Evaluating Invariances in Document Layout Functions

Alexander J. Macdonald
Document Engineering Lab
School of Computer Science
University of Nottingham
Nottingham, NG8 1BB, UK
ajm@cs.nott.ac.uk

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

John Lumley
HP Labs
Filton Road, Stoke Gifford
Bristol, BS34 8QZ, UK
john.lumley@hp.com

Abstract

With the development of variable-data-driven digital presses - where each document printed is potentially unique there is a need for pre-press optimization to identify material that is invariant from document to document. In this way rasterisation can be confined solely to those areas which change between successive documents thereby alleviating a potential performance bottleneck.

Given a template document specified in terms of layout functions, where actual data is bound at the last possible moment before printing, we look at deriving and exploiting the invariant properties of layout functions from their formal specifications. We propose future work on generic extraction of invariance from such properties for certain classes of layout functions.

Categories and Subject Descriptors

E.1 [Data]: Data Structures Trees; I.7.2 [Document and Text Processing]: Document Preparation Markup Languages; I.7.4 [Document and Text Processing]: Electronic Publishing.

General Terms

Algorithms, Documentation

Keywords

XML, XSLT, SVG, Document Layout, Optimisation

1. INTRODUCTION

This research is concerned with optimising the rendering of template-based documents as exemplified in the DDF approach [1] [2] in which the placement of items in the document is defined by nested layout functions. An Example of such a function is an *X Flow*, which places each item in the flow to the right of the previous item.

FINAL DRAFT of paper accepted for:
DocEng'06, October 10–13, 2006, Amsterdam, The Netherlands.

In every case the geometric bounding box of an item is of crucial importance to the overall layout

In what follows we may assume, without loss of generality, that the final rendering paradigm will be SVG, which combines sophisticated graphics with the advantage that its XML syntax participates smoothly in an all-XML approach to the overall document. Our view of a document is then a very simple one. The **Item**s that make up the document are either a **LayoutFunction** or an **Atom**.

A simplifed BNF grammar for these types can be written as follows. Note that, for brevity, not all of the types are fully defined. It can be assumed that the types for position and dimension all reduce to a floating point numerical representation. Note that an **Atom** can consist of a partially- or fully-parameterised SVG document component.

$$\begin{aligned}
\mathbf{WidthBounds} &= (\mathbf{MinWidth}, \mathbf{MaxWidth}?)\\
\mathbf{HeightBounds} &= (\mathbf{MinHeight}, \mathbf{MaxHeight}?)\\
\mathbf{SVG} &= (\mathbf{Xpos}, \mathbf{Ypos}, \mathbf{Width}, \mathbf{Height})\\
\mathbf{Unbound} &= (\mathbf{Xpos}, \mathbf{Ypos}, ((\mathbf{Width}, \mathbf{HeightBounds})\\
&\quad |(\mathbf{WidthBounds}, \mathbf{Height})\\
&\quad |(\mathbf{WidthBounds}, \mathbf{HeightBounds}))?)\\
\mathbf{Atom} &= \mathbf{Unbound}|\mathbf{SVG}\\
\mathbf{Item} &= \mathbf{Atom}|\mathbf{LayoutFunction}
\end{aligned}$$

The use of such a simple document definition makes it easier to define the various layout functions that need to be examined and to prove where invariances, if any, exist. Since a layout function can be used anywhere an atom can be used it is possible to build up arbitrarily complex layouts. It is anticipated that certain combinations of layout functions will give rise to invariances that might not exist in the individual functions.

Given that a document might contain partially bound atoms it is not always possible to evaluate it to a fully grounded state (i.e. where there are only SVG atoms). However there may exist invariant parts of the document that can be evaluated and encapsulated[3], and this research begins to investigate just how much of this invariance can be identified and extracted.

2. IMPLEMENTATION

The system is implemented as an XSLT program which takes as its input a mixed namespace document containing both SVG and elements in its own *layout* namespace. It then processes this document, possibly with some extra data, and produces another mixed content document as the output. By successively re-evaluating the resultant document with additional data it reaches a stage where there is no more work to be done and the output is a fully grounded document which can be rendered by an SVG renderer.

