively simple documents but when applied to more complex documents certain shortcomings became apparent. Chief among these was a tendency for 'gutter-hopping' when applied to a multi-column document. In other words if the inter-column gutter was rather narrow it might not be recognised as a page feature and the inferred reading order would then be left-to-right across the whole page rather than going down each column in turn. Clearly a more sophisticated method of content clustering was necessary.

6.4.1. Caching & Abstraction of Content

When analyzing the PDF document it was necessary to pass over the content of the document many times. In order to optimize performance by minimising the number of potentially expensive calls to the PDF document API, we decided to cache results in our own content model. This resulted in the creation of an abstract model in which to wrap up the Acrobat API methods and function calls.

The first stage of the landmark setup process was to construct the content model. The PDF was iterated over using the PDE layer of the Acrobat API. By iterating over the PDEContent and PDEElements, we can extract all the text-runs from the document. Text runs are represented as PDEText structures under the Acrobat API and they have a 1:1 mapping with the Tj [3] operators in the content stream of the document.

Each text-run is examined, and the following data is extracted:

1. Bounding box co-ordinates
2. Typographical information
3. Textual content

A virtual text-run is then created in the cache system, which wraps the PDEText object and stores all the extracted data for future rapid access.

Text runs in PDF are not guaranteed to contain entire words, neither are they guaranteed to contain explicit space demarcation, since a text-run maps directly to a Tj operator (analogous to PostScript's show operator) in the content stream

Each text-run in the PDF document may be represented by more than one virtual text-run in our content model. Text runs are examined for white space content. If a run contains white space, it is represented by a single virtual text-run for each contiguous span of non-white space content.

It is necessary to discard inter-word spacing information altogether given that there are so many ways in which word-spacing can be achieved in PDF. For example, explicit space characters can be used but, equally, word positioning can be achieved by a translation of co-ordinates completely external to the text-runs. This means that explicit white space information may or may not be present, and even if it is present, it cannot be relied upon. Instead of trying to interpret the spacing from the PDF, we used the XML file as our canonical reference for inter-word white space by modifying the approach already described in [1].

6.4.2. Statistics Gathering

As each virtual text-run is constructed, statistics are built up from the document. This information is later used by the classification engine to classify the content blocks. Examples of statistics that are gathered include:

1. Font-size – (min, max, mode, mean)
2. Typeface – (mode)

6.4.3. Page Segmentation

The pages of the document must be segmented into blocks, this is achieved by performing structure analysis upon each page. Document structure analysis is an established field, mainly with respect to raster graphics, however efforts have been made to perform structural analysis on symbolic layout such as PDF [15,16].

Two features were identified as being critical to this work; the algorithm must cope with multi-column layouts and must also be relatively fast. We chose to use a modified Run Length Smoothing Algorithm (RLSA) [17]. The original purpose of RLSA was to separate text from graphics in bitmap images. This was then adapted to obtain a bitmap consisting of black and white areas, which represented different types of data. In essence, this algorithm transforms an input bitmap X into an output bitmap Y by evaluating the following rules:

1. 1 bits represent the presence of data. 0 bits represent the absence of data.
2. 0 bits are transformed into 1 bits in Y when there are fewer than C adjacent 0s (C is a predefined constant).
3. 1 bits in bitmap X are also 1 bits in bitmap Y

This process is often known as a *blur* and it has the effect of linking black areas that are separated by a gap which is less than C. This blurring process is performed in both the X and Y directions, the results of which are logically ANDed together. A further horizontal blur is then performed upon this resultant bitmap. Different values of C are applied in different directions. The result of this process is a bitmap representing a set of distinct blocks of content.

To adapt this clustering method for use with PDF documents we generate a small bitmap representing the page being blocked. This bitmap has the RLSA process performed upon it and the results are then mapped back to the PDF content. This procedure is fundamental to the blocking process; it is critical that it is as efficient as possible since it will be run on each page of the document.

It was determined that plotting the bounding box of each text-run on a given page gave sufficient accuracy for the blocking process, while maintaining the desired performance.

Notice that there is indeed a potential performance impact, because document structure analysis is relatively costly. In a production system it might be performed in a just-in-time fashion, or in the background as the user performs other tasks.

