Configurable Editing of XML-based Variable-Data Documents

John Lumley, Roger Gimson, Owen Rees
HP Laboratories
HPL-2008-53

Keyword(s):
XSLT, SVG, document construction, functional programming, document editing

Abstract:
Variable data documents can be considered as functions of their bindings to values, and this function could be arbitrarily complex to build strongly-customised but high-value documents. We outline an approach for editing such documents from example instances, which is highly configurable in terms of controlling exactly what is editable and how, capable of being used with a wide variety of XML-based document formats and processing pipelines, if certain reasonable properties are supported and can generate appropriate editors automatically, including web-service deployment.

External Posting Date: October 6, 2008 [Fulltext] Approved for External Publication

Internal Posting Date: October 6, 2008 [Fulltext]

Published and presented at DocEng'08, September 16-19, 2008, São Paulo, Brazil

Configurable Editing of XML-based Variable-Data Documents

John Lumley, Roger Gimson, Owen Rees
Hewlett-Packard Laboratories
Filton Road, Stoke Gifford
BRISTOL BS34 8QZ, U.K.
{john.lumley,roger.gimson,owen.rees}@hp.com

ABSTRACT

Variable data documents can be considered as functions of their bindings to values, and this function could be arbitrarily complex to build strongly-customised but high-value documents. We outline an approach for editing such documents from example instances, which is highly configurable in terms of controlling exactly what is editable and how, capable of being used with a wide variety of XML-based document formats and processing pipelines, if certain reasonable properties are supported and can generate appropriate editors automatically, including web-service deployment.

Categories and Subject Descriptors

I.7.2**[Computing Methodologies]**: Document Preparation — *desktop publishing, format and notation, languages and systems, markup languages, scripting languages*

General Terms: Languages

Keywords: XSLT, SVG, Document construction, Functional programming, Document editing

1. INTRODUCTION & MOTIVATION

In recent years there has been much research on formats, semantics and implementation techniques for variable-data documents, driven in part by new business opportunities in customised publishing. Our own exploration has focussed on the *Document Description Framework* [1] (DDF), an architecture which treats a variable content document as an extensible function with distinct separation of data, logical structure and presentation. It is heavily dependent upon XML representations and technologies, using an XML tree as the main syntactic representation, with constructional semantics supported by sections of XSLT[2] and an SVG-based geometric presentation by a hierarchical tree of layout instructions[3].

During early research such documents (which can of course be highly programmatic) have been constructed 'by hand', but developing innovative methods of authoring and editing has always been one of the goals. Users have become very used to editing *on a visual form of the final document* (WYSIWYG rather than declaring intent such as using LaTeX), but when the document is highly variable and there are very many different possible instances, how to do this is not immediately obvious.

We were also keen to consider that, especially in complex commerical document workflows, there may be many distinctly different roles of 'editor' and 'author' for such documents. Original designers may generate branding material and document exemplars that can result in various elements and styles of document templates being fixed for other downstream editors. Other individuals may have edit capability which is restricted ('you can choose the style but not edit the style'; 'you can edit the mapping to data sources, but not style nor fixed content') .

Editing used to be the province of dedicated standalone editing tools, of increasing complexity and cost. But in the system and service deployment scenarios we envisage, many of these editing actions must also be supported through web-deployed facilities.

We sought an extensible architecture that could support the possible editing and authoring of sets of variable-data documents that:

- Was highly configurable in terms of tuning what editing can be performed on documents and by whom
- Can edit variable-data documents of high complexity, from understandable visual presentations of *sample document instances*
- Can support a wide variety of XML-based document types
- Can operate within workflows where multiple documents are merged and correctly edit appropriate components.
- Does not depend upon any *intimate* knowledge of the processing pipeline that creates instances of variable documents, nor even the detailed semantics of the documents themselves
- Requires minimal or zero disturbance of the document processing pipeline tools when supporting editing.
- Is deployable both in standalone and web-service situations, with very little change in configuration.

In this paper we will usually discuss examples using DDF, but the techniques are applicable to a wide range of potential XML-based variable document technologies, provided a few key requirements are satisfied. We'll show that the basic technique is very effective and highly flexible when the causality between source and result is 'local'. Adding more knowledge of the semantics of the source language supports modifying documents in less local scopes.

We'll start by illustrating editing on an instance of a variable document then outline the basic implementation, including how 'edit-

DocEng'08, September 16-19, 2008, São Paulo, Brazil.

ability' may be described and how modifications are actually performed on source (template) documents. We'll then discuss using a 'view' of a document, remote editing using the technique and how we can edit the extent of modification other users can perform. Limitations and specialist treaments are then outlined, followed by a survey of prior art and future directions.

2. EDITING: ON A DOCUMENT INSTANCE

A variable content document is just that - *variable*; its appearance usually depends upon the data that is bound to it. The human consumer reads the intended message in a visualisation of a document *instance*. Consequently editing within such a visualisation is preferable, provided we don't have to compromise the 'smartness' of the document substantially. We'll outline the issues and approaches on an example 2-page tourist flyer that is a true variable document:

Figure 1. An example 2-page variable content brochure

This example is moderately complex as a variable data document. Each holiday is constructed as a separate sub-component, which varies its appearance dependent on whether it is marked as being a special offer and the number of photographs available for the tour. The tours are grouped in pairs by page and arranged as a horizontal flow. Other elements, such as the selected customer are very simple interpolations. The background is constant for each page. Figure 2 shows some alternative results for other customers:

