MAPPING AND DISPLAYING STRUCTURAL TRANSFORMATIONS BETWEEN XML AND PDF

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

ABSTRACT

Documents are often marked up in XML-based tagsets to delineate major structural components such as headings, paragraphs, figure captions and so on, without much regard to their eventual displayed appearance. And yet these same abstract documents, after many transformations and 'typesetting' processes, often emerge in the popular format of Adobe PDF, either for dissemination or archiving.

Until recently PDF has been a totally display-based document representation, relying on the underlying PostScript semantics of PDF. Early versions of PDF had no mechanism for retaining any form of abstract document structure but recent releases have now introduced an internal structure tree to represent the so-called 'tagged PDF'.

This paper describes the development of a plugin for Adobe Acrobat which creates a two-window display. In one window is shown an XML document original and in the other its tagged PDF counterpart is seen, with an internal structure tree that, in some sense, matches the one seen in XML. If a component is highlighted in either window then the corresponding structured item, with any attendant text, is also highlighted in the other window.

Important applications of correctly tagged PDF include making PDF documents reflow intelligently on small screen devices and enabling them to be read out in correct reading order, via speech synthesiser software, for the visually impaired. By tracing structure transformation from source document to destination one can implement the repair of damaged PDF structure or the adaptation of an existing structure tree to an incrementally updated document.

Keywords

XML PDF document structure transformation

1. INTRODUCTION

For 20 years or more the field of digital documents has been split into two distinct cultures, which can be broadly characterised as the 'layout based' and the 'structure based' approaches. The structure-based approach has drawn inspiration from SGML and XML and has focused on abstract transformation of document structure and the `multi-purposing' of marked up documents. By contrast, the layout based approach puts a heavy emphasis on the graphic richness of the final document and the exact details of pagination and layout; this approach has concerned itself with issues such as fonts, colour spaces and the capabilities of PostScript and PDF. The distinct lack of common ground between the two approaches is exacerbated by the fact that there truly is a large 'semantic gap' between thinking of a document in these two very different ways. Figure 1 below tries to summarise the current state of affairs, where it is now quite routine to use a single document authoring system, such as MS-Word, and to exploit different output options to proceed either to an XML-based world or to a 'layout based' world where PostScript and PDF are firmly in place as the *de facto* standards.

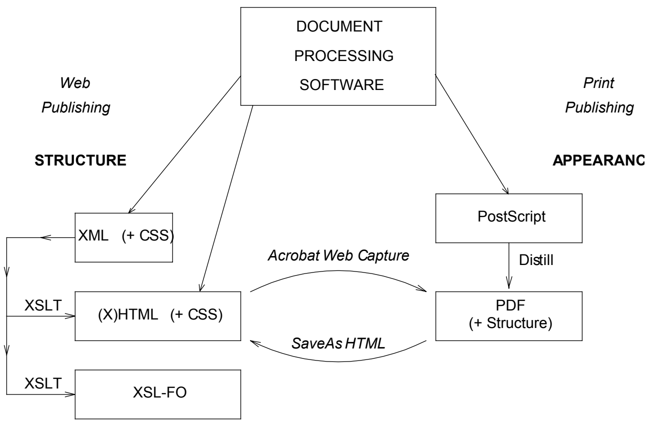

Figure 1. Document Processing Software.

In recent years the success of HTML has been seen in some quarters as a genuine synthesis of the 'structure' and 'layout' approaches and yet even a cursory examination of HTML tags shows that, for all that they are cloaked in SGML syntax, any notions of document structure have been largely abandoned in favour of achieving acceptable layout effects in browsers. To some extent this is now being addressed by allowing more general XML documents to be viewed in browsers and by styling the document either with Cascading Style Sheets (CSS) or via XSL Formatting Objects (XSL-FO) transformations.

However a glance at the right hand side of figure 1 reveals the other side of the story; PostScript and PDF have evolved, until very recently, with little or no regard to any abstract document structure. Worse still a PostScript or PDF page can often be rendered on screen in a way that bears no resemblance to any concept of 'reading order'. Indeed, it has been common for many years to render PostScript in a manner which optimises the performance of some target printing device. Thus, for example, the motion of a bromide sheet in a typesetting machine can often be minimised when typesetting a two-column article if the text is set in 'baseline sort' ordering i.e. by hopping across the inter-column gutter and setting fragments of sentences in turn.

More recently the vital importance of structure and correct reading order has been realised because of the increasing need to read out PDF files to the visually impaired and to be able to intelligently reflow PDF material for small-screen hand-held devices. Starting with release 1.3 of the PDF specification [1] there is now an option to include a structure tree in a PDF file. The tags within the structure tree can either be from a default tagset (at a level of functionality roughly equivalent to that of HTML) or they can be user-defined tags. Although the tree representation is in PDF syntax, rather than XML, it can still represent the structure of the document and the lower levels of this tree are pointers to imageable material within the PDF Page Content tree. Badly structured PDF, that is not in reading order, is handled by having the low levels of the structure tree point to a linked list of text fragments; traversal of this list recovers the text in reading order as opposed to rendering order.