6.4.4. Separation of Blocks

After the RLSA process is completed for a page, we can use the resultant bitmap to partition the page content into separate blocks. The first step in this procedure is identifying each block in the bitmap. This is achieved by the use of a fast-fill algorithm which assigns each disjoint area a different integer value. This integer value becomes the block's id with respect to the page. The cached page content can then be quickly iterated over and the block id determined by relating the co-ordinates to a bitmap block. The page content is then grouped by block id. Within each block, an X-Y sort is performed to produce a reading order.

So far we have examined only the geometric properties of the page content. These properties are enough to determine paragraphs and columns but will often fail to pick out in-line features, such as in-line emboldening etc. At this stage, we needed to split the blocks further using typographic analysis. Each block was examined and split into sub-blocks consisting of adjacent content with matching typographic properties. These sub-blocks became child blocks of the parent block.

6.4.5. Block Classification

The next stage in the landmarking process was to assign a type to each block of page content. We use the statistics generated during the caching of the text-runs to determine the most likely candidates for each type of block, based upon the typefaces in use and their frequency of occurrence in the document. During the initial phase we recognise the following types of block, which are analogous to their HTML namesakes: H1, H2, H3, P

The final stage of block preparation is for certain blocks to be classified as possible artefacts. By this we mean that certain characters may appear fewer times (or not at all) in the XML file compared to the PDF. Examples include auto-generated numbers in listings, page numbers and running header/footer material. At present, two forms of artefact reduction are employed. Numbers are removed from the start of lists by iterating through the blocks. When adjacent, non-inline, blocks are traversed which begin with numbering, the number is removed. Secondly the first and last blocks of each page are evaluated to determine whether they are either a page number or a running header, by comparing them to the first and last blocks of other pages. These artefact blocks are marked as such to aid the landmark identification process.

6.4.6. Selection of Landmarks

After the blocks have been finalised, we can decide upon the landmarks for a given document. It is at this stage we need to decide on the granularity of the landmarks. What level of granularity is acceptable for an accurate binding of the XML to the PDF? At one extreme, we could bind each and every character (and even the white space) in the PDF using document landmarks for each individual glyph. At the other extreme we could simply bind the XML and PDF documents at the level of their root nodes.

A sensible middle ground between these two extremes needs to be established. Binding every character is clearly over elaborate: the matching would be highly accurate but very inefficient in terms of the amount of space taken by the link file. Secondly it would not be an elegant or a robust matching strategy in the face of small document alterations. The resulting link file would also be very difficult to manipulate for use in different contexts.

The blocks that are created can be split into two categories; inline and non-inline. For our initial investigations we have taken non-inline blocks as our set of landmarks.

6.5. Anatomy of a Landmark Reference

Although we are attempting to make this work as tagset-agnostic as possible, we do assume certain things about the nature of the input XML document: the document content is assumed to be in reading order and stored as `#PCDATA` inside elements, rather than as attribute values.

With this in mind, we set out to produce a method of linking from our application-specific markup (the XML file) and the document landmarks to the underlying content in the PDF.

The basic premise of this method is that a structurally similar duplicate tree of the XML document is created, analogous to a shadow tree created by using XBL. This duplicate tree contains references to the document landmarks, rather than the original `#PCDATA` content. A similar method was proposed by the TEI Workgroup on Stand-Off Markup for implementing stand-off markup within TEI documents. TEI Standoff markup [18] uses the XInclude syntax for encoding references [19].

XInclude, in its standard form, was not suitable for our purposes because of its relative verbosity and the sheer number of references that will be included in an average PDF file. The referencing system we employ has similar semantics to those of XInclude, but it is significantly less verbose.

Examples of XInclude and our own LandMarkInclude might be (respectively):

```
<xi:include   href="b.xml"   xpointer="/p[2]"
parse="pdf"/>
```

```
<land:inc ref="/p[2]"/>
```

We use a simplified XPointer-like syntax for referencing the content in the document via the landmarks.

To avoid ambiguity, `ref` and `endref` must both be a location-set consisting of a single location. Landmark references are always inserted in the Landmark namespace to differentiate them from the markup of the document.