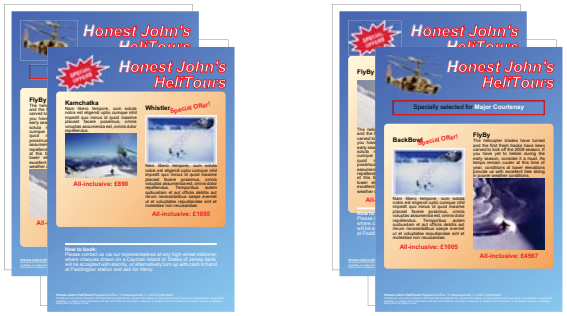

Figure 2. Two differing instances of the brochure

Now if we want to alter the template, can we do so by simple selection and dialogue actions on this visual instance? Can we point to a text block that is a 'description' of one of the tours and edit its style for *all* such descriptions in that instance document and all other instances that will be generated subsequently from the template? Figure 3 shows such a case, where we've selected one of the text blocks in the result document and have an editing dialogue that allows us to alter various properties, such as borders and margins:

Figure 3. Editing an element of the template

In this example we have selected a text block for editing[1] and this has caused the 'text block' editing dialogue to be displayed and populated with the correct properties for the text block in question (and indeed for all text blocks, such as that under the right hand picture, that were generated from the same element in the template source.) These properties can be altered and eventually an 'Apply' action performed. This new set of properties is then written onto the correct source element and the instance document re-built. Figure 4 shows details of the 'before' and 'after' states of the selected text blocks where we added a border.

Figure 4. Changes to text blocks generated from the same source element - before & after

Note that whilst we selected the text in the 'FlyBy' tour, another text block in the 'BackBowl' tour was also edited - they were generated from the same point in the source template (the code fragment that processes 'description' data sections) and hence *both* these have altered. This is desirable for promoting coherence of styling and branding.

It will also be apparent that the picture on the left-hand tour has moved down after the change to the text styling - this is because

[1] What exact form of mouse and/or keyboard action triggers the event isn't important - many schemes could be used with the same underlying architecture.

the tour has been laid out as a group, flowing the title, description and picture below each other (the 'BackBowl' tour flows in a different order as it is a special offer). Can we edit the properties of this flow? If as in this case, we can select the group, then we can:

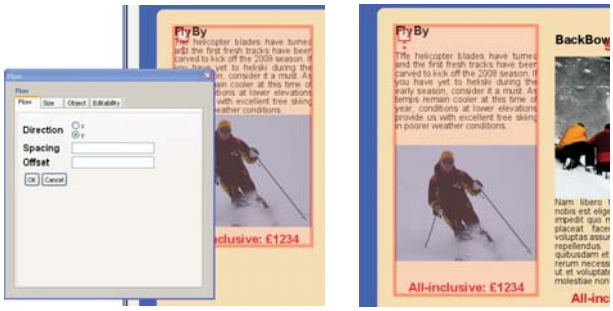

Figure 5. Altering a compound layout - changing flow spacing

We can select the group because an element in the final result corresponds to the 'flow' instruction in the original template and a suitable editor is then displayed. Using this we can change the spacing property on that instruction and in the subsequent re-build we get the new result. Note that the other tour ('BackBowl') has *not* changed its spacing - it is defined in this template by a different part of the code. We can add a new element if we select the flow and then add a new child of the corresponding source element:

Figure 6. Adding new elements to a compound layout

Here we have added a new (horizontal) flow to our product group, moved it up in the order to place it between the description and picture and added two ellipses to it.

A single instance of a variable document isn't of much interest or value - it's the variability that matters. So we should consider showing multiple instances. We can do this in several ways: the easiest of course is to show a vector of results corresponding to a vector of instances of the variable data. We would expect to be able to point to a text block that fulfils the description role in *any* of the instances, make an alteration and see the results in *all* of them. With the implementation we'll describe next, this is exactly what would happen — each of the instances would be edit-sensitive in the same way and changes would happen on the common template. Thus if document dependency is being tracked all the instances displayed would now be 'out-of-date', since the template upon which they depend is now 'younger' than the instances and a synchronisation rebuild would propagate the change to all instances.

3. EDITING: BASIC IMPLEMENTATION

In our editing scenario we are presented with an instance of a variable document and we select a displayed block of text. If we then want to change the font size, 'where is that defined?' and 'what can I change it to?' In normal document editing this is usually straightforward - the editor has intimate knowledge of the mapping between source element and display element - but in variable documents that need not be the case. For example we might want to change the text, *but the text involved is not in the source template, but is variable data.* In this case we may want to change either the variable element from which this text is generated (e.g. from *last_name* to *first_name*) or even the actual variable data itself (e.g. change *Lumley* to *Gimson* in the customer record) - we'll discuss the latter in section 8.2 .

Equally well how might we control that only certain pieces of text may be edited by this particular (human) editor, and only in prescribed ways? For example we may only permit the font size to be altered to one of a finite set of possibilities, or choose from a particular font-family/size/style combination...

3.1. Linking Result To Cause

The key technique of this approach is to ensure that in the instance document being viewed there is just sufficient information to:

i) identify what editing might be performed on (or through) this displayed element

ii) where in the source document(s) should any change take place

Figure 7 shows a schematic example production process for a very simplified variable document (whose only variation is in the number of white 'rating-point' crosses it presents):

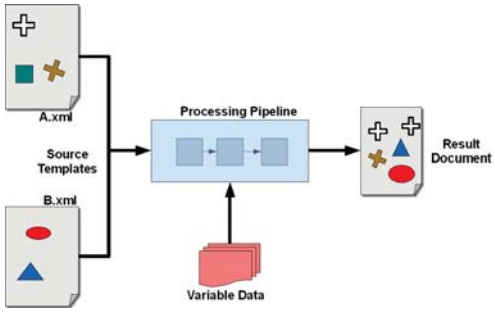

Figure 7. Merged variable documents and result

The variable data document is constructed from some merge of two source templates, containing both program and symbolic output elements. This merge is convolved with an instance of the variable data, by a black-box processing pipeline creating a result document. Some, but not all, of the elements in the source templates appear in the result, because of conditionality over the variable data. (The green square didn't appear, the white cross appears twice, presumably because of some data-dependent iterator.)