Figure 1 makes plain that the world of XML is reaching out towards layout-based documents via techniques such as CSS, XSL-FO and even by introducing a low-level set of graphic primitives as an XML application (Scalable Vector Graphics — SVG). Conversely PDF is now reaching out towards the world of structured documents by having inside itself a structure tree. Nevertheless the gap between the two approaches is still very wide and tools are needed to help the chasm to be bridged.

The most direct way to tackle the problem of adding structure to a PDF file is to subject the pages in the file to document recognition techniques that try to identify features such as headings, paragraphs, footnotes etc. These techniques were first applied to PDF by Lovegrove and Brailsford [5] and, more recently, a similar approach, in Adobe's Make Accessible plugin for Acrobat, is usually successful in planting enough inferred structure within the PDF to enable it to be intelligently reflowed or to be read out loud, via a sound synthesiser program, in correct reading order.

But our concern is not just to add structure where none existed previously. We also want to correlate structure in a marked up structured document, which constitutes the starting point for a set of transformations including page layout and typesetting, with that appearing in the structure tree of the corresponding final-form PDF. The chain of transformations between the XML tagged starting point and the final PDF will often be very long. It is important that any structure verification and repair tool we create should rely solely on an XML starting point and the corresponding PDF finishing point; it should be independent of the particular 'typesetting middleware' that has been used to create the PDF.

2. ADVANTAGES OF AN EXTERNALISED XML REPRESENTATION

It is becoming increasingly common for academic and commercial publishers to require the creation of an XML version and a PDF version of their various documents. For archiving purposes there is then the comforting feeling that both structure- and layout-based versions will be available for future re-purposing; for more immediate use the PDF version can be used to create printing plates while the XML version can, for example, be processed with various XSLT scripts to create an XHTML document for the Web.

However, in all of this, it is easy to lose sight of the fact that almost never is there any check that the structure of the XML version matches up with any structure that might have been placed in the corresponding PDF. So long as the domains of structure- and layout-based documents did not overlap then any structural mismatch would be of little consequence, but as structured PDF becomes more common and the ways of creating it ever more varied, the need for correlating structure in XML and PDF manifestations of a document assumes a new importance.

Figures 2 and 3 illustrate two common ways for creating XML and PDF versions of a document.

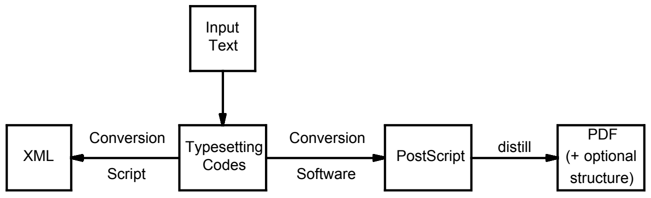

Figure 2. Creating a structured PDF file — the 'middle out' approach.

The first and perhaps more common of these can be characterised as the `middle out' approach of figure 2. One starts by keying in the document using a macro set or style file for the text-processing or typesetting application of choice (e.g. MS-Word, LaTeX, *troff*, Quark Express etc.). Proceeding to the right of the figure, the source text is then processed, via PostScript and Adobe Distiller, into PDF. Proceeding to the left of the diagram a separate conversion script (perhaps written in PERL or Omnimark) converts the input script and its macro codes into a corresponding XML document, tagged up in accordance with the publisher's Document Type Definition (DTD) or schema.

An alternative method is to proceed via the workflow shown in Figure 3 which can be characterised as the `top down' or 'left to right' approach.

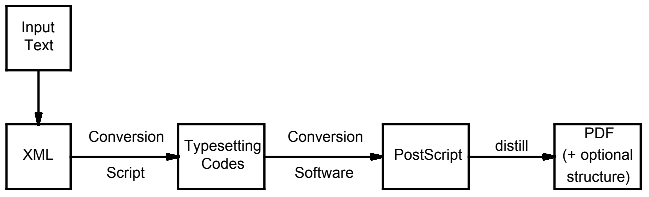

Figure 3. Creating a structured PDF file — the 'top down' approach.

Here the source text is entered using the publisher's XML tagset and a transformation script (again, this could be coded using PERL or XSLT) converts this text into a suitable input for the particular typesetting software in use. The question now arises as to whether this transformation script can somehow pass on the *structure* of the input text as well as the correct codes for typesetting it. If this is to succeed it implies that the typesetting middleware has to be persuaded to pass on the structure information into the PostScript output and in a suitable form to be converted into a PDF structure tree.