Attribute	Purpose
`ref`	The `ref` attribute of the include element contains a child sequence with an optional predicate. (Compulsory)
`offset`	Specifies an offset for the selection in characters from the start. (Optional)
`endref`	Specifies a child sequence which terminates the selection. (Optional)
`endoffset`	Specifies an offset into the terminating node in characters. (Optional)
`chars`	Number of characters in the XML that this rule maps to in the PDF (Optional)

Table 1: Landmark Reference attributes

Recall that space characters are discarded and offsets are measured in characters (excluding white space) from the node specified, with white space excluded. In some cases, the mapping between the characters in the XML and the PDF may not be exactly 1:1. For example ligatures are widely used in high-quality typesetting and they do exist in Unicode [20] and so can be represented in XML. Unfortunately, it is very uncommon to see ligatures handled correctly in this way Instead, ligatures are generally inserted automatically by the text-processing package, rather than existing in ligature form in the source XML. Additionally, they may well be inserted into the typeset output using non-Unicode positions in some older font encoding such as Adobe Standard or Macintosh. Using the `chars` offset to specify the number of characters in the XML file allows us to represent these non-1:1 mappings.

Figures 2 through 5 show a simple example of an XML file, its corresponding PDF, the identified landmarks in the PDF and the representation of those landmarks in the shadow tree. It should be noted that Figure 4 is a notional representation of the Landmarks tree established within the Acrobat plugin — it is never actually externalised in the form shown.

```
<article>
    <title>Enhancing Composite Documents</title>

    <section>
        <title>Abstract</title>
        <para>Document representations can rapidly
        become unwieldy if they try  to
        encapsulate all possible document
        properties, ranging from abstract
        structure to detailed rendering and
        layout.</para>

        <para>We present a composite document
        approach wherein an XML based
        document</para>

    </section>

</article>
```

Figure 2: XML input file

Enhancing Composite Documents

Abstract

Document representations can rapidly become unwieldy if they try to encapsulate all possible document properties, ranging from abstract structure to detailed rendering and layout.

We present a composite document approach wherein an XML-based document

Figure 3: Resultant PDF output

```
<h1>Enhancing Composite Documents</h1>

<h2>Abstract</h2>

<p>Document representations can rapidly become
unwieldy if they try  to encapsulate all possible
document properties, ranging from abstract
structure to detailed rendering and
layout.</p>

<p>We present a composite document approach
wherein an XML based document</p>
```

Figure 4: Landmarks calculated from the PDF in Figure 3

```
<article>
    <title><land:inc ref="/h1[1]"/></title>

    <section>
        <title><land:inc ref="/h2[1]"/></title>
        <para><land:inc ref="/p[1]"/></para>
        <para><land:inc ref="/h1[2]"/></para>
    </section>
</article>
```

Figure 5: Landmark references

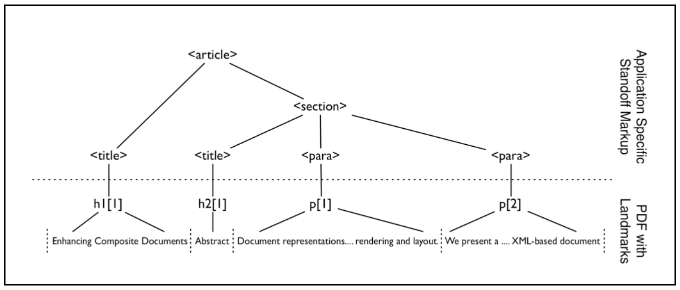

<article>
<section>
<title>
<title>
<para>
<para>
h1[1]
h2[1]
p[1]
p[2]
Enhancing Composite Documents
Abstract
Document representations.... rendering and layout
We present a .... XML-based document
Application Specific Standoff Markup
PDF with Landmarks

Figure 6: Tree representation of landmark references

6.6. Comparison of PDF & XML

6.6.1. Landmark Reference Authoring

The first stage in the process of using standoff markup with the LandMark system is to relate the XML document to the PDF document and to generate the relationships between the landmarks and the application-specific XML markup. This is a procedure performed just once for any particular document/XML pair and the results are serialised to a landmark references file.

