Ollis, James A .J. (2011) Optimised editing of variable data documents via partial re-evaluation. PhD thesis, University of Nottingham.

Optimised Editing of Variable Data Documents via Partial Re-Evaluation

James A. J. Ollis, BSc (Hons)

Thesis submitted to the University of Nottingham
for the degree of Doctor of Philosophy, 2011

Abstract

With the advent of digital printing presses and the continued development of associated technologies, *variable data printing* (VDP) is becoming more and more common. VDP allows for a series of data instances to be bound to a single template document in order to produce a set of result document instances, each customized depending upon the data provided. As it gradually enters the mainstream of digital publishing there is a need for appropriate and powerful editing tools suitable for use by creative professionals. This thesis investigates the problem of representing variable data documents in an editable visual form, and focuses on the technical issues involved with supporting such an editing model.

Using a document processing model where the document is produced from a data set and an appropriate programmatic transform, this thesis considers an interactive editor developed to allow visual manipulation of the result documents. It shows how the speed of the reprocessing necessary in such an interactive editing scenario can be increased by selectively re-evaluating only the required parts of the transformation, including how these pieces of the transformation can be identified and subsequently re-executed.

The techniques described are demonstrated using a simplified document processing model that closely resembles variable data document frameworks. A workable editor is also presented that builds on this processing model and illustrates its advantages. Finally, an analysis of the perfomance of the proposed framework is undertaken including a comparison to a standard processing pipeline.

Acknowledgements

I would like to thank my supervisor, David Brailsford, for his much valued support and guidance throughout the last 4 years. I would also like to thank the following people:

The members of the Document Engineering research group at the University of Nottingham, both past and present, whose suggestions and contributions have helped with this thesis as well as the various talks and papers I have written over the course of my research. Specific thanks are due to my co-supervisor, Steven Bagley, for his input and guidance throughout.

The members of the Web Services and Systems Lab at HP Labs Bristol for their help and encouragement during my numerous visits. Particular thanks are due to John Lumley, my industrial co-supervisor, for sharing his insight and knowledge as well as helping steer the research in the right direction.

The other PhD students and staff members that I have worked alongside at the University of Nottingham who have made suggestions towards this work, but more importantly have made the experience both enjoyable and rewarding.

Finally I must thank my friends and family for their continued support and encouragement, without which this work would not have been completed. I am especially grateful to my partner, Charlotte, for the help and understanding that she continues to give.

This work was funded by the EPSRC and Hewlett Packard Laboratories Bristol with additional help from the University of Nottingham.

Table of Contents

List of Figures

List of Tables

Chapter 1:

Introduction

Variable data printing is a recently developed technique, rapidly growing in importance, that allows for the production of high-quality personalized documents. This personalization has major implications on the way in which such documents are created and processed, and so new methods of authoring and editing must be considered. This thesis examines the inherent difficulties with creating and editing variable data documents in a user-friendly way and proposes techniques to support real-time WYSIWYG (What-You-See-Is-What-You-Get) editing by performing optimal re-processing of the editable document instance.

However, before discussing how variable data printing (VDP) is achieved we first need to recall how traditional printing is undertaken.

1.1 Traditional Printing Methodologies

High-volume document production has traditionally been the domain of offset printing presses, where the basic principle has not changed significantly for more than 200 years. This process of image transfer from metal plate to rubber sheet, and then finally to paper, has been gradually improved over the years so that modern presses can acheive very high throughput while still producing a high quality product. However this high speed and quality comes at the cost of inflexibility — all documents in the print run are imaged from the same plates and so are identical. In traditional situations, such as large-scale book publishing, this high-speed replication is a positive aspect of the technology. Unfortunately modern printing extends well beyond these basic principles into areas that are not well served by the traditional model. One example is seen in the world of advertising, where documents detailing products or services are more effective when produced specifically for each recipient, rather than sending out a generic version targetted at an, often non-existent, 'average' consumer. Studies show [1] that such targetted advertising has almost ten times the chance of generating interest than generically produced documents.