Evaluation occurs in two depth-first passes over the document. The first pass replaces unbound items in the source document with any provided arguments that make these items become more fully bound. The result of this first phase of evaluation is then used as the input for the second phase. The second phase again recursively descends the document copying through, unchanged, anything in the SVG namespace and attempting to evaluate items in the layout namespace. Note that given the nature of the processing these two passes could be combined into one pass however they have been seperated out for reasons of code tidyness and ease of debugging.

The template matching system in XSLT is ideal for the above process because the code to evaluate each different layout function can be written as a matching template for that function e.g. $<\mathit{layout}:\mathit{xflow}/>$. These templates are separated out from the main program and included using the 'import' instruction in XSLT which allows different files to be included and executed as though they were a single program.

The semantics of this system are such that each layout function in a document is self contained and its ultimate geometric size is in no way affected by either its siblings or ancestors. This benefits the evaluation code because it is guaranteed that once a layout function has been passed through the layout processor and partially evaluated there is nothing more that can be done unless more unbound items become bound. Also, from a performance point of view, this allows for a high degree of parallelisation.

3. LAYOUT FUNCTIONS

Although the various layout functions that are implemented in this system take different types as their inputs they all evaluate to a single **Item**. If the function is fully invariant and capable of evaluation then the **Item** returned will actually be an **Atom** of sub-type **SVG**, otherwise it will be a (possibly modified) copy of the original **LayoutFunction**.

The first layout functions investigated here are linear flows. To evaluate a *Linear Flow*, where each item is fully bound, one would place each item adjacent to the preceding item in the flow. Interestingly even if there are unbound items within the flow there is still much that can be evaluated.

A *Linear Flow* exhibits associativity:

$$\mathit{linearflow}(x, \mathit{linearflow}(y, z)) \equiv$$
$$\mathit{linearflow}(\mathit{linearflow}(x, y), z)$$

This is important when extracting invariance because it means that adjacent, fully bound, elements can be evaluated as though they are a sub-flow, and grouped together.

In the following example b represents a fully bound item and u an unbound item:

$$\mathit{linearflow}(u, b, b, b, u, b, u) \equiv$$
$$\mathit{linearflow}(u, \mathit{linearflow}(b, b, b), u, b, u) \equiv$$
$$\mathit{linearflow}(u, b', u, b, u)$$

where b' now represents a bigger bound component composed from the three inner bs. It is also easy to see that a *Linear Flow* with a fully bound item at the head can be turned into a fully-bound item followed by a shorter *Linear Flow'* which starts from a translated origin.

$$\mathit{linearflow}(b, u, u, u) \equiv b, \mathit{linearflow}'(u, u, u)$$

With these two ways of extracting invariance we can see that the children of a *Linear Flow* will always end up following the regular expression pattern:

$$u\ (b?\ u)^*\ b?$$

where it is now understood that some of the bs may actually correspond to composite, merged, items (i.e. the b' of the discussion above).

Our investigation of linear flows has focused on two special cases, one where items are aligned with the X axis (an *X Flow*) and the other where items are aligned with the Y axis (a *Y Flow*). In the next section we examine a simple compound of these, and show how its evaluation proceeds as unbound items are replaced by fully bound items. Each diagram represents the state of the document after a certain number of passes through the layout processor.

It should be noted that the diagrams do not come from rendering a given file, but from passing the file in question through the layout processor in debug mode. This highlights the existence of layout functions and binds a wildcard element (in this case a rectangle with a question mark inside it) to any unbound item so the whole document can be fully evaluated for the purposes of visualisation.

In the following diagrams the rectangles containing question marks represent unbound **Atom**s. All other items are fully-bound pieces of **SVG**. To illustrate where the layout functions are – and what state of boundedness they are in – various styles of rounded rectangles will be used, as explained in the following table:

Dashed	Shows a part of the document where a layout function exists, still to be evaluated
Dotted	Represents a group of items which are all positioned but not yet fully invariant and which may change into a solid group as more data becomes available.
Solid	Represents a fully invariant section of the document.

Figure 1 shows the source document before any evaluation has taken place. The document contains an *X Flow* with five children, one of which is itself a *Y Flow* with three children. Some of the children are fully bound pieces of SVG but there are also two unbound items.