At this stage it is worth noting that PDF has always provided facilities for various forms of 'hyperstructure' (e.g. book marks, electronic sticky notes, links) but until fairly recently the anchors for placing these items were simply the low-level coordinates of bounding boxes to define the source or destination area on a particular (hard-coded) numbered page. From the earliest PDF definition a set of parameterised pdfmark operators for the various 'hyperfeatures' was defined in PostScript syntax. If these pdfmarks were correctly inserted into the PostScript output, from a given typesetting program, then it was easy to arrange for PostScript printer drivers to ignore the pdfmarks but for Distiller to seize upon them and convert them into the corresponding feature inside the PDF. This approach requires only that the middleware be capable of passing on, transparently and without question, the required pdfmarks. Fortunately most text processing software has a method of incorporating encapsulated PostScript for diagrams, photographs etc. and this can usually be subverted for purpose of passing on pdfmarks. This approach was pioneered in the CAJUN project [7] and has also been extensively used in software such as Adobe's PDFMaker, which converts MS-Word files into PDF.

So, if the middleware can pass on pdfmarks for the 'unstructured hyperfeatures' of PDF there is no problem at all in using the same mechanism for passing on the pdfmarks that correspond to the building of PDF structure trees. However, the main problem with this approach is that every piece of typesetting middleware becomes a new challenge in terms of finding ways to pass on pdfmarks.

One obvious solution to the problem is to combine the left hand two boxes in Figure 3 (Adobe's FrameMaker would be a good example of this approach) so as to devise unified authoring software that is XML aware and which can pass on the XML structure tree into an equivalent tree in the Postscript or PDF output. But given the sheer diversity of document creation software there is no chance of such a unified solution being universally acceptable.

Before we leave this section some further issues need to be addressed with reference to the 'middle out' and 'top down' approaches we have outlined. Firstly, a key problem with 'middle out' is that the left-going and right-going transformations shown in Figure 2 can all too easily lead to valid structures in XML and PDF which still differ in subtle ways, especially if the two transformation scripts have been written by different people. The 'top down' approach of figure 3 is in principle the more sound but even here the only way to strive for structural integrity is to insist that any structural or layout shortcoming in the final PDF should be addressed by totally regenerating the PDF file, starting with the XML. Tinkering with the middleware typesetting codes should definitely *not* be allowed. And yet any XML schema that tries to control typeset effects will probably be unwieldy; moreover reverting to XML to correct small blemishes on a final form PDF, ready to go to a platemaker, is a sure recipe for expense and frustration.

The inevitable conclusion is that, even when PDF files have been created from an allegedly structured starting point, the corresponding structure tree in the PDF could be either non-existent or at the very least damaged in some way with respect to the starting point. It is precisely this circumstance that our Acrobat plugin is designed to address. Its key advantage is that it enables starting and destination structure to be reconciled, independently of any particular typesetting middleware.

3. STRUCTURED PDF

The internal structure of a PDF is built up from a series of trees. Each of these trees represents a specific aspect of the PDF content, with the Pages tree representing the majority of the typeset content. The Pages tree contains a sequence of page nodes, each containing streams of data which specify the content for that page. These Page nodes can be logically grouped to share appearance properties. Within the streams of data belonging to each page are commands somewhat similar to those found in PostScript, containing the text, images, lines, etc. that are to appear on the page.

The PDF 1.3 specification added support for logical structure in PDF documents. The newly added PDF Structure Tree is designed to impose structure on the content of the document in a representation that is conceptually (but not syntactically) similar to that of XML and SGML. This tree is separate from the Pages tree, but contains links into the content of each page. It does this by inserting commands into the data streams of the page content. These commands demarcate separate areas of content belonging to the structure tree. Element nodes within the structure tree use these marked sections to indicate which groups of content logically belong to them.

This structured approach certainly aids in the repurposing of documents, but does not completely solve all the associated problems. The blocks of marked content pointed to from the structure tree do not have to be in reading order and there is no easy way to specify the semantics of the tagset chosen to represent the document structure.

The PDF 1.4 specification [2] addresses these issues by introducing a more standardised usage of PDF, for logically structured documents, called 'Tagged PDF'. Tagged PDF uses the logical structure as defined in PDF 1.3 and extends it to make it more useful for cases such as text extraction, reflow, conversion, and accessibility.

A Tagged PDF must conform to a set of rules, which allow the document to be more accessible. These properties can be separated into three categories:

1. Page Content
    - All represented text is in a form that can be converted to Unicode.
    - Word breaks are explicitly represented.
    - Actual content is distinguished from artifacts of layout and pagination.
    - Content must be given an order related to its appearance on the page.
2. Structure Types
    - Standard structure types are used within the structure tree to convey the semantics of the structure.
    - When using customised tagsets, these must be mapped to their closest equivalent standard structure types.

3. Structure Attributes
    - Standard structure attributes used to preserve styling information from authoring applications.