1.1.1 Modern Laser Printers

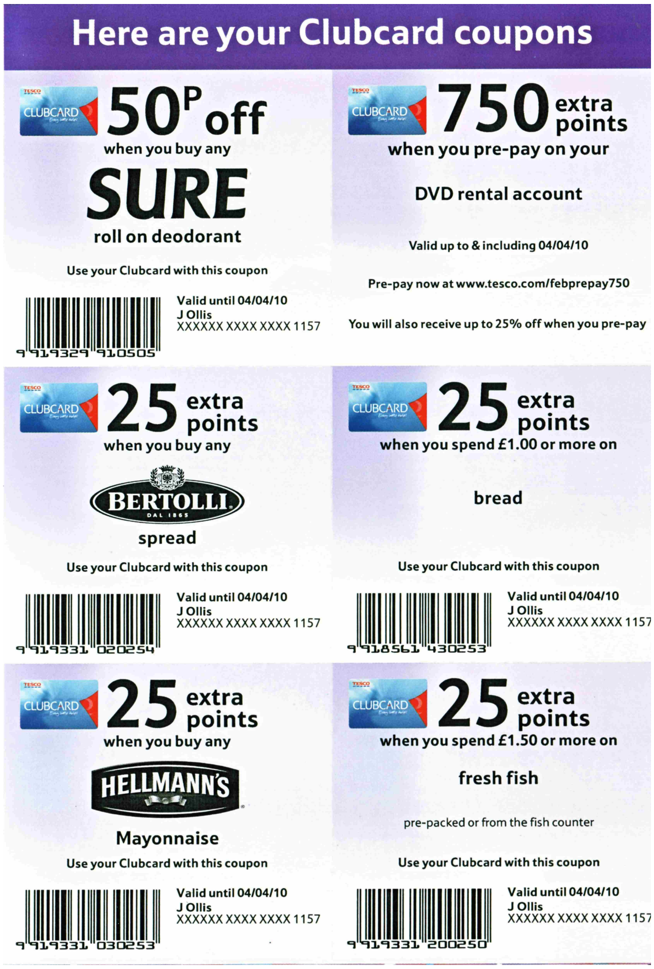
The opposite end of the printing spectrum from an offset press is the domain of home/office laser printers. Unlike offset presses, no metal plates are required to produce the image, but instead a laser and an electrostatically charged drum are used. By scanning the laser over the surface of the negatively charged drum and precisely controlling when the laser is active, areas with no electric charge are created on the drum. These areas correspond to the pixels in a mirrored version of the image to be printed and when the drum is exposed to negatively charged toner particles, the toner is repelled from the negative areas on the drum and collects only at the points on which the laser was shone. The toner is then transferred to the printing medium and fused to the paper by heating it with a *fuser*.

Laser printers produce documents that are generally of lower quality than those produced on offset presses, but they are much more flexible in their operation. The loss in quality and speed is more than compensated for by the decreased cost and complexity, especially when producing short-run or one-off documents. Clearly these types of printer are designed for a different purpose than large-scale offset presses, but there are instances where the strengths of both of these printing technologies are required.

1.1.2 Copy-Hole Printing

One solution to this apparent conflict between flexibility and quality/speed is to produce documents in a two-pass process: the first pass using an offset press and the second using a simple laser printer. This is the approach currently taken in a number of situations where the personalization is limited to simple information such as names and addresses. An example of such a document is given in figure 1.1.

In order for the second pass printing of the laser printer to be clearly visible, the initial offset print must contain a blank area into which this second printing can be placed. In the example document shown in figure 1.1, the variable content includes the details of each discount voucher, which can easily be identified as such by inspection of the original document. This type of printing is commonly referred to as *copy-hole* due to the 'holes' left in the base image into which the variable content is later placed.



Here are your Clubcard coupons

50P off
when you buy any
SURE
roll on deodorant
Use your Clubcard with this coupon
Valid until 04/04/10
J Ollis
XXXXXX XXXX XXXX 1157
9 919329 910505

750 extra points
when you pre-pay on your
DVD rental account
Valid up to & including 04/04/10
Pre-pay now at www.tesco.com/febprepay750
You will also receive up to 25% off when you pre-pay

25 extra points
when you buy any
BERTOLLI
DAL 1865
spread
Use your Clubcard with this coupon
Valid until 04/04/10
J Ollis
XXXXXX XXXX XXXX 1157
9 919331 020254

25 extra points
when you spend £1.00 or more on
bread
Use your Clubcard with this coupon
Valid until 04/04/10
J Ollis
XXXXXX XXXX XXXX 1157
9 918561 430253

25 extra points
when you buy any
HELLMANN'S
Mayonnaise
Use your Clubcard with this coupon
Valid until 04/04/10
J Ollis
XXXXXX XXXX XXXX 1157
9 919331 030253

25 extra points
when you spend £1.50 or more on
fresh fish
pre-packed or from the fish counter
Use your Clubcard with this coupon
Valid until 04/04/10
J Ollis
XXXXXX XXXX XXXX 1157
9 919331 200250