Let us assume that we wish to edit these source documents, but only in areas where we permit, and from a view of the result document (**What You See Is One of the Ones You'll Get**). Firstly we can somehow define *which* elements in the source are variable, and how. Then we need to arrange that the user can select a piece in the res-

ult document corresponding to an editable element in the source, open up a suitable and correctly populated editing dialogue and then perform the requested editing action *on the correct element in the correct source template.*

As we will show, this can be acheived if two pieces of information can flow from the source to the result, without otherwise disturbing the result, and can then be identified in an interactive editing rendition of the result document. These are:

i) the name of the specific editor or editing component to take control of the editing action

ii) an address for the specific source element that caused this result piece to appear - the exact source template responsible and a unique position within that document.

This means we need to add the sorts of links shown in Figure 8:

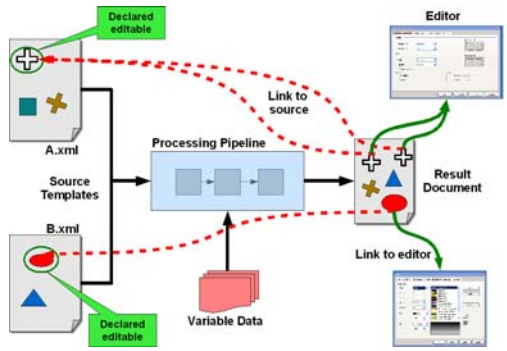

Figure 8. Links from result to editors and source

If we know **what** we want editable in the source then we could arrange to modify all appropriate source documents with such information. For XML-based formats we can do this simply through programs (XSLT transforms are certainly the simplest) that match such situations and attach one or more additional attributes to the elements under consideration. The source fragments (in pseudo-SVG for brevity) that made up the schematic example might be:

```
<doc href="A.xml">
  <layout function="X">
    <cross rotate="30" fill="brown"/>
    <layout type="flow">
      <xsl:for-each select="rating-point">
        <cross fill="white" .../>
      </xsl:for-each>
    </layout>
    <square fill="green"/>
    <xsl:call-template name="bar"/>
  </layout>
</doc>
<doc href="B.xml">
  <xsl:template name="bar">
    <ellipse fill="red"/>
    <triangle fill="blue"/>
  </xsl:template>
</doc>
```

Figure 9. A sample source spread across two documents

with just the white cross and the red ellipse editable, might be transformed to:

```
<doc href="A.xml">
  <layout function="X">
    <cross rotate="30" fill="brown"/>
    <layout type="flow">
      <xsl:for-each select="rating-point">
        <cross fill="white" e:source="A.xml#1/2/1/1"
        e:edit="cross"/>
      </xsl:for-each>
    </layout>
    <square fill="green"/>
    <xsl:call-template name="bar"/>
  </layout>
</doc>
<doc href="B.xml">
  <xsl:template name="bar">
    <ellipse fill="red" e:source="B.xml#1/1"
    e:edit="ellipse"/>
    <triangle fill="blue"/>
  </xsl:template>
</doc>
```

Figure 10. The annotated versions of Figure 9

where we now use an encoding of the source file URI and a relative tree position within that file to link back to the source[2], and some appropriate name for the edit component to take responsiblity for action. If this document is processed through a pipeline such that these two additional attributes (shown here in namespace mapped to a prefix `e:`) appear attached to every instance of the elements appearing as a causal consequence of the source, then we can implement a generic editor that is independent of the document type.

Note that in our example *two* white crosses appear (presumably because there were two rating-points). Any one of these can be used to edit its properties, or indeed the iterator context surrounding it.

This critical requirement (transmission of source pointer and required edit component through the processing pipeline) may at first glance appear expensive, especially if almost the whole of the document is editable, but the situation isn't as bad as you might think:

- If the pipeline is principally XML-based and doesn't serialise and re-parse XML trees in intermediate stages (i.e. the trees stay in memory), the additional burden of the tracing attributes is relatively slight - they only need simple copying.
- This additional overhead is *only* required for editing - production runs know nothing about these additional markings, that have not been written onto the real source templates.

What we do however require is that the processing pipeline acts as a 'good XML citizen'. That means:

i) It is **not** an error for (unknown) foreign namespace attributes to appear on XML elements being processed by stages in the pipeline. For example a `e:source="`*`path`*`"` attribute on an `fo:block` element within an XSL-FO document

ii) Where an XML element appears in the output corresponding to a given input element, then 'unknown' attributes should be copied across, and

iii) If appropriate the output result tree topology should locally follow that of the input.

[2] Several schemes could be used for this linking, XPath[4] being a general form. That shown here is textually 'tight' when serialised but easy to follow down the tree to identify target elements.

By supporting these principles, in-band communications channels can operate through the pipeline, letting these systems work. In our example the final single result document (in pseudo-SVG with position attributes omitted) might be:

```
<svg:svg href="result.svg">
  <cross rotate="30" fill="brown"/>
  <svg:svg>
    <cross fill="white" e:source="A.xml#1/2/1/1"
    e:edit="cross"/>
    <cross fill="white" e:source="A.xml#1/2/1/1"
    e:edit="cross"/>
  </svg:svg>
  <square fill="green"/>
  <ellipse fill="red" e:source="B.xml#1/1"
  e:edit="ellipse"/>
  <triangle fill="blue"/>
</svg:svg>
```

Figure 11. The final result version of Figure 9

where the elements have been positioned by the layout functions in `A.xml`. Now we have information on the definition of the visual result that will allow i) editable pieces to be identified, ii) what class of editing might be performed and iii) where in the source files the exact declaration of the element is.