To perform this task, the PDF file is loaded into Acrobat and the user selects the menu option: 'Generate LandMark References'. This brings up a file requester that allows the user to select the appropriate XML file for the comparison. This XML file is then loaded into a DOM using Apache Xerces.

The DOM tree is traversed with a pre-order traversal, using a recursive depth-first tree-descent algorithm. This causes each node to be processed in the same order that it would appear in its serialised XML form.

A recursive function is called which processes the children of the root DOM node. As each node is visited in the DOM tree, its type is evaluated and, dependent on the outcome, various different processes are performed. XML elements cause the function to recursively call itself for that element. In the case of text, the content is processed further. In the same manner as previous work [2] string matching is performed upon the XML content to match it with the normalised, cached PDF content. When the best match is found, a landmark reference is created. When there is no more remaining material to be correlated, the landmark references tree is serialised to a file.

6.7. Binding the documents

Binding is the procedure that occurs when users require structural enhancement for their PDF document. Essentially this loads the XML document and Landmark References. A shadow tree of landmark references is then built up in conjunction with the XML document. The landmarks of the PDF document are calculated and the final step is to traverse the landmark references shadow tree to set up back pointers from the virtual text-runs to the landmark references.

6.8. Using the Landmarks

6.8.1. Looking up PDF from XML

A common use case which requires structure in PDF documents is the use of a screen-reader. The term 'screen-reader' is something of a misnomer, since the screen reader does not actually interpret the contents of the screen; it typically hooks into the parent application using an Accessibility Interface API such as MSAA (Microsoft Active Accessibility) under Windows. It is not possible to replace the interface between Acrobat and the screen-reader using the plug-in API but we can simulate some of the functionality that might be required.

The logical XML representation is generally more useful to the screen reader, since it is semantically richer than low-level PDF. The visual representation of a document (in this case the PDF content) is, to a blind user, merely a side-effect of the logical representation. In order to read the document aloud, the most efficient method would be to traverse the XML tree of the document.

Retrieving the PDF content that corresponds to a given part of the XML document is relatively straightforward. The sub-set of the Landmark tree that is equivalent to the selected content in the XML tree is iterated over to discover `<land:inc>` elements. Each of these elements is then used to retrieve the virtual text-runs from the page content cache. The bounding boxes of these virtual runs are extracted and used to construct a text-selection, using the PDTextSelect methods. Equally, instead of creating a text-select, other operations could be performed upon the PDF content.

6.8.2. Looking up XML from PDF

Users may want to export a selection of a document as its XML equivalent, for example they may want to extract the MathML [21] for an equation or play some music which has been encoded as MusicXML[22]. This requires a facility for relating PDF content to the equivalent XML content.

The first task is to calculate the set of landmarks that correspond to the selected content. The cache of text-runs is used to look up the appropriate landmark references that were calculated in the initial binding process. This set of landmarks is then trimmed appropriately to correspond to the content which is selected. It is important that the landmark references are as simple as possible, so that landmarks which can usefully be merged are indeed merged at this point.

We can then relate this to the XML by traversing the binding tree and following the back pointers from the landmarks to the referencing node in the stand-off markup and thence to the corresponding node in the XML document.

6.9. Specialised Markup

While the Landmark system is, in general, tagset-agnostic, that is to say it is not optimised for any one tagset in particular, there is a mechanism for feeding *tag-hints* into the system. Certain more abstract tag-sets are almost impossible to infer from a PDF document without detailed knowledge of their semantics – a prime example of this would be MathML.