Figure 1.1 — Example copy-hole document

Although this is an improvement over the generic “no variability” situation, the amount of variability that can be introduced into the document is limited. Firstly, since the quality offered by the laser printer in the second-pass is relatively low, any variable content is usually limited to simple content such as monochrome text. Also, since the variable content is exactly that, i.e. *variable*, the copy holes left in the document produced by the offset press must be large enough to accommodate the largest name, address, etc. that might occur thereby running the risk that shorter instances of names etc. will appear to be drowned in excessive whitespace. In practice, things are often more difficult still. The process of moving the printed material from the offset press to the laser printer, for the second pass, makes misregistration an issue. Therefore, the copy holes are often made even larger than strictly necessary to avoid the variable content overprinting other parts of the document that lie outside the copy hole. In the example document, these issues are evident from the excessively large blank areas into which the variable data is placed.

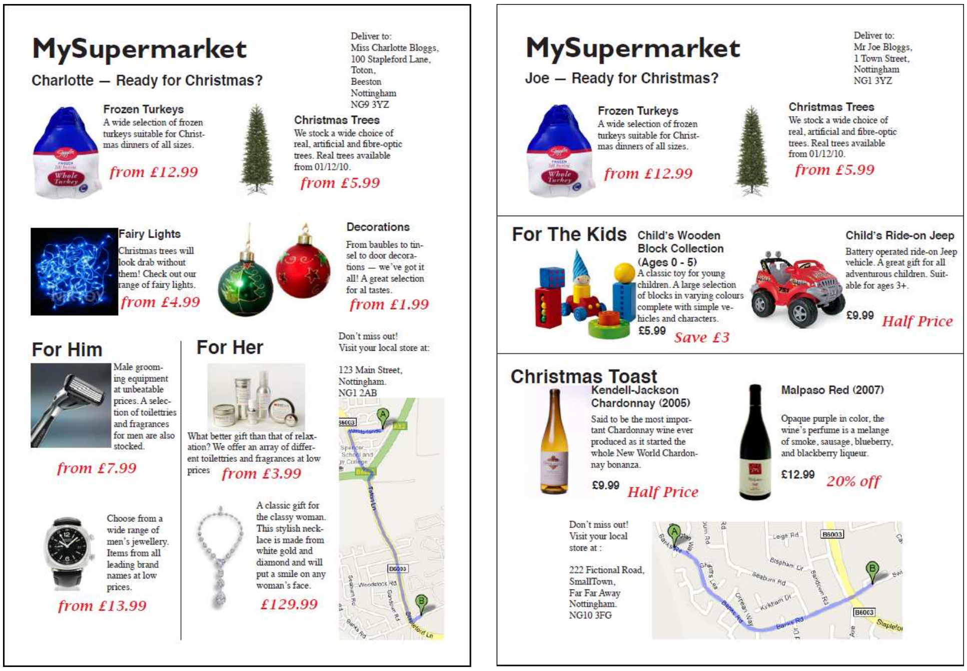
1.2 Introduction of the Digital Press

The holy grail for variable data printing was to develop a technology that married the speed and quality of an offset press with the flexibility of a digital laser/inkjet printer. In 1993, at the Ipex [2] trade show, two products were released promising this ability: the Indigo E-Print 1000 and the Agfa Chromopress. These *digital* presses did not require a metal plate to be produced, and so removed the main inflexibility of traditional offset presses. In essence, these digital presses can be thought of as extremely high quality laser printers that operate on a much larger scale. With no need for a metal plate, the cost and complexity of low-volume print runs need no longer be any more than that of high-volume runs. Taking this to its logical conclusion, it was now possible for every page to be different whilst still maintaining the high quality and speed associated with offset press printing.

In truth, these first digital presses could not reach the speed and quality of high-end traditional offset presses. However their modern counterparts are now sufficiently evolved that they are commercially viable.

1.3 Fully Variable Documents

The possibility of printing fully personalized documents on digital presses exposes the need for tools that can create such documents. Many document creation programs, such as Adobe InDesign, provide support for including simple areas of variable content; this is typically implemented as a placeholder component within the document that will be later replaced by the actual variable content. A more complete review of existing document creation applications is given in Chapter 3. To appreciate the problems and restrictions imposed by such an editing paradigm let us consider a document with a large degree of variability.



MySupermarket

Charlotte — Ready for Christmas?