Editing is not confined just to source 'leaf' primitives. The same technique can be used to edit combinators, such as layout functions, which are *not* tree leaf nodes. Suppose we wish to permit editing the `flow` layout function (e.g. allowing a spacing to be altered). If we now arrange that `A.xml` is annotated to become:

```
<doc href="A.xml">
  <layout function="X">
    <cross rotate="30" fill="brown"/>
    <layout type="flow" e:source="A.xml#1/2"
    e:edit="flow">
      <xsl:for-each select="rating-point">
        <cross fill="white" e:source="A.xml#1/2/1/1"
        e:edit="cross"/>
      </xsl:for-each>
    </layout>
    <square fill="green"/>
    <xsl:call-template name="bar"/>
  </layout>
</doc>
```

Figure 12. Annotating a layout function

Then if the 'agent' who processes the flow and generates the result is a good XML citizen the result document will now look like:

```
<svg:svg href="result.svg">
  <cross rotate="30" fill="brown"/>
  <svg:svg e:source="A.xml#1/2"
  e:edit="flow">
    <cross fill="white" e:source="A.xml#1/2/1/1"
    e:edit="cross"/>
    <cross fill="white" e:source="A.xml#1/2/1/1"
    e:edit="cross"/>
  </svg:svg>
  <square fill="green"/>
  <ellipse fill="red" e:source="B.xml#1/1"
  e:edit="ellipse"/>
  <triangle fill="blue"/>
</svg:svg>
```

Figure 13. Result document corresponding to Figure 12

The `svg:svg` element in the output corresponds to the layout function in the template. If this element can be selected in the resulting visualisation (as distinct to both its children who are also editable), then we can edit the flow itself. This is one area where we make fairly stringent demands on the document processing pipeline: we need to preserve structure in the result presentation that is implied in the source. In this case the group of white crosses remains a structural group in the result (as an `svg:svg` sub-tree with its own local co-ordinate space), rather than being flattened and each cross being positioned absolutely. In our layout model the benefits of such preservation (especially in being able to build higher-order layouts) makes this feature almost universal.

We can develop this a little further. Suppose we want to edit what the ranking stars measure, i.e. the attachment to the `rating-point` in the `xsl:for-each`. If we can arrange that `cross` elements as direct children of `xsl:for-each` are edited differently (see later for how we do this), then we could substitute another edit operation `e:edit="for-each-cross"` on suitable elements, which will enable the value of the selector to be altered, and we can do this without altering the editing of normal white crosses.

This succeeds because it operates *locally* within the XML tree and, if the processing pipeline behaves as a good XML citizen, the editor need only match elements within *local* scope in the source tree.

3.2. Other Components

We have outlined the fundamental mechanism of signalling from editable parts of the source to the result document. To build a real editor we must then address several problems above this:

- How do we describe *what* we want or permit to edit (and how might we control such editability to suit different roles or users)?
- How do we describe *what* can be edited on those parts of documents that are editable?
- How do we inject the necessary tracking information into the source documents?
- How do we display the editability in the result?
- How do we construct or configure the necessary editing components, both in terms of their interactivity, their initial states and their modification action on the source template?
- How do we update displays with the consequences of changes?

Our approach is to use declarative descriptions of intent as much as possible and generate appropriate code from these automatically. There are two starting points: an XML description of all the *editability* supported within this editing application and an XML description of the document processing workflow within which the editing is expected to be supported. Different editing applications can be built by varying the editability; differing document workflows can be supported by altering the workflow description.

3.3. Describing the Editability

When we describe the editability we require, we assume that elements that we wish to edit in different ways *are distinguishable in the source templates*. In our simple example that means that we can distinguish between a cross which is white and any other general form of cross (which presumably is not). Our overall description

of editability is a set of descriptions, each tied to a specific type of source element we wish to allow to be modified. In our implementation, with a little simplification[3], this can look like:

```
<target match="cross[@fill='white']">
  <bindings>
    <bind name="stroke" nodeset="@stroke"/>
    <bind name="size" nodeset="@size"/>
    <bind name="rotate" nodeset="@rotated"/>
  </bindings>
  <group>
    <select1 label="Stroke" bind="stroke">
      <item>red</item>
      <item>black</item>
      <item>green</item>
    </select1>
    <input label="Size" bind="size"/>
    <checkbox label="Rotated" bind="rotate"/>
    <button label="Delete"
    action="delete-element"/>
  </group>
</target>
```

Figure 14. Declaring editability for a simple element

which declares that white-filled crosses can have their border (stroke attribute) set to either red, green or black, their size property set to a number and an optional 'rotated' boolean value. What these actually mean in terms of the final view of the cross doesn't matter - that's a matter for the document's semantics. As far as editing is concerned, we're just altering properties.[4]

The first issue is how we target this edit requirement to the correct source elements. We use a boolean pattern (the value of the `match` attribute), using XPath[4] semantics, such as `cross [@fill='white']`. This can be extended easily: editing crosses that are white or red or triangles would use a pattern `cross [@fill=('white','red')]|triangle`. If we wish to edit crosses that are not white differently we would require a mutually-exclusive pattern `cross[not(@fill='white')]`. Alternatively and more usefully, the use of a priority ordered set of patterns `cross[@fill='white']` (priority 1), `cross` (priority 0) can give more natural declarations.