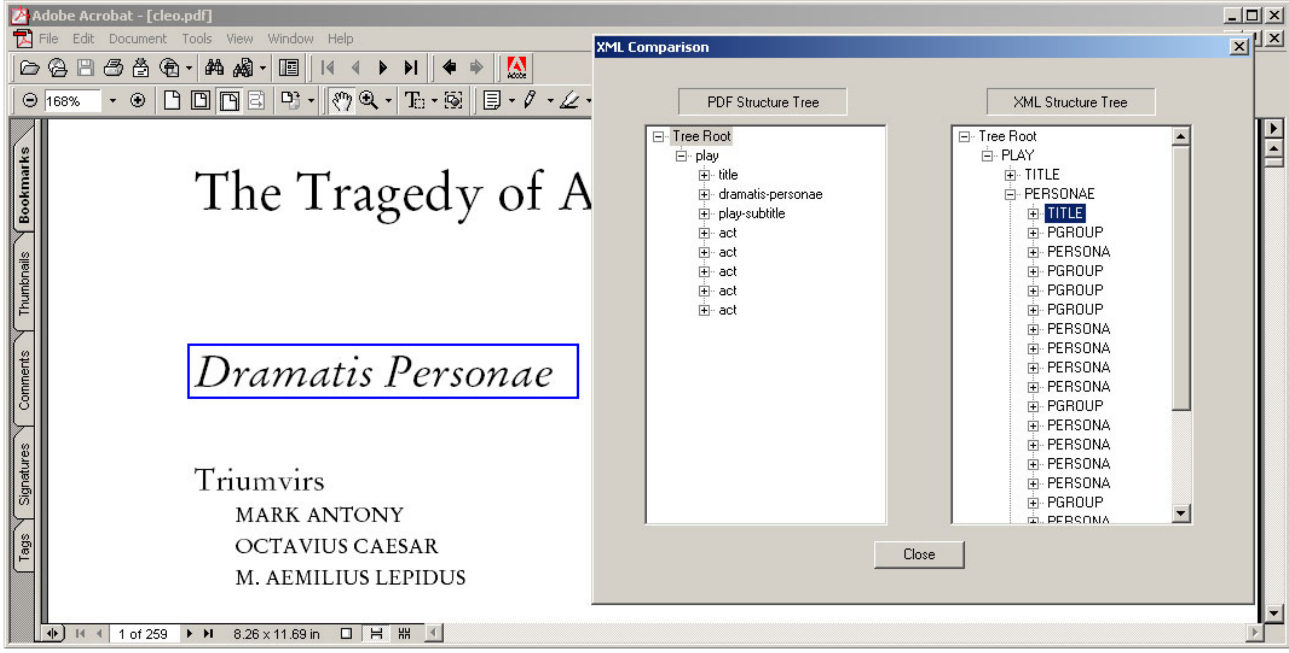

Figure 3. XML Comparison Plugin Usage

The standard tagset defined by Tagged PDF is very much aimed at document layout, and fast PDF to HTML conversion, rather than being an exercise in abstract document structure. In both Structured (1.3) and Tagged (1.4) PDF, a customised tagset can be defined and there is now provision for a *role-map* to provide a mapping between the custom tagset and the predefined Adobe standard tagset.

If tags other than the default are used, each custom tag must be mapped to the standard type that has the closest fit. This is done by using a dictionary to store key pairs representing the custom tag followed by the tag it is similar to. Table 1 contains examples of standard tags and their usage.

Table 1. List of Standard Structure Types

Tags	Usage
P, H, H(1-6)	Paragraph and Heading tags containing textual content.
L, LI, LBody	List tags describing a List, List Item and List Body respectively.
Table, TH, TR, TD	Table tags for display a Table, Table Headings, Rows and Data respectively.
Document, Art, Part, Sect, Div	Standard structure types used for grouping content.
Figures, Form	Tags representing figures and interactive form elements.

An example of a standard usage of role-mapping comes when using the Web Capture plugin provided with Adobe Acrobat. This plugin allows for a Web page to be captured to a Tagged PDF. Although the default tagset is similar to that of HTML, it is not identical, so a mapping is provided e.g. the 'OL' tag maps to 'L' and the 'I' tag maps to 'Figure'.

This mapping of tagsets allows applications using Tagged PDF files to understand the semantics of the custom tagset, making it easier to repurpose the documents.

4. XML COMPARISON PLUGIN

To facilitate the checking of logical structure within a Tagged PDF, a plugin has been developed for the Adobe Acrobat application. An Acrobat plugin is a helper application which runs within the Acrobat environment and extends its functionality. Access to the currently loaded documents is provided through an API, giving the plugin the ability to view and manipulate the contents of a PDF.

The 'XML Comparison' plugin has a number of functions. Its first function is to compare a Tagged PDF with an XML document that purportedly represents the same structure and content. By comparing these documents, we can see whether the structure has been passed through the document process correctly.

4.1. Document Comparison

Document comparison is effected by comparing the structure tree of the provided XML file to the structure tree inside the Tagged PDF. When the specific plugin function is selected, the plugin checks the currently active PDF document for a Structure Tree and a flag that indicates whether the PDF is 'Tagged'. If this is the case, it loads the XML file into a Document Object Model (DOM) parser. The DOM is a model for representing serialised XML trees as true trees.

A similar process then ensues to create a DOM representation of the Tagged PDF. This eases the process of comparing the two documents, because the plugin now has both documents in a canonical form.

The first comparison function offered by the plugin is an aid to allow the user to visually compare the two structure trees. The plugin creates a dialogue containing two tree-views. The left tree-view contains a representation of the PDF's internal structure tree, taken from the DOM and the right tree-view shows the structure and content of the XML starting document.