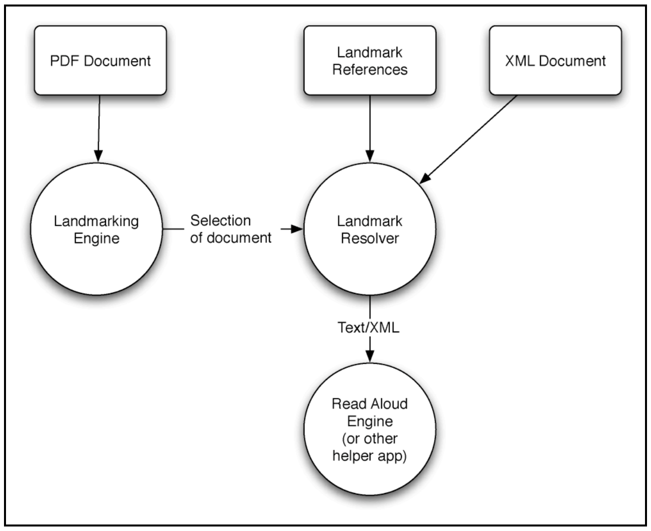


Figure 7: Enhanced functionality gained from stand-off markup

Without customised, tagset-specific, behaviour the mathematics seen in Figures 8 and 9 would be incorrectly structured. The `x`, `=`, `4`, `c` and `a` characters would be correctly structured, but the `&InvisibleTimes;` would fail to be recognised because it is only implicitly manifest in the PDF document by virtue of character adjacency.

In order to add structure to MathML typeset articles at the finest granularity, we would need to recognise the actual semantics of the mathematics from the PDF document alone and then bind it to the XML. Mathematics is notoriously difficult to recognise from typesetting information alone, as demonstrated by Suzuki et al. [23].

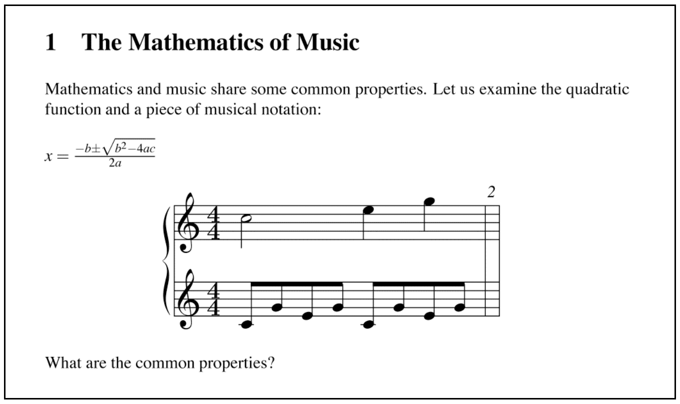
Since, in the initial instance, we are trying to avoid tagset-specific behaviour, we need some way for the system to be able to cope with 'problem' tagsets such as MathML. In this case, we would want our system to be able to identify the content within a certain bounding box of the PDF as being 'mathematics' and hence a suitable candidate for matching to MathML However if we look aside at the XML tree and see that blocks of material that we have already identified, in the PDF, lie before and after a MathML node in the XML tree then we can be reasonably sure we have found a correct match. However, we would then not expect any detailed matching of the complex PDF mathematical typesetting effects we see i.e. we would not expect the markup matching to descend any further than the top level `<math>` element in the XML file.). This still allows us to take advantage of standoff markup to provide added value to the document, such as export of the MathML to a separate file, a custom read-aloud or — in the case of MusicXML — the associated music could be played aloud.

In order to cope with these special cases, we have a separate configuration file that allows us to specify the specialised tagsets. These are identified by namespace. For instance, in the case of MathML, the configuration file specifies that references which bind MathML to PDF should not be any more granular than the `<math>` element but that the child `#PCDATA` under it can be used to produce a match with the PDF document.

In addition to (optionally using) the `#PCDATA`, in the case of specialised markup the Landmark Binding Engine can perform look ahead to the next element. If this element is not a special case, then the binding engine will bind that element and bind the unassigned content between that element and the previous element to the special case element.

In the case of multiple adjacent special case elements, some disambiguation must take place to distinguish between them. The configuration file also allows us to specify various weighted hints which allow us to perform the disambiguation, one such 'hint' allows us to disambiguate based on the typeface in use. Although, taken on its own this is not enough to differentiate between two elements of the same type it is certainly effective when the elements are of different types. Combining these weighting rules with the partial match from the `#PCDATA` allows us to disambiguate between elements of two different types as shown for MathML and MusicXML in figures 8 and 9.

Differentiation between elements of the same type must be based on the `#PCDATA` content and on clues from the boundaries with adjacent 'easy' material.



1 The Mathematics of Music

Mathematics and music share some common properties. Let us examine the quadratic function and a piece of musical notation:

$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

What are the common properties?