Only elements which match these patterns are editable - all others will not get 'annotated' and thus are immutable by construction. If we wish to configure what is editable dependent upon 'role' then we generate a different set of patterns, presumably smaller subsets as editing capability is reduced. With a little ingenuity these patterns can be used to give subtle changes. For example a text block may be declared editable through the pattern `fo:block`, but a text block that was inside an iterator ('for each address-line - text') can be matched by the more specific pattern `xsl:for-each/fo:block` and the ability to alter the context for the block (i.e. the pattern over which the iteration occurs) can be added.

Clearly this basic technique has limitations. For example in an XSLT-based template a cross might be coloured white by several means other than a direct attribute: by a computed attribute value (an attribute value template), an attribute-generating child instruction or indirect application of an *attribute set*. (We'll discuss these in section 8.)

With suitable use of such patterns we can cover some interesting application-specific cases, with no alteration to the underlying machinery. For example if we wish to use a document element that is either a text-block or an image dependent upon some property of the data, then XSLT code for it might be that shown in Figure 15:

```
<xsl:choose>
  <xsl:when test="CONDITION">
    <svg:image xlink:href="picture.png"/>
  </xsl:when>
  <xsl:otherwise>
    <fo:block>
      <xsl:value-of select="."/>
    </fo:block>
  </xsl:otherwise>
</xsl:choose>
```

Figure 15. A complex variable document element

We can recognise this as a single entity with a pattern of two parts: `xsl:choose[xsl:otherwise/fo:block]/*/svg:image` and `xsl:choose[xsl:when/svg:image]/*/fo:block`. Using this would arrange that *either* of the image or text block appearing in the result would be editable, though the editing 'effector' (the program that actually peforms the change to the source) would have to be smart enough to edit suitable properties on both elements as well as the test condition. This is an example of a case where any generators of these edit components would have to be provided with increased knowledge of the source language to accomodate non-local behaviour.

Having identified an editable element, we must consider how it can be altered. There are three main types of manipulation:

- Altering a property, such as a size or colour or some control parameter. This is the most common operation.
- Altering the element itself completely - either by deleting it, cloning it, altering its position with respect to its siblings or replacing it with another element completely. (Some of these operations need to be performed within the element's parent.)
- Adding a permitted child to it.

In our example the `bind` elements describe properties (attributes on the source XML element) that can be altered and their type. The `group` construct describes specific editing controls (with type, label and permitted values) that can be displayed within any edit dialogue, and the binding between these controls and the properties. The `button` with the 'action' describes a possible action on the element itself. Simple extensions can add other useful features, such as validity testing (e.g. the size must be positive and not 0), relevancy between the editing controls (e.g. if there isn't a stroke colour set, then a 'stroke-width' control is inoperable) and further conditionality (the size could be either set statically or linked to some data-dependent variable)

From this description we can generate a program to create a suitable editing dialogue and include the links to some data instance corresponding to the element currently being edited. For reasons that will become apparent later, we consider this editing dialogue

[3] We use some syntax and semantics from XForms[5] in this area

[4] Care is needed that these declarations do not have unfortunate side-effects. Changing the fill colour of the white cross could result in a new cross which *wasn't* white, and thus no longer editable.

to be a specialist view of the result document (the 'white-cross-editor') which happens to change its details as the selection in the main view alters. By simple extensions to these editability declarations we can for example include new archetypes of the element for adding new copies to the document, describe suitable visual indicators and selectors for them and indicate whether the elements are capable of receiving additional children.

3.4. Modifying the Source Document(s)

When new values for properties of the part have been decided by the human editor, she usually commits the changes via some 'Apply' control. The editing dialogue can then package up the new (and unaltered) values as a data structure, along with the source pointer to the exact element being edited, such as Figure 16:

```
<edit target="A.xml" position="1/2/1/1"
effector="white-cross">
  <bind name="stroke">red</bind>
  <bind name="size">13</bind>
  <bind name="rotate"/>
</edit>
```

Figure 16. Edit modification request for the white cross

This data structure is then passed to a suitable 'effector' program (identified as 'white-cross') which is responsible for altering the properties on the element at the position *1/2/1/1*. There are many ways this could be achieved, such as direct manipulation of a DOM tree, or in the case we use, by invoking an XSLT transform (actually called 'white-cross') which copies the document, modifying those parts (of which there should be one only) which is at the indicated position. These modifications could be altering a property attribute, adding a given child or cloning or deleting the element itself. Some operations may involve an additional modification of the *parent* of that element, such as altering the sibling order of the element or changing a specific property which is defined in the parent itself, such as the 'context-iterator' for a text-block within an iterator.

4. EDITING: THROUGH A VIEW

Thus far we have described how we can identify what is editable and how, and how this information can pass through intermediate processing stages to end up on the instance result document. We could then have a smart editor that understood all these annotations and responded appropriately. Such an editor would render the view into a visual form, attach suitable processing to interaction events such as mouse or keyboard clicks and arrange for selection indications and editing dialogues to be displayed and populated as needed. But such an approach is not as flexible as it could be. In this section we show how a further level of abstraction, *editing through a view* enables many different types of manipulation to be performed with a canonical final viewer.