Selecting a specific node in either tree gives access to the corresponding content in the opposite document. For example, if a node is selected in the generated XML tree, it causes the corresponding content in the Tagged PDF to be internally selected and then highlighted. An example of the plugin being used in this manner can be seen in Figure 3. Implementation details are discussed in Section 5.

Should the selected XML node not exist within the PDF, a warning is given to indicate that there has been a mismatch between the two comparison documents. Should a node that is selected from the PDF Tree-view not exist within the XML, the corresponding text within the PDF is highlighted in red.

A node selected from the PDF tree that does have a corresponding node in the XML document will also be referenced, but not visually represented.

The second function of the plugin is to perform a full document comparison. This has similar functionality to the manual checker, but automates the process of comparing the documents. By comparing the DOM representations, the plugin can determine which nodes do not belong to both documents. Any nodes not appearing in both DOMs are highlighted within the tree-views and the content belonging to those nodes in the PDF is highlighted in red.

Selecting these nodes is the first step, but after this, the content of each of the matched nodes must be compared. This enables the plugin to check for differences in the content as well as allowing it to look for 'appearance artifacts'. These artifacts are text strings on the page that don't exist within the content of the abstract XML starting point. An example of such an artifact would be the (automatically generated) numbers appearing as part of main- or sub-headings. These numbers gives the reader some information about the document, but it is unlikely that they have been hard coded in the XML; rather they are likely to have been added by the document typesetting process. Text in the document that the plugin decides is of this type is highlighted in black.

4.2. XML Extraction

The plugin's second function is to extract the contents of a Tagged PDF to an XML representation. The majority of the code involved in doing this was already implemented for the comparison functionality of the plugin. An XML DOM tree was constructed for the comparison component of the plugin. Since the DOM representation is merely an alternative way of representing an XML document, the standard DOM-compliant parsers give automated methods for saving modified DOM trees back to a serialised XML representation.

Although the comparison part of the plugin already creates an internal XML representation of the Tagged PDF, it is held in a simplified format that gives just enough detail for the comparison algorithm to work on. By providing the plugin with an XML export functionality we extend its comparison and tree repair capability by enhancing the information placed in the DOM tree.

If a document has no XML starting point, retrieving an XML version of the PDF is particularly useful. For example, a number of planned extensions to this work are described in Section 6 and most of these require that an XML representation of a PDF document be available to them. The ability to create an XML representation of a PDF document for which there is no source available would also be extremely useful should anyone wish to repurpose the document for use on the Web.

Even if a document does have an XML starting point, it can still be very important to get a second XML representation from the PDF. There are a number of reasons why the generation of a second XML can have added value over the original. One reason is that extra metadata is often added by the documentation workflow that created the PDF. Authors, contributors, etc. will be recorded and stored within the document and it is essential that all this information, which can include Resource Description Framework (RDF) and Extensible Metadata Platform (XMP) data, be recorded and stored if some of the further work is to be realised. A second use for this document is to exhibit how the original XML file (which may have had a rather abstract form of markup) has been transformed into the 'layout based XML' that corresponds to the PDF file. One could then infer the nature of the typographic processes it has to go through to become a PDF document. The Tagged PDF representation has information concerning page layout, 'appearance artifacts', etc. that cannot appear in the abstract XML document. By adding this information into the new XML DOM using a custom XML Namespace [6], the extracted XML can have 'added value', while remaining separate from the original XML's structure and content.

The metadata described above is available to the plugin through the Acrobat API. The plugin uses this to enhance the DOM tree produced from the comparison stage of the plugin. Using XML namespaces as described above, the different forms of metadata are differentiated from the content and from each other.

To mark out the 'appearance artifacts', the plugin can do two types of checking. The first needs information from an already existing XML reference document to determine which parts of the PDF content exist in the XML and which have been added by the document workflow. The same algorithms as used by the document comparison process can be used to determine these 'appearance artifacts'. The other process looks for certain types of content at the beginning of blocks of marked content e.g. a number at the start of a heading is likely to be an 'appearance artifact' and not original content. Again, having an original XML starting point means that the discovery of 'appearance artifacts' becomes more deterministic.

Once the plugin has the newly enhanced DOM tree, it can either display this in a tree view for the user to check, or it can serialise the tree to an XML file.

4.3. Structuring Legacy PDF

The third and final function the plugin currently has is the ability to add structure to legacy (un-structured) PDF documents. The starting point for this is once again an XML reference document and a PDF (not Tagged).

In a structured or Tagged PDF, the structure tree references the content by using Marked Content Identifiers (MCIDs). A block of logical content is wrapped with commands to begin marking content and to end marking content. Each of these blocks has an MCID number associated with it. Each node containing content in the structure tree references the page and the MCID number to indicate which block of marked content belongs to it. In some cases, multiple MCIDs are associated with one structure element node in the structure tree.