In Figure 2 we pass the source document through the layout processor with no additional data. This shows that even

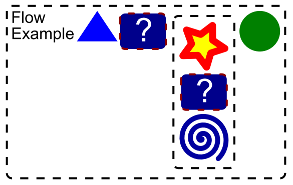

Figure 1: The source document.

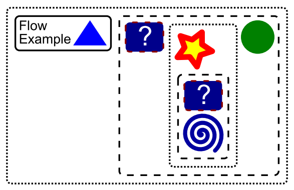

Figure 2: Document after processing with no additional data

though the document is not complete there is still a significant amount that can be evaluated. Note how adjacent bound items such as (Flow Example, ▲) are grouped together. Also note that the first fully bound item in both the *X Flow* and *Y Flow* is moved outside the flow.

In Figure 3 the unbound item in the *Y Flow* becomes bound causing the whole *Y Flow* to become bound. And because the *Y Flow* is adjacent to the circle in the *X Flow* both of them are encapsulated. This leaves a document with one *X Flow* which contains an unbound and a bound child.

Figure 4 shows the final step of processing which occurs when the last unbound item is replaced. This allows the remaining *X Flow* to be evaluated which leaves the document in a fully-bound ground state where no more evaluation can take place.

4. CONCLUSION AND FUTURE WORK

This work shows very clearly the possibilities for pre-press optimisation of advanced template-based documents in a variable-data-driven workflow. If for example a retail company wanted to customise its catalogue based on its subscribers' tastes and preferences they might create a variable data document which uses the information in their database to drive a digital press.

By analysing the document before printing begins it will be possible to take advantage of the bound and invariant sections (the $\mathfrak{b}$ items in the foregoing discussion) which can be identified as fully-bound SVG and possibly pre-rasterised. Even the late-bound sections ($\mathfrak{u}$), with customer-specific information, may still benefit from an analysis of the properties of each unbound item with respect to ranges of values of the missing arguments.

The underlying premise of our work is that each layout function operates on axis-aligned rectangles so, in principle, the layout process only requires to know the relative x and y coordinates of any given item and the relative coordinates of its bounding box. It may be possible to determine that some properties of a given rectangle's contents (e,g, fill colour, line width) are to be late bound and yet whatever values they eventually acquire cannot possibly affect the bounding box. Under these circumstances the layout process can still go ahead for later items in that particular flow and, if desired, a PPML job specification could be drawn up to help a digital press to optimise the rendering[4].

At the moment the code to evaluate invariances is hand-generated for each function. A formal standardisation and declaration of the interface to each function (including resources used, graphical and combinatorial properties e.g. associativity and an indication of the as-yet-unbound arguments) would enable SVG code generation to take place based simply on the values in the interface and without having to know the internal details of the function's coding.

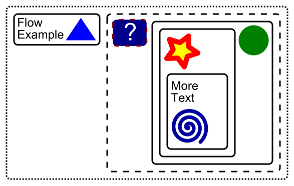

Figure 3: Document after first unbound item has been replaced

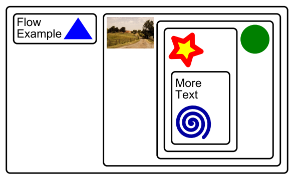

Figure 4: Final document after last unbound item has been replaced

5. ACKNOWLEDGEMENTS

Thanks are due to Hewlett Packard (UK) and EPSRC for supporting Alexander Macdonald's PhD studentship.

6. REFERENCES

[1] John Lumley, Roger Gimson, and Owen Rees. A Framework for Structure, Layout and Function in Documents. In *Proceedings of the 2005 ACM symposium on Document engineering*, pages 32–41. ACM Press, November 2005.

[2] John Lumley, Roger Gimson, and Owen Rees. Extensible Layout in Functional Documents. In *SPIE/EI 2006 Digital Publishing Conference*, January 2006.

[3] Alexander J. Macdonald, David F. Brailsford, and Steven R. Bagley. Encapsulating and Manipulating Component Object Graphics (COGs) using SVG. In *Proceedings of the 2005 ACM symposium on Document engineering*, pages 61–63. ACM Press, November 2005.

[4] Steven R. Bagley and David F. Brailsford. Page Composition using PPML as a Link-editing Script. In *Proceedings of the 2004 ACM symposium on Document engineering*, pages 134–136. ACM Press, 2004.