Recall that our result document is an XML tree, with editing annotations added to certain nodes that can be manipulated - it's usually an SVG tree, but doesn't have to be, provided we can convert it into some form of suitable visualisation, preserving the essential annotations. For example we could visualise the result document as a (picture of) an XML tree:

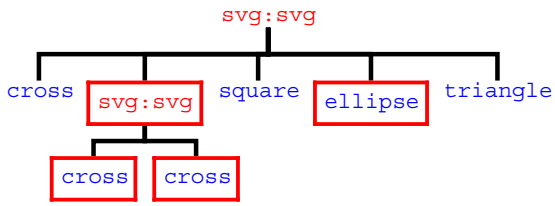

Figure 17. An XML tree view of the final result document

If the view that made the tree picture was also a good citizen, then the elements corresponding to the editable nodes (the white crosses, the ellipse and the `svg:svg` that's a flow) - in this case the text tags, will still have the editing annotations attached. (In this view we have decorated the elements which are editable with a border surround, but they are not necessary for the method.) Thus these parts *are still editable in this view*. No other alterations are required to the editing framework whatsoever. Moreover multiple views of the same result document can co-exist, each of them having potential access to the full editing of the document sources. The critical idea is that the visual elements which are tagged for editability as an `svg:image` don't have to actually *be* an image in the final view, they just have have the correct editing attributes on them to invoke the 'image' editing dialogues when selected.

The program that generates this resulting view can take many forms, but as input and output are XML trees, XSLT has considerable advantages, and as described elsewhere makes it distinctly possible to generate suitable view-generation programs automatically. In our implementation there is a message exchange mechanism whereby views can generate messaging events and respond to certain classes of these messages - this is used extensively to provide synchronisation between different views of the same result document, supporting facilities such as co-ordinated selection displays[5].

This use of a specific view to observe a result document can have very many uses. For example a multi-page document could be converted into an overlapping set of single pages with additional visual control elements controls to set the visibility on the page in focus. Similarly we could produce a view that *only* showed pieces that were editable, either as perhaps a (long) list of correctly sized visual pieces, or as a view of the document with uneditable sections greyed out. Just as importantly we can consider the editing dialogues themselves to be specialist views of the result document (possibly in a different format such as XForms) which also respond to the selection messages to set appropriate values for the properties.

5. EDITING: BUILDING THE PROGRAMS

We've described how by annotating source documents appropriately and then detecting these annotations within some final view, we can build a configurable editor. We've also described how the editability can be declared and what are appropriate editing 'effector' programs to alter the sources. There is now the issue of how these components and annotations are actually generated and injected into the workflows. To acheive this we must have a machine-readable ver-

[5] Clicking on one result element generates a selection event which triggers the display of selection for *all* result parts that share the same source element.

sion of the workflow being used. Unsurprisingly the key is to develop a compiler that operates on the editability specification and generates several program components and stitches them into modified workflows. We need to build five separate types of output:

- Programs that annotate source templates (or variable data files) with editability tracing attributes (the 'annotators')
- View generators that take the result document and produce the appropriate graphic view complete with sensitivity information or interactivity
- Edit dialogue generation programs which make the interactive controls corresponding to the required editing (e.g. text, ellipse under interator etc.) and the specific instance of that type being operated on. This should also include code for type verification, relevancy etc. These programs can be split into two separate sections for efficiency if required: a static component (the display, verification code, possible values etc.) and a dynamic portion, giving binding for a particular instance.
- Edit effecting programs that apply the requested change to the selected element in the selected source template.
- A modified workflow such that the appropriate programs (especially the annotators) are added into the process where needed.

We need the workflow of our example to be modified to:

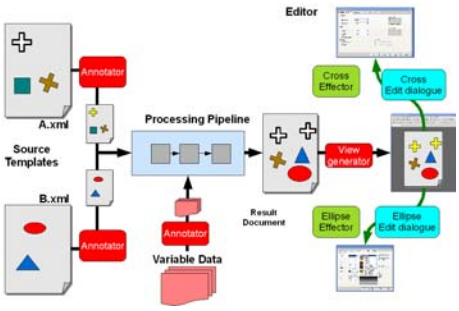

Figure 18. Modified workflow

The additional workflow components (the annotators and view generators) need to be 'stitched into' the existing workflow. If the annotation, view & edit-dialogue generation and edit effector programs operate on XML sources then XSLT is a suitable language, which has the advantage that they too can be generated as XML, by a compiler itself written in XSLT. A workflow being presented in XML also helps as well of course.

The annotator programs are generated from the editability description as XSLT transforms with an `xsl:template` instruction for each of the editable pieces - this copies the element across, adds the 'position trace' and 'edit dialogue name' tracking attributes and then recursively processes children for more matches. Similarly the view generator is responsible for arranging that only editable pieces are selectable, how selection might be indicated, attachment to the correct edit dialogue etc. It is similarly built from the editing description as an XSLT transform with templates to match result elements which request various editors. And unsuprisingly all the other components - edit dialogue generators and effector programs are built from the edit description as XSLT transforms.

6. EDITING: LOCAL & REMOTE

'Everything as a web service' is a common mantra, and we need to consider whether the editing of variable documents can be supported similarly. Luckily, establishing a few fairly simple interfaces can make this possible, at least for simple forms of editing such as via dialogue boxes. Much depends upon the expected smartness of the browser client. There are a number of options from completely dumb (using merely (X)HTML and forms), through standard supported extensions (such as Javascript or Flash), to a browser-specific specialist plug-in. But the essential approach remains similar. The dumbest options require large amounts of communication between client and server, the smartest fewest.

Firstly we need to enable the browser to display the result document. If SVG is assumed available then that is preferable, but a server-side image rendition of the viewed document (an image for each page) can of course be employed. Then a simple HTML page with the image referenced will suffice.

Secondly we need to make editable regions of the document view sensitive. At the simplest we can provide an image map, though we will have to arrange the map order to account for specificity of action. In this case we link each area back to a specific request for a new dialogue.