The first job the plugin must perform is to match the freeform content in the PDF to blocks of marked content. To do this, the plugin needs to compare the content of the PDF to the content in the XML reference document. This is done by extracting the contents of the PDF to a textual representation. This representation contains links back into the PDF, so that the content can be re-associated with the PDF once it has been grouped and structured. There are a number of methods in the Acrobat API that aid the plugin in doing this and help to give a meaningful reading order to the extracted content (see Section 5 for details).

With this information the plugin can reference the text that appears within the XML nodes, to the text extracted from the PDF. A new DOM tree is built using this information, effectively generating a new XML document. This XML DOM contains the structure from the original XML document, the text from the PDF and the extra information that references the text back into the PDF.

Before the plugin attempts to add the structure to the PDF, it is very likely that in the above process, some text will still not have been assigned a position in the structure tree. An example of such text would be the 'appearance artifacts' discussed earlier in this section. By doing some position checking, the plugin determines which text is likely to be an artifact and then decides where it is likely to belong. This information is then added into the DOM tree, but is marked to indicate that it is not part of the original XML document's content.

The plugin now builds a skeleton structure tree inside the PDF document. This has no content as yet, but contains all the necessary structure nodes for the placement of the content. Once this has been completed, the plugin places MCIDs around the appropriate blocks of content, as indicated by the new XML DOM tree. These MCIDs are then associated with the appropriate nodes in the structure tree.

Once the plugin has completed this, the user is presented with the option of specifying the role-map for the newly inserted tagset. Should the user decide to do this, the structured PDF will become a Tagged PDF.

5. IMPLEMENTATION DETAILS

In Section 4 the functions performed by the XML Comparison plugin were described. These functions split into three distinct categories: Document Comparison, XML Extraction, and Structure Addition. This section looks at the implementation details behind each of these categories.

5.1. Sample Tagset

Before the implementation details can be described, the creation of test documentation for this project will be discussed.

For test purposes, a custom tagset has been defined. This tagset was specifically designed for documentation and maps closely to the default Tagged PDF tagset. Figure 3 contains a sample of the tagset used by the test documentation and Table 2 contains the mappings between the test tagset and the default tagset.

Table 2. Sample Tagset Mappings

Custom Tagset	**Default Tagset**
article	Art
title	H
section	Sect
heading	H
para	P
image	Figure
Table, TH, TR, TD	Table, TH, TR, TD

As is shown by Table 2, the mapping between the custom tagset and the default tagset is very close. This is to simplify the testing procedure.

Using the custom tagset described above, a series of XML documents have been created. A simple conversion script was then used to convert the XML documents to *troff*—this particular choice of typesetting middleware being prompted by the fact that a command in *troff* allows arbitrary PostScript to be directly inserted into the *troff* output stream. This mechanism had already been extensively used in the CAJUN project [7] for inserting unstructured hyperlinks into a PDF and thus it was a simple matter to adapt these techniques to pass on the pdfmark calls corresponding to PDF Structure Tree nodes into the output PostScript. When distilled to a PDF document, these pdfmarks are processed by Adobe Distiller and result in the addition of a structure tree to the PDF. This process follows the 'top down' approach to the document workflow as described in Section 2 and Figure 3.

5.2. DOM Tree Construction

The creation of a DOM tree from the Tagged PDF is used by both the Document Comparison and XML Extraction functions of the plugin. The ability to generate the XML DOM form of the PDF's structure and content is therefore central to the plugin's functionality.

The Microsoft XML parser (MS-XML) is used by the plugin to create and manipulate the DOM tree. It works on top of the MFC framework under the Microsoft Windows platform and is fully compliant with the W3C DOM Standard [4].

To build the DOM tree, as used within the Document Comparison function, the plugin navigates the PDF Structure Tree using the Acrobat API. The plugin uses an API call to acquire the currently active document. The Acrobat API is split into a number of layers. A few of these give access to different parts of the document and others give access to the Acrobat application. The *PDSEdit* layer gives access to the Structure tree within a PDF and methods and objects of this type are prefixed by PDS. Using the *AV* (Acrobat Viewer) layer, the plugin can gain access to the PDSTreeRoot of the currently active document. The PDSTreeRoot is the topmost node in the structure tree and using it, the plugin can navigate the entire structure tree of the document.

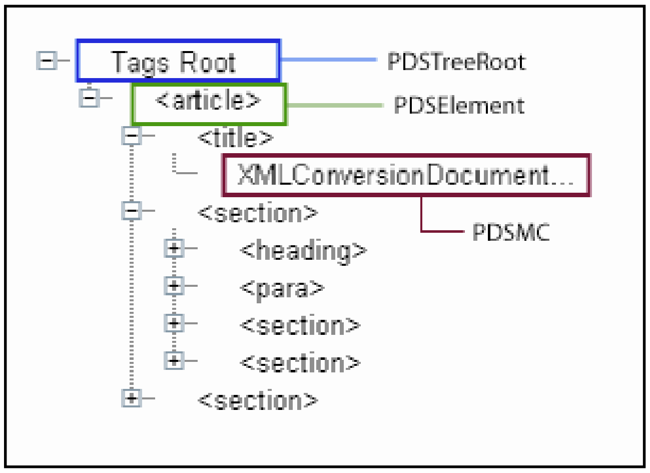

Figure 3. Structure Tree from a sample PDF.