Figure 8: A sample document created from specialised markup

7. CONCLUSIONS

When we first contemplated completing our PDF structure insertion work by tackling standoff markup (see section 5.3 of reference [2]) we saw it as little more than a tidying-up exercise and a 'back door' method of emulating a Structured PDF file (via an external PDF structure tree) for base material that was actually unstructured and 'read only'. As a result of considering what form the 'externalised PDF structure tree' should actually take, it gradually dawned upon us that there were real advantages in using the XML tree itself as the repository of external structure coupled with an XBL-like shadow tree of pointers and landmarks to reference the corresponding PDF material.

We are now firmly convinced that the time is right to revisit the advantages of standoff markup and composite, multi-representational documents. The size overheads of carrying around two or more representations of a document are not too severe these days (especially if .zip containers are used). The gains are that each of the document representations can play to its major strengths — for example digital signatures and MD5 hashes make far more sense in a binary format such as PDF, whereas software to exploit structural markup is far easier to develop for XML than for Structured PDF.

```
<?xml version="1.0"?>

<article>
  <section role="section">
    <title>The Mathematics of Music</title>
    <para>Mathematics and music share some
    common properties. Let us examine the
    quadratic function and a piece of musical
    notation</para>
    <para>
    <inlineequation role="inline">
    <math
xmlns="http://www.w3.org/1998/Math/MathML">
      <mi>x</mi><mo>=</mo>
      <mfrac>
  <mrow><mo>&#x2212;</mo><mi>b</mi><mo>&#x00B1;</m
o>
  <msqrt><mrow><msup><mrow><mi>b</mi></mrow><mrow>
<mn>2</mn></mrow></msup>
  <mo>&#x2212;</mo><mn>4</mn><mo>&InvisibleTimes;
</mo><<mi>a</mi><mo>&InvisibleTimes;</mo><mi>c
</mi></mrow>
    </msqrt></mrow>

  <mrow><mn>2</mn><mo>&InvisibleTimes;</mo><mi>a</
mi></mrow>
      </mfrac>
    </math>
    </inlineequation>
  </para>
  <para>
  <score-partwise>

    <!-- MusicXML omitted for brevity -->

  </score-partwise>

  </para>
  <para>What are the common properties?</para>
  </section>
</article>
```

Figure 9: Specialised markup for mathematics and music

Our investigations so far have shown the feasibility of identifying blocks of `difficult' material such as mathematics and music by using simple document recognition techniques based largely on identifying landmarks in the easy-to-recognise text which precedes or succeeds the specialist material. As we have shown this strategy succeeds very well in matching features seen on the PDF display with the appropriate nodes in a structured XML representation.

We envisage a future where the simultaneous generation of an XML and a PDF representation into a common zipped container may well be commonplace. The advantages of the dual representation we are proposing are then immediately available. A far stiffer challenge lies in cases where only an unstructured PDF is available, with no XML equivalent. Text extraction followed by document recognition can establish a moderately plausible DocBook equivalent to simple textual material but as [23] indicates the inference of MathML from typeset mathematics is distressingly difficult, if not impossible, to achieve. Even so there is an argument which says that already much work has to be done by hand in making important documents more accessible for visually impaired readers. Having accepted this, it would be far easier to create tools to insert, by hand, into a DocBook tree, a MathML equivalent to typeset mathematics seen in a PDF, than it ever would be to insert that same custom structure into a Structured PDF

8. ACKNOWLEDGEMENTS

Thanks are due to Adobe Systems Inc. for supporting Peter Thomas during his Graduate Internship at Adobe in 2004 and also during his graduate studies generally. In particular we thank Bill McCoy, and Phil Levy of Adobe Systems Inc., for much administrative and technical help, and Loretta Guarino for technical information on PDF accessibility issues.

REFERENCES

[1] Matthew R B Hardy and David F Brailsford, ‘‘Mapping and Displaying Structural Transformations between XML and PDF,’’ in *Proceedings of the ACM Symposium on Document Engineering (DocEng’02)*, pp. 95–102, ACM Press, 8–9 November 2002.