Thirdly, we need to define the dialogue that will perform the edit. In the simplest case this might be a complete HTML page with some suitable form elements. In more complex situations this might be an abstraction of the edit 'form' which is then interpreted client-side.

7. EDITING: THE EDITABILITY

A common requirement in large document workflows is that different users have different capabilities of modifying documents, and that some may have the ability to control what editing other (lower power) users can perform. Sometimes this is on a large class of documents, sometimes it might be on a specific template or a given instance. The architecture discussed here can support this with comparatively modest extra facilities. We can split the problem into two parts: i) different editing powers for different users and ii) controlling the editability scope within a document. The first is comparatively trivial - we could write different descriptions for different roles, but it's better to generate different projections from the same description. In our example we add some extra decorations:

```
<target match="cross[@fill='white']"
e:edit-level-required="user">
  <bindings>
    <bind name="stroke" nodeset="@stroke"/>
  </bindings>
  <group>
    <input label="Size" bind="size"/>
    <checkbox label="Rotated" bind="rotate"/>
    <button label="Delete" action="delete-element"
    e:edit-level-required="master"/>
  </group>
</target>
```

Figure 19. Role-variable editing

We indicate that white crosses are only editable when some 'power level' equals or exceeds 'user' level and that the ability to delete an element requires master powers. It is comparatively trivial then to generate different projections of the editability description for

different user roles and thence to completely different sets of components of the editor, by evaluating these predicates on various elements. (The technique can use many different models: levels, capabilities etc. with trivial changes in the underlying machinery).

If we want one user to be able to control the editing scope for another then again a modest addition to the declaration can support it:

```
<target match="cross[@fill='white'][$user-level ge
(@e:edit-requires,0)[1]]">
  <bindings>
    <bind name="edit" nodeset="@e:edit-requires"/>
  </bindings>
  <group>
    <select1 label="EditLevel" bind="edit">
      <item>0</item>
      <item>1</item>
      <item>2</item>
    </select1>
  </group>
</target>
```

Figure 20. Setting the editability within the document

In this case we arrange for the addition of another optional property `e:edit-requires` to the white crosses. Assuming there is a global variable `$user-level` set to the current level of the user's power, then if the user's level is below that recorded in a `e:edit-requires` attribute on a given white cross in the source, then no 'tracking' annotations will be added to that white cross and it will thus be uneditable *by construction* for that user, though of course other white crosses requiring lower levels (with a default here of 0) might be. Similarly the `e:edit-requires` attribute on an element can be edited *just like any other property*, so a simple added edit control is all that is required to support this.

Finally, and incestuously, the editability descriptions are XML documents, so we could develop a graphical view for such documents, and an editability description for elements within them and hence develop and modify editability declarations graphically.

8. LIMITATIONS & SPECIAL TREATMENT

This technique clearly depends heavily on pieces that are editable appearing as direct elements in the source XML tree and corresponding elements appearing in the result XML document instances. But there are other forms of generation that might be used for which this is not the case. The first of these is where an element is constructed (`xsl:element` is the XSLT instruction) and the type might be data dependent. This would be difficult, but not impossible, to track and would require understanding of XSLT semantics to alter the template 'code' to inject appropriate annotations.

8.1. Style Sets

The most important practical feature that requires specialist treament is editing style sets, the use of which should be encouraged to increase document repurposing. Within XSLT such styles are usually coded as `xsl:attribute-set` elements, which contain named sets of applicable properties and can be cascaded through a simple static inheritance graph. These styles are used by result fragments referencing applicable attribute sets by name - these properties can be further overridden by element-specific values, or additional child attributes (Figure 21 where 'attribute' has been shortened for compactness). So if we select a text block and want to edit its font-size, where can we look ?

```
<xsl:attr-set name="base">
  <xsl:attr name="font-family">Times-Roman</xsl:attr>
  <xsl:attr name="font-size">9pt</xsl:attr>
</xsl:attr-set>
<xsl:attr-set name="A" use-attr-sets="base">
  <xsl:attr name="font-size">11pt</xsl:attr>
</xsl:attr-set>
<xsl:attr-set name="B">
  <xsl:attr name="fill">green</xsl:attr>
</xsl:attr-set>
<fo:block font-size="12"
          xsl:use-attribute-sets="A B"> Some text
          </fo:block>
```

Figure 21. Styles in cascaded attribute sets

The font-size could be defined directly on the text-block itself. In this case we can alter it directly in the usual way. Alternatively it might in our example be contained in any of the attribute-sets `A`, `B` or `base`. To actually determine where the definition is to edit, we would need to 'thread' the attribute sets in the source document, retrieving them in appropriate priority order. For simple documents we can do this by following the 'pointer' to the `fo:block` and then searching around the XML document tree to determine the attribute set order. But if multiple documents have been merged to produce a result (often styles are held in one document but referenced from another) then this is much more complex.

It becomes clear that these stylistic interpolations can be edited by the current approach, but only with enhanced understanding of the document semantics - in this case the editing program elements must be aware of the semantics of XSLT and its `attribute-set` and particularly its non-local nature[6]. Choice of the style to apply to a given element (if any) is altogether much simpler - this is merely editing the `xsl:use-attribute-sets` property and is little different than any other property as regards editing.