The PDSTreeRoot contains a set of nodes. The child nodes of a PDSTreeRoot must be of the type PDSElement. A PDSElement is equivalent to a tag in an XML document. It also contains a set of nodes. The main types of node the plugin is interested in are either PDSElement nodes or PDSMC nodes. A PDSMC is effectively a node of document content. Figure 3 shows a sample PDF structure tree and the mappings between it and the types described above.

The plugin effectively performs a depth-first iteration over the structure tree, selecting a node at a time, and mapping this into the newly constructed DOM tree. Since the structure tree is navigable in a similar way to that of the DOM, the mapping is almost one to one. As each node is discovered, its type is checked. If it is of type PDSElement, a new *Element Node* is added to the DOM tree. The checking is more complex for a PDSMC, as the nodes can then contain many types of content. The API classifies each type of data and this information can be retrieved. The content is then placed into the tree in the appropriate form (e.g. text, image, etc.).

5.3. Document Comparison & Highlighting

The Document Comparison functionality of the plugin uses the newly created DOM tree, described above, as the framework for its comparisons. As described in Section 4, a DOM tree is also constructed from the XML reference document. It is fortunate that XML's architecture requires all of the internal nodes of any well-formed XML-based tree to be clearly demarcated. This, in turn, means that relatively straightforward tree matching algorithms and string comparison methods (see for example [8]) are adequate for relating the nodes and content extracted from the PDF to the original XML starting point.

Content appearing in both documents is easily discovered, for our test set, because both the source XML and the typeset PDF have their content in reading order. Any blocks of text not matched are then marked to show that they were not part of the original XML content.

5.4. XML Extraction

Section 5.2 describes the process of building the DOM tree from the Tagged PDF and, as mentioned in Section 4.2, the information stored in this for use by the Comparison function, is not sufficient for the purpose of full extraction.

For Document Comparison, one of the requirements is that both a Tagged PDF and an XML document are available. This is not the case for the XML Extraction function of the plugin.

If there is an available XML document, the plugin makes use of it in the same way as for Document Comparison, using it to indicate sections of content that do not appear in the original source document. However, if this is not available, it is still possible to mark up a number of types of 'appearance artifact'. The most obvious example is when a block of marked content has been found that is stored within a heading or a list. In the case of the heading, any preceding numbers will very likely be section numbers, so these can be appropriately marked as such. In the case of a list, it is quite possible to have number or bulleted lists and so the plugin checks the start of every line for any consistent starting character(s).

Tagged PDF uses a packet format to allow for encapsulation of any XML data associated with the file. This can be any form of markup, though the common types are RDF and Adobe's own XMP. The Acrobat API gives access to all of this metadata and enables us to add it to our DOM tree within its own namespace.

6. PROGRESS SO FAR

As we have already outlined, our plugin exhibits the correspondence between a structured PDF and an external XML representation of the text; if these two representations differ it highlights where the differences occur. If no XML version of the text exists the plugin is also capable of extracting a fully tagged PDF into an equivalent external form, with an XML version of the PDF tags and with the text extracted from the PDF in reading order.

Our experience so far has convinced us of the many advantages of a abstracting a Tagged PDF document into an external XML form for the purposes of maintenance and repair. The Acrobat interface to the software has to be done via a plugin because only here are the methods exposed through the Acrobat API for manipulating, extracting and inserting both text blocks and tree structure. However, the XML window can be administered by using standard tools such as MS-XML and COM.

Looking to the future we can now see three further applications of our work, all of them addressing real-life problems in maintaining an increasing number of Tagged PDF documents.

6.1. Adding structure to legacy PDF files

If an externalised XML representation of a legacy (unstructured) PDF has been obtained, by any of the methods outlined in the previous section, one of its most obvious uses is to act as a template for rewriting the PDF file with a correct structure tree inside it. This work is now actively under way. The new features are that the PDF Structure Tree is now built up by the plugin itself using API methods. The runs of text indicated by the external XML original are cross-correlated with those found at the leaves of the PDF Pages tree and the cross-links to this content, from the PDF structure tree, are set up as part of the tree-building routines. At present the routine works correctly on test material that is in reading order, and where there is a very close textual match between the XML and PDF versions, but further work is needed for it to cope robustly with building the linked lists of text fragments that are needed in situations where the PDF text runs are *not* in reading order.

