Cascaded Dynamic Templates for Active Documents

John Lumley, Alfie Abdul-Rahman

HP Laboratories
HPL-2009-143

Keyword(s):
XSLT, SVG, Document construction, Functional programming, Document editing

Abstract:
Documents that are intended to be 'active', with high variability and context responsivity, are increasingly attractive building blocks for applications, inevitably defined in XML syntaxes. But many such documents within an application will have features in common, such as branding, models of variablity and user interaction, which need to be defined in single locations. We outline an approach using cascaded networks of 'templates', each of which is a first class document and containing fragments of functional programs in separate spaces, that can support sharing all necessary information, including dynamic behaviour.

External Posting Date: July 24, 2009 [Fulltext]
Internal Posting Date: June 21, 2009 [Fulltext]

Approved for External Publication

Cascaded Dynamic Templates for Active Documents

John Lumley, Alfie Abdul-Rahman
Hewlett-Packard Laboratories
Long Down Avenue, Stoke Gifford
BRISTOL BS34 8QZ, U.K.
{john.lumley,alfie.abdul-rahman}@hp.com

ABSTRACT

Documents that are intended to be 'active', with high variability and context responsivity, are increasingly attractive building blocks for applications, inevitably defined in XML syntaxes. But many such documents within an application will have features in common, such as branding, models of variablity and user interaction, which need to be defined in single locations. We outline an approach using cascaded networks of 'templates', each of which is a first-class document and containing fragments of functional programs in separate spaces, that can support sharing all necessary information, including dynamic behaviour.

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

There is growing interest in *document centric* applications to make automated business operations more accessible to the non-IT specialist. *Ad hoc* solutions based on standard passive office tools - word processors and spreadsheets and semi-active distribution services such as email servers, coupled with lightweight macro and similar languages have been used by many in enterprises, but they suffer from poor coherence models, poor quality in security and robustness and insufficient 'activity' to supplant purpose-developed centralised workflow applications. Consequently much opportunity for business *flexibility* is lost or exploited inadequately.

A new class of *active documents* is being explored, through extensions and amalgamations of widespread extant formats[1] or in novel new designs[2]. The intent is that much of a medium-scale application can be built by configuration and combination of documents and document fragments, all control declared entirely within the documents themselves. Such documents will include several different 'spaces' such as presentation, content, data and behavioural response, as well as attachment to external resources and workflows. XML representations and technologies are critical parts of most contemplated designs.

Our research raised the problem of how and where to define the vital *commonality* amongst such documents in an application, especially on coherence of appearance ('branding') and representation of internal state and state-change mechanisms. This paper covers the issue of defining a representation and mechanism for relating the common behaviour and presentation, placing such representation in the documents themselves. We start by describing the main document model and operations thereon and then introduce our model of *templating*. We assume familiarity with XSLT[3].

1.1. Example Document Model

Our simplest document is defined by a single self-contained XML tree - all the 'state' of a specific instance of such a document is contained within that tree, as well as all necessary declarations about the document's behaviour[1]. Thus a document that had been bound to some variability (e.g. a specific customer) contains somewhere within it all necessary information for that customer. Apart from existential operations (create, delete) our design supports two primary operations on a document:

- *Presenting:* generating a view of the document may involve a requested format or channel characteristics and other parameters such as the role or permissions of the viewer. The document contains declarations about how to respond to such requests: alternate presentations for different roles, interaction elements which could trigger changes of state, projections of other parts of the document's state and simple arrangements of layout.
- *Updating:* external *events* may cause a document to update its state in response[2]. Documents are expected to contain declarations on how (and whether) to modify themselves as a result of such an event. For example an event/request might be '*change variable v1's value to 1001'* - our expectation is that of a closed object model - the document contains information about variable *v1* (if it exists) and may decide to honour the change request; if the variable is described as read-only then the request will presumably be denied (as indicated somewhere around the variable *inside the document*); if it is rewritable then the value element will be rewritten (and other consequences propagated); if it is *single assignment* then the assignment type indication for that variable will be altered to read-only after the update.