8.2. Editing Variable Data

With the architecture described we can edit static text and the 'mapping' of dynamic text, i.e. those elements within the variable data for a given instance from which we wish to interpolate text. Can we use this mechanism to support editing the *variable data* itself, such as changing the price for the 'FlyBy' tour, directly from the result document? We could generate a specialist view of the data 'record' for 'FlyBy' and edit it as a static XML document, but with some additional knowledge of (and modification of) the document template semantics we can do this consistently. Note that we track editable source elements into the result though annotations added by specialist stages in the workflow. We can arrange that elements in (XML) data records be annotated similarly. To complete the process, we need to alter the document template to pass these annotations across into the result elements. Figure 22 shows the modifications required to a simple `fo:block` operating within a contextual iterator to track data dependency

[6] It can be highly dynamic - both values *and names* of attributes can be computed from application context - such dynamic uses would essentially be uneditable in the proposed architecture.

```
<xsl:for-each select="address-line">
  <fo:block>
    <xsl:sequence
    select="@e:data-posn,@e:data-editor"/>
    <xsl:value-of select="."/>
  </fo:block>
</xsl:for-each>
```

Figure 22. Tracking variable data causality

where the attributes `@e:data-posn` and `@e:data-editor` may appear on (pre-processed) editable data, and point to the data position and necessary editor respectively. This scheme can be extended with increasing understanding of the semantics of the source template.

9. PRIOR ART

The most common user-controlled variable-data documents are probably those used for 'mail merge' in word processors like MSWord. In this case the template is created with reserved structures (often with specialist field codes) to indicate the presence of a variable interpolation point and how the value of the variable should be interpolated. Typically this is then processed using a 'wizard' to complete the mapping process and project the output data. These variables are usually only interpolating text, which means the styling is taken from the text in which the reference is written, and the layout (usually a text-flow) is again taken from the context.

Recent developments in variable data publishing technologies have introduced a small number of products that support a more general solution. Pageflex[6] provides standalone and web-deployed tools for editing presentation material in a grounded-layout and text-flow copyhole model, with support for editing configuration and a suite of standard tools. DialogueLive[7] supports role-based editing of document instances against a computed variable document model, which is extensible.

Quint discusses the use of structured editing in XML[8], where DTD-driven constructor programs are created on a data instance within the Amaya editor. He and his colleagues also approach editing documents where the stylistic and to some extent layout appearance of the document is defined in a separate CSS stylesheet[9]. In this case the CSS stylesheet is analysed to produce an inversion, to discover *which* part styled a particular result element.

When the document processing is performed by XSLT, as is increasingly the case in the XML world, theoretical studies of incremental and reversible XSLT execution are relevant. Villard & Layaïda [10] describe an approach to incremental processing which recomputes only those sections of output that need to be from changes to source or transformation. The use of an execution flow tree structure could perhaps be used to trace back from the output to causal points in source document or transformational template and thus support template editing from interaction in the result document.

Source-code debugging of programs[11] has employed some similar ideas - the compiler (which of course has complete knowledge of the language semantics) can modify the code to include additional tracking information to trace back to source code (usually line/column) and add extra sections to trigger suitable (and differentiated) breakpoints, making it possible to run program code under inspection. Means to make the tracers more general and less language/processor specific have been explored for many years [12]. However they don't support a general approach to 'what caused this piece of the final output' design queries.

10. STATUS, FUTURE & THANKS

The architecture discussed is operational and has been used as the basis of some current field trials for models of variable document creation services, exploiting the client-server interface described. An extensible compiler is used to create all necessary program elements and a multi-view, multi-document framework exists which permits a wide variety of document types to be edited.

Future directions include adding models for more interactive editing, such as mouse-based manipulation (drag-n-drop, rubber-band), supporting 'partial rebuilding' of documents (to decrease editing response time) and exploring the use of 'document type' information within the framework to support larger-scale authoring and exploring multiple-view editing.

The authors are grateful to Philip Fennell and the development team in HP Brazil, led by Alexis Cabeda and Eduardo Lutz, for their valuable help respectively on the implementation of early trial and development versions of this framework.

11. REFERENCES

[1] Lumley, J., Gimson, R. and Rees, O. A Framework for Structure, Layout & Function in Documents . In *Proceedings of the 2005 ACM symposium on Document engineering*. 2005.

[2] W3C, World Wide Web Consortium *XSL Transformations (XSLT) Version 2.0* . http://www.w3.org/TR/xslt20/. 2007.

[3] Lumley, J., Gimson, R. and Rees, O. Extensible Layout in Functional Documents . In *Digital Publishing, Proc. of SPIE-IS&T Electronic Imaging, Vol 6076*. 2006.

[4] W3C, World Wide Web Consortium *XML Path Language (XPath) 2.0* . http://www.w3.org/TR/xpath20/. 2007.

[5] W3C, World Wide Web Consortium *XForms 1.0 (Third Edition)* . http://www.w3.org/TR/xforms/. 2007.

[6] Bitstream Inc. *Pageflex* . http://www.bitstream.com/publishing/products/pageflex/index.html. 2008.

[7] Exstream *DialogueLive* . http://www.exstream.com/Products/DialogueLive/. 2008.

[8] Quint, V. and Vatton, I. Techniques for authoring complex XML documents. In *DocEng '04: Proceedings of the 2004 ACM symposium on Document engineering* pages 115--123 ACM. 2004

[9] Quint, V. and Vatton, I. Editing with Style . In *Proceedings of the 2007 ACM symposium on Document engineering*. 2007.

[10] Villard, L. and Layaïda, N. An Incremental XSLT Transformation Processor for XML Document Manipulation . In *Proc. 11th World Wide Web Conference, Honolulu*. 2002.

[11] Hennessy, J. Symbolic Debugging of Optimized Code. In *ACM Trans. Program. Lang. Syst.*Vol 4 , no 3 , pages 323--344 ACM. 1982

[12] Ramsey, N. and Hanson, D. A retargetable debugger. In *SIGPLAN Not.*Vol 27 , no 7 , pages 22--31 ACM. 1992