6.2. Incremental Updating of a Tagged PDF

There are many circumstances where a PDF document has to be produced using existing typesetting systems with no capability for embedding a PDF Structure tree. This PDF document may well be widely circulated and of considerable importance (e.g. a user manual, a set of safety instructions or a tax form) with a corresponding need for a structure tree to aid in accessibility and intelligent reflow. As we have seen, software such as Make Accessible can infer and embed enough structure to enable a PDF document to be reflowed, or to be read out to a visually impaired user, but in many cases there will be a need to embed a company-defined tagset within the PDF (possibly role-mapped to the Adobe standard set) to aid in repurposing, structured searching etc. Using our plugin it is possible to extract any existing structure tree from a document together with its associated text. This standardised external representation could then be transformed (perhaps via an XSLT script) into a form which correctly uses the company's standard tagset. If a new version of the PDF document is now created (again, via a document flow which produces an unstructured PDF) then, armed with a standard XML starting point for the previous version of the document, a combination of the tree insertion capability described above, coupled with a tree repair function, would allow the document to be incrementally updated. Clearly the tree repair function will be needed to allow for the fact that some text may have altered in the new document and, possibly, some new material will have been added. Such a repair capability is a natural extension of the tree comparison routines already used in our structural integrity check but successful tree repair depends on there being reasonably modest alterations to the original document; it will not be feasible if there are gross mismatches between the two versions.

6.3. Standoff Markup

A final and very attractive prospect opened up by our techniques is to provide the facilities of Tagged PDF for a PDF file that is not only unstructured but is also ‘read only’. Such PDF documents are becoming increasingly common whenever organisations want to protect their intellectual property interests by saving it, using the Acrobat security routines, in a form where any alterations are prohibited. In such cases there is no prospect of rewriting a new version of the PDF with an embedded structure tree. However it would be feasible to produce a new version of our plugin where we firstly create a structured and external version of the PDF document but then we enhance the plugin's DOM tree representation of the document with a list of pointers to the text and image objects within the Pages tree of the PDF. The plugin could then display structured tags, bookmarks etc. for the PDF document exactly as if it were a tagged PDF but using only the external tree for navigational guidance.

Such a technique of ‘externalised’ or ‘standoff’ markup also has the great virtue that it enables different PDF Structure trees, perhaps with different markups in different tagsets, to be applied to a single ‘pure’ copy of the underlying document. This separation of structure from content is reminiscent of the program/data separation in virtual memory computers and many of the advantages of program shareability carry over into the document domain [3].

7. CONCLUSIONS

PDF is firmly established as a *de facto* document standard for the accurate page-based display of rich and complex material. Therefore it will be important, for some time to come, that existing PDF files should be capable of being enhanced, with as much structure as possible to assist in archive maintenance and repurposing of existing material.

The advent of Tagged PDF and Acrobat's routines for converting PDF very approximately to and from HTML (Save As HTML and Acrobat Web Capture respectively) mark a distinct step forward in providing a framework for representing document structure within PDF. However, professional publishing requires more than this: it requires that a PDF should be relatable not just to HTML but to an arbitrary structured starting point, almost certainly involving the author’s or a publisher's own XML tagset. Furthermore, this tagset should be capable of being embedded in the final PDF file as a Structure tree that is capable of maintenance, repair and upgrade, even if the intervening typesetting software either damages the originally intended structure, or fails to pass it on at all. For all these reasons we believe that our plugin, and developments from it, will play a significant part in the maintenance of Tagged PDF documents.

ACKNOWLEDGMENTS

We thank Adobe Systems Inc. for sponsorship of Matthew Hardy's Ph.D. studies and for providing a Summer Internship for him in 2001. In particular we thank Loretta Guarino, Richard Potter, Dan Rabin and other members of the Tagged PDF effort at Adobe, for answering technical questions and for suggesting applications for our plugin.

REFERENCES

[1] Adobe Systems Incorporated, *PDF Reference (Second Edition) version 1.3,* ISBN 0-201-61588-6, Addison-Wesley, July 2000.

[2] Adobe Systems Incorporated, *PDF Reference (Third Edition) version 1.4,* ISBN 0-201-75839-3, Addison-Wesley, December 2001.

[3] David F. Brailsford, ‘‘Separable hyperstructure and delayed link binding’’ *ACM Computing Surveys*, vol. 31, no. 4es, December 1999. `http://doi.acm.org/10.1145/345966.346029`

[4] *The Document Object Model (DOM).* `http://www.w3c.org/TR/2000/REC-DOMLevel-2-Core-20001113/`

[5] W.S. Lovegrove and D. F. Brailsford, ‘‘Document Analysis of PDF Files: Methods, Results and Implications,’’ *Electronic Publishing—Origination, Dissemination and Design*, vol. 8, no. 2 & 3, pp. 207–220, June & September 1995. `http://cajun.cs.nott.ac.uk/compsci/epo/papers/epodd/epoddtoc.html`

[6] *Namespaces in XML.* `http://www.w3c.org/TR/1999/REC-xml-names-19990114/`

[7] Philip N. Smith, David F. Brailsford, David R. Evans, Leon Harrison, Steve G. Probets, and Peter E. Sutton, ‘‘Journal Publishing with Acrobat: the CAJUN project,’’ *Electronic Publishing—Origination, Dissemination and Design*, vol. 6, no. 4, pp. 481–493, December 1993. `http://cajun.cs.nott.ac.uk/compsci/epo/papers/epodd/epoddtoc.html`

[8] *The* treediff *project.* `http://www.alphaworks.ibm.com/tech/xmltreediff`