[1] In practice documents will be held in multi-resource containers (such as ZIP) with unique naming of 'parts' - this can be placed in one-to-one correspondence with a tree

[2] How that document has been registered for such notification is another matter. It could be a request directed from a user, another document, some time 'alarm'. ..

```
<doc title="Sample1">
  <variable name="v1" update="input">1234</variable>
  <presentation>
    <p>Introductory paragraph in document <xsl:value-of
    select="@title"/>
    </p>
    <ol>
      <li>Some list item</li>
      <li>Item with v1=<val select="v1"/>
      </li>
    </ol>
  </presentation>
  <update>
    <template match="presentation//li[last()]"
    guard="$event[@name='addList']">
      <copy-of select="."/>
      <li>New list item</li>
    </template>
  </update>
</doc>
```

Figure 1.A simple document with a state variable

Figure 1 is a simple example document, defined in XML, containing a simple variable (with some properties), a description of a (mostly HTML) presentation and a partial program to update the document in response to some declared event, in this case *addList*.

1.2. Generating Presentation & Updating State

The key is to build appropriate *programs* that will generate a presentation or an updated version of the document in question, modifying these programs in response to the declarations buried in those documents. We assume that those declarations may contain programmatic elements in some canonical language - in our case XSLT. Figure 2 shows the approach:

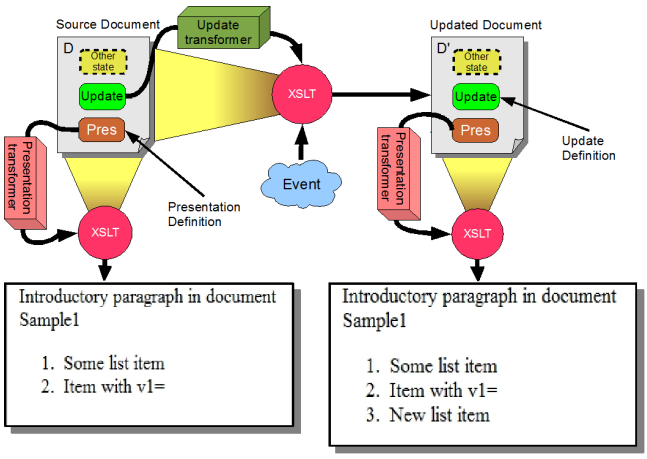

Figure 2.Generation of a document presentation and document update

For the presentation, we search for the presentation definition and build an executable (an XSLT transform) which is then executed with the original source document as the principal argument, and characteristics of the device channel as additional parameters. This produces an output in some final format, and thence to a viewable image either through a client (e.g. a web browser) or a server where a grounded geometry form such as PDF or an image is formed. The simple output from our example is shown lower-left in Figure 2.

The presentation definition contains an XSL fragment, which interpolates the value of the document's title (attribute), but the variable interpolating element `val`, after the 'v1=', is not used (it's actually in the presentation but ignored by the web browser) as there is nothing that 'understands' that concept, nor the link to the variable representation. The styling of the rest of the document is set by HTML and browser defaults.

Now we update the document in response to some event. Again we assume it may contain programmatic elements describing specific methods of update, in our example responding to an `addList` event by adding a new item just after the last item of a list. Processing is similar: extract the update definition from the document, convert it into an executable program and run it with the original document and the event as principal arguments[3]. The result can be in several forms - in the diagram we assume it is an updated document, but it could equally well be some definition of a workflow of which a new value for the document is one component. The presentation of the updated example is shown lower right in Figure 2. Note that the document is still able to respond to `addList` events, increasing the lists by one item per event.

These basic mechanisms are highly flexible, but to make useful applications we need coherent presentational styles and behaviours across a set of documents. In our example we might wish to:

- Style text elements
- Exclude or modify certain types of element.
- Add some common additional content
- Use `variable` as the declaration of a document-instance-specific state variable and interpolate its current value into a presentation (i.e. implement `val`)
- Update such variables coherently across many documents.

To do this we add dynamic, cascaded templating declarations to the documents and implement with recursion of the basic approach.

2. TEMPLATES IN PRESENTATION

Let's assume that we want a common style with some sidebar material, top-level paragraphs emboldened, alteration of the list numbering, variable values interpolated through the `val` instruction and an input form for a variable marked as updateable. That is we'd actually like our *sample1* document to appear as:

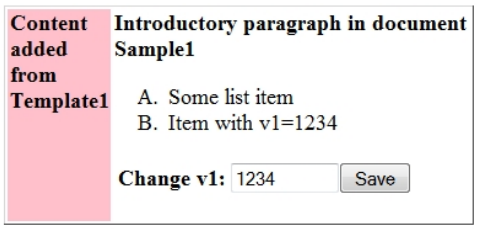

Figure 3.A templated document presentation

Some of these (the emboldening and the styling of the list elements)