[2] Matthew Hardy, David Brailsford, and Peter Thomas, ‘‘Creating structured PDF files using XML templates,’’ in *Proceedings of the ACM Symposium on Document Engineering (DocEng’04)*, pp. 99–108, ACM Press, 27–31 October 2004.

[3] Adobe Systems Inc, *PDF Reference (Third Edition; PDF 1.4),* Addison Wesley, 2002. ISBN 0201758393

[4] *OpenDoc Programmers’ Guide,* Addison Wesley Publishing Company, 1995. ISBN 0-202-47954-0

[5] Heinz Fanderl, Kristian Fischer and Jurgen Kamper, “The Open Document Architecture: from standardization to the market — Technical” *IBM Systems Journal* December 1992.

[6] Thomas A. Phelps and Robert Wilensky, ‘‘The Multivalent Browser: A Platform for New Ideas,’’ in *Proceedings of the ACM Symposium on Document Engineering (DocEng’01)*, pp. 58–67, ACM Press, 9–10 November 2001. Atlanta, Georgia

[7] David F. Brailsford, ‘‘Separable Hyperstructure and Delayed Link Binding,’’ *ACM Computing Surveys,* vol. 31, no. 4es, December 1999.

[8] Henry S. Thompson and David McKelvie, ‘‘Hyperlink semantics for standoff markup of read-only documents,’’ in *Proceedings of SGML Europe 1997*, May 1997. Barcelona, Spain

[9] Jung Ding and Daniel Berleant, ‘‘Design of a Standoff Object-Oriented Markup Language (SOOML) for Annotating Biomedical Literature,’’ in *Proceedings of 7th International Conference on Enterprise Information Systems (ICEIS)*, May 24–28, 2005. Miami

[10] Steven DeRose, ‘‘Markup Overlap: A Review and a Horse,’’ in *Proceedings of Conference on Extreme Markup Languages*, 2004.

[11] XBL W3C Note. http://www.w3.org/TR/2001/NOTE-xbl-20010223/

[12] W3C Comment on XBL Submission. `http://www.w3.org/Submission/2001/05/Comment`

[13] S-XBL Working Draft. `http://www.w3.org/TR/sXBL/`

[14] Adobe Systems Incorporated, *Acrobat Core API Reference.*, 2002. San Jose, CA: Adobe Systems Incorporated

[15] W. S. Lovegrove and D. F. Brailsford, " Document analysis of PDF documents: methods, results and implications." *Electronic Publishing, Origination, Dissemination and Design.* 1995, 8(2 and 3), pp. 207–220.

[16] Karin Hadjar, Maurizio Rigamonte, Denis Lalanne and Rolf Ingold "Xed: a new tool for eXtracting hidden structures from Electronic Documents" *Proceedings Document Image Analysis for Libraries* 2004, Palo Alto, California, January 2004, pp. 212-221

[17] F. M. Wahl, K. Y. Wong, and R. G. Casey, ''Block segmentation and text extraction in mixed text/image documents'' *Computer Graphics Image Processing*, vol. 20, pp. 375–390., 1982.

[18] Text Encoding Initiative Consortium, TEI Workgroup on Stand-Off Markup, XLink and XPointer [online], October 2004. `<http://www.tei-c.org/Activities/SO/>`

[19] World Wide Web Consortium, XML Inclusions (XInclude) Version 1.0 [online], December 2004.Available at: `<http://www.w3.org/TR/xinclude/>`

[20] Unicode Consortium, *The Unicode Standard: Worldwide Character Encoding, Version 1.0.*, Addison Wesley,, 1991. Vols. 1 & 2.

[21] World Wide Web Consortium, Mathematical Markup Language (MathML) Version 2.0 (2nd ed.) *[online].* Available at: `<http://www.w3.org/TR/MathML2/>`

[22] Recordare, *MusicXML Definition* [online]. Available at: `<http://www.recordare.com/xml.html>`

[23] M. Suzuki, F. Tamari, R. Fukuda, S. Uchida, and T. Kanahori, ''INFTY—An Integrated OCR System for Mathematics Documents,'' in *Proceedings of the ACM Symposium on Document Engineering (DocEng'03)*, pp. 95–104, ACM Press, 20–22 November 2003.