Deliver to:
Miss Charlotte Bloggs,
100 Stapleford Lane,
Toton,
Beeston
Nottingham
NG9 3YZ

Frozen Turkeys

A wide selection of frozen turkeys suitable for Christmas dinners of all sizes.

from £12.99

Christmas Trees

We stock a wide choice of real, artificial and fibre-optic trees. Real trees available from 01/12/10.

from £5.99

Fairy Lights

Christmas trees will look drab without them! Check out our range of fairy lights.

from £4.99

Decorations

From baubles to tinsel to door decorations — we've got it all! A great selection for al tastes.

from £1.99

For Him

Male grooming equipment at unbeatable prices. A selection of toilettries and fragrances for men are also stocked.

from £7.99

Choose from a wide range of men's jewellery. Items from all leading brand names at low prices.

from £13.99

For Her

What better gift than that of relaxation? We offer an array of different toilettries and fragrances at low prices

from £3.99

A classic gift for the classy woman. This stylish necklace is made from white gold and diamond and will put a smile on any woman's face.

£129.99

Don't miss out!
Visit your local store at:

123 Main Street,
Nottingham.
NG1 2AB

MySupermarket

Joe — Ready for Christmas?

Deliver to:
Mr Joe Bloggs,
1 Town Street,
Nottingham
NG1 3YZ

Frozen Turkeys

A wide selection of frozen turkeys suitable for Christmas dinners of all sizes.

from £12.99

Christmas Trees

We stock a wide choice of real, artificial and fibre-optic trees. Real trees available from 01/12/10.

from £5.99

For The Kids

Child's Wooden Block Collection (Ages 0 - 5)

A classic toy for young children. A large selection of blocks in varying colours complete with simple vehicles and characters.

£5.99 Save £3

Child's Ride-on Jeep

Battery operated ride-on Jeep vehicle. A great gift for all adventurous children. Suitable for ages 3+.

£9.99 Half Price

Christmas Toast

Kendell-Jackson Chardonnay (2005)

Said to be the most important Chardonnay wine ever produced as it started the whole New World Chardonnay bonanza.

£9.99 Half Price

Malpaso Red (2007)

Opaque purple in color, the wine's perfume is a melange of smoke, sausage, blueberry, and blackberry liqueur.

£12.99 20% off

Don't miss out!
Visit your local
store at :

222 Fictional Road,
SmallTown,
Far Far Away
Nottingham.
NG10 3FG

Figure 1.2 — Separate document instances with large variability

The two documents shown in figure 1.2 are separate instances of the same (fabricated) variable data advertising leaflet designed for a fictional supermarket chain[1]. All of the instances have some common components, such as the leaflet heading, however much of the rest of the document is dependent upon the variable data.

Some of this variability is not much more complicated than that of typical copy-hole printing. For example, the recipient's name and their address (in the upper right corner of the page) are simple pieces of variable text. However, unlike copy-hole printing, the areas allocated to the name and address are dependent upon their size and the position of any other surrounding components is automatically changed if required.

Other parts of the document go beyond the simple notion of variability required for custom address fields. Inspection of the two document instances in figure 1.2 shows that there are sections present in one instance that are replaced by completely different sections in the other. For example, the first instance has sections for men's and women's gifts as well as an extended ‘generic’ items section. By contrast the second instance has a section of gift ideas for young children as well as a section advertising wines. The logic controlling which of these differing sections is included can be based on the known shopping habits of the targetted customer — in the example the recipient of the second leaflet might regularly buy wine and also items that indicate that they have a young child, whereas the recipient of the first leaflet might not exhibit such clear shopping habits. Clearly the scope for different advertising sections is vast and there may be a large number of possibilities, each one having a wide selection of products that can be advertised.

[1] This type of advertising is made possible through the ubiquity of store loyalty cards, from which a person's shopping habits can easily be discovered.

The final piece of variability in the document is the map included near the bottom of the page. It has been decided in this fictional scenario that each customer is provided with the address of, and a location map for, their local store. Although this seems superficially no more complicated than the inclusion of different products to be advertised, the example document highlights an important difference. Because the map can take one of three general orientations — landscape, portrait or square — the *layout* of the document must change to accommodate it. This is the cause of the male and female gift ideas sections in the first leaflet being tall and thin, rather than wide and shallow as is the case with the other sections. Extending this example beyond just two customers, it is conceivable that in the set of all customers there are some that share the same characterstics as the two shown, with the exception that the maps to be included are in alternative orientations. In these cases, the majority of the content of each document would be the same, but its layout would not.

This great variability in both the content and layout of the documents leads to new problems when creating and editing them. The most obvious of these is that an editing environment that utilizes blank placeholder components for variable content will result in the majority of the document not being shown when editing takes place. Chapter 3 looks at the problems related to visual editing of such documents as well as a variety of approaches taken by existing commercial document preparation tools and by academic research projects. The following chapters consider the need for an underlying processing framework to support an editing paradigm specifically tailored for variable data documents of the type shown in the example.

Document Description Framework (DDF)

As well as the problems in designing and editing variable data documents, there is also the question of how such documents are represented. Traditional document formats were designed to represent ‘fixed-form’ documents and extensions to incorporate variability into the document often consider only a small part of the document workflow.

DDF (Document Description Framework) [3] is a document format that provides support for a more complete workflow to be supplied within one file. The contents of the document are split into three sections: raw variable data, structural data with document-like semantics, and a final-form presentation. DDF allows for the inclusion of program code in these sections that will be executed at different stages of the processing pipeline in order to transform one section to the next. This *programmatic* capability is necessary to bind the variable data into the document when it is processed, but it also provides the possibility of producing largely diverse final-form documents as a result of differing variable data inputs.

This notion of defining the document as a series of programmatic transformations that operate on some input variable data has major implications for the editing process. No longer are we concerned with making changes to a series of declarative drawing operators (as is the case with formats such as PDF and SVG) or to an abstract markup such as DocBook or XHTML. In order to change the appearance of the document, we must instead make changes to the *code* that is embedded in the file such that when it is executed the resulting output will give the desired appearance.

In the case of DDF, the code to be modified is typically XSLT embedded within the XML notation of the rest of the document. Throughout the remainder of this thesis, we shall consider a simplified document format similiar to that specified by DDF. The idea of a programmatic (XSLT) transformation of some XML-based input to a final-form page description (SVG) is preserved, however the intermediate 'structural' form and the associated transformation are omitted for simplicity. Chapter 2 gives an detailed description of these technologies, which are required for the discussions in later chapters.

Given this programmatic nature of variable data documents, and the need to provide an intuitive editing environment as previously discussed, the later chapters of this thesis discuss a method of optimally re-processing a document in support of such an environment. The problems associated with such re-processing are considered and a solution is proposed and implemented into a working document workflow. This solution, and its implementation, are discussed in detail in chapters 4 through 9 and the performance analysed using a series of document scenarios in chapter 10.

1.4 Thesis Overview

This thesis is split into three major parts. The first gives detailed explanations of the technologies and languages that are utilized in later chapters. As a part of these explanations, there is discussion of how each of the technologies works, as well as a rationale for why each one is used.

The second part of the thesis describes the research undertaken and the results that were found. It finishes with an analysis of the performance characteristics of the various techniques and a discussion of how the techniques benefit VDP and what work could be done in the future to improve them.

The final section contains Appendices which provide extra information that is referenced throughout the thesis. A glossary of terms and abbreviations together with a bibliography of references can be found at the very end of the thesis. References throughout the thesis are numbered sequentially and can be found listed in numeric order in the bibliography.

Chapter 2:

Introduction to XML and XSLT

2.1 Extensible Markup Language (XML)

There are many instances where data must be stored in a structured way, and digital documents are no exception. There are many proprietary ways of achieving this structure, but there are substantial advantages to using a common metasyntax to describe it. XML is just such a metasyntax and it has become the *de facto* representation for structured data since becoming a W3C recommendation [4] in 1998.

XML defines a very basic structural syntax that can be used to construct custom tagsets. However it adds no extra information pertaining to the contents of the file — it merely provides a common metasyntax that allows many of the low-level, tedious tasks, such as parsing, to be handled by general tools rather than having to write custom parsers etc. to work with custom file formats. In particular, the lexical analysis of the document is easily be automated because of the standard metasyntax, and parsing is aided by the fact that the heirarchical tree structure guaranteed by a valid XML file is *deterministic* and *context free*.

The underlying data structure described by an XML document is a tree. The XML specification defines different types of nodes that can appear within such a tree structure, the most fundamental of which is an *element*. Elements are constructed using the following general syntax:

<*namespace*:*element_name* `attribute="foo"`> … </*namespace*:*element_name*>

where an arbitrarily long list of attributes can be provided using the same syntax as the one shown. The namespace component of the element name can be omitted if the element belongs to the default namespace (see next section) and attributes are not mandatory, therefore it is permitted to construct elements using only its name and the associated angle bracket syntax. The tree structure of the document is defined by the correct nesting of these elements such that an element that exists within another is a child node of that outer element. Figure 2.1 shows a simplified XML document that utilizes exisiting tagsets to illustrate these ideas.

```
<xhtml:html>
   <xhtml:head>
      <xhtml:title>XHTML &amp; SVG Example</xhtml:title>
   </xhtml:head>
   <xhtml:body>
      <xhtml:p>