[3] The update transformer should implement sensible defaults, such as providing identity transforms, so that the update definition in the document only need **describe parts that change during update**.

could be handled by simple attachment of a CSS [4] stylesheet, but the construction of the sidebar requires altering the document topology (surrounding the original content with a table cell), and the interpolation of the variable value requires some form of search of the document. Our solution is firstly to attach a template *Template1* (by reference) to the document. The template looks something like:

```
<doc title="Template1">
  <metadata>
    <showVars/>
  </metadata>
  <presentation>
    <style> p { font-weight: bold; } </style>
    <table frame="box">
      <tr>
        <td style="background-color:pink">
          <p>Content<br/>added<br/>from<br/>
            <xsl:value-of select="@title"/>
          </p>
        </td>
        <td>
          <TEM:apply-templates select="//presentation/*"/>
          <xsl:if test="//metaData/showVars">
            <TEM:apply-templates select="//variable"/>
          </xsl:if>
        </td>
      </tr>
    </table>
    <TEM:template match="val">
      <TEM:value-of select="$originalDocument//
      variable[@name=current()/@select]"/>
    </TEM:template>
    <TEM:template match="ol">
      <ol type="A">
        <TEM:apply-templates select="*"/>
      </ol>
    </TEM:template>
    <TEM:template match="variable[@update]">
      <form>... <input type="text">...</input>
      </form>
    </TEM:template>
  </presentation>
  <update>...</update>
</doc>
```

Figure 4.Presentation in a simple templating document

The template is still a document, but its presentation contains not only XSL programmatic fragments, but additional elements in a separate namespace (prefixed `TEM`). The semantics of these are the same as those of the usual (XSL) presentational program, but they operate *in a different phase* and *with a different source*. Some of these are independent templates, others (`apply-templates` in this example) are instructions to interpolate the intermediate presentation into the result. As the semantics are that of XSLT, considerable flexibility is supported - for example variables within the display could be sorted by use of `TEM:sort` directives.

These two spaces (`XSL` and `TEM`) can be intermingled and are executed in sequence. The `XSL` operates with the *templating* document as source producing a result which may contain `TEM` components. In our example the existence of a metadata directive `showVars` controls whether automatic variable updating should be included. (*Template1* could contain update code that would allow this directive to be deleted or inserted, enabling 'central' control of variable updating - as the templates are consulted dynamically this will work for *all* documents exploiting this template).

A program is then constructed from that intermediate result, 'warping' `TEM` components into the XSLT namespace and then executed with the *templated* document's presentation result as source. (There are some additional parameters available - *$originalDocument* allows access to the source tree of the original document.) The implementation is shown in Figure 5:

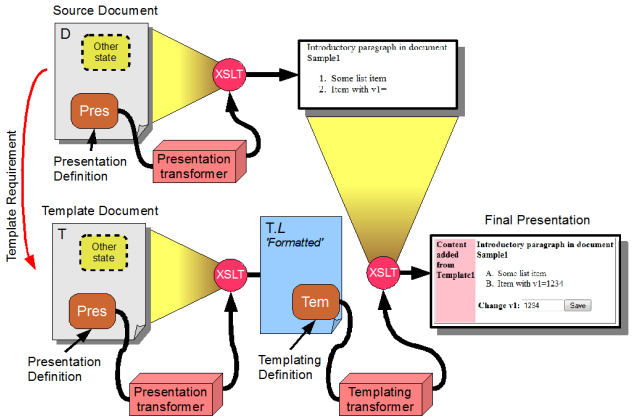

Figure 5.Template operation in presentation

The template may itself require a further template - then the process recurses, passing the accumulated final presentation as an argument. In this way an application may use a set of branding templates that each depend upon some core 'behavioural template' which might implement a coherent state-variable system for all documents in the application. There is more information available - in particular it is known at the point of generating a presentation as to whether this document is acting as a template for another document or is standalone - this could be exploited to generate different projections. For example a branding template viewed 'standalone' could generate examples of the different styles it contains.

What we've presented so far has a single thread of templates. We'll probably want to handle composing from several different template sources. If the programmatic definitions are functional (as in XLST), then we can merge elements from additional templates into the mixture.

3. TEMPLATES IN UPDATE

There are two different approaches to using templates in updating a document: either some similar recursively nested approach or a flat method where all the constructional elements from reachable templates are collected into a single program. Which is preferable depends upon whether the *templates* themselves are required to alter their own state. In our examples so far the templates are considered invariant and only the state of the 'leaf' documents changes, so we use the latter method: collect all updating fragments through the template tree when forming the updating program.

Figure 6 shows the updating fragment of our template example. The component rewrites the value of a variable if an (event) parameter names that variable as a target:

```
<doc title="Template1">
  <presentation>...</presentation>
  <update>
    <template match="variable[$param[@name eq
    current()/@name]]">
      <copy>
        <sequence select="@*"/>
        <value-of select="$param[@name eq
        current()/@name]/value"/>
      </copy>
    </template>
  </update>
</doc>
```

Figure 6.Update description of the templating document

The predicate in the template match ensures that only a suitably named variable is targetted. All template program fragments are considered to have similar priority - overloading and relative priority will be needed eventually. In the alternative approach, suitable when the templates have their own mutable state, and the result of the update is a 'workflow', we take the result of one stage as an input to the next templating stage, building up a composite workflow that may eventually end up modifying several documents. This also allows us to have deeper templates overriding decisions taken closer to the document instance.

4. DISCUSSION

This method of defining and implementing templates may seem rather complex and if this were merely limited to presentation it might be, but the power comes from the same template definition chain being used for document updating, meaning that a deeper template can co-describe both presentational and state-changing behaviour. Our example uses settable document-borne variables, but the underlying system (including the system transformers) *has no knowledge of the existence of such variables, nor how they are represented*. This is entirely described in the lowest-level templates - how to process updates and how to interpolate values into presentations (strictly how to generate the correct presentation for an agreed interpolating instruction).

As examples of the power of this approach, one of our applications added vector and structured variables into its documents, another added a full transclusion support system *as a single low-level template document*[5], both without alteration to the underlying platform.

Of course there are distinct similarities with an object-oriented design approach. Essentially we are much coarser-grained, exploiting functional programming properties and with a different dynamic system of inheritance. What we need from the combination of platform and document representation is agreement on:

- discovery, syntax and semantics of document-embedded program fragments for presentation and update
- canonical description of additional parameters for updating and presentation programs: events for update, device characteristics and 'roles' for presentation.
- canonical description of the result of executing these programs, e.g. presentation in a small number of standard formats or description of a consequential 'workflow'
- Declaration of template linkages within a document
- Consistent inter-document naming schemes.

Though this could be acheived in many ways, using XML-based techniques with reserved namespaces, attribute decorations and XSLT functionality makes implementation relatively easy.

5. PRIOR ART

Templates have long been used as the basis of document editing and publishing, typically in three roles: i) a starting point for the authoring of a new document, ii) the declaration of a named common set of styles and iii) the definitions of common constructs as *macros* - (parametric) sequences which effectively add some higher-level aspects to the 'language' of the document.

These roles are actually rather distinct but confusingly 'template' is often used for all three - some tools support only some of these aspects. MSWord and similar use templates for authoring initial states and common style sets only, TeX [6] and its precursors support macros and style sets.

The most common documents using 'active templates' are those used for 'mail merge' in word processors like MSWord or in VDP systems like Dialogue[7]. 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 and the platform (e.g. the MSWord application) executes the interpolation against a number of well-understood input forms, usually with a user-defined mapping. But in none of these cases can the form of the interpolation be extended or modified by document-borne declarations.

6. STATUS, FUTURE & THANKS

The architecture discussed in this paper is being used for experiments on active documents. Being able to add functionality purely in documents helps experimentation, such as generating audit trails and debugging by loading extra documents. In future we will need to refine the approach for using multiple templates, and updating documents deep in the templating chain.

The authors thank Roger Gimson for suggestions on the structure of this paper and Angelo di Iorio for exercising the approach.

7. REFERENCES

[1] Boyer, J. Interactive Office Documents: A New Face for Web 2.0 Applications . In *Proceedings of the 2008 ACM Symposium on Document Engineering*. 2008.

[2] Birman, K. et al. Edge Mashups for Service-Oriented Collaboration . *IEEE Computer*. Vol?, 90-94. May 2009.

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

[4] W3C, World Wide Web Consortium *Cascading Style Sheets, Level 1* . http://www.w3.org/TR/CSS1. 1999.

[5] di Iorio, A. and Lumley, J. From XML Inclusions to XML Transclusions . In *Proccedings of the ACM Conference - Hypertext 09*. 2009.

[6] Knuth, D.*TEX the program*. Addison-Wesley Pub. Co., Reading, Mass. 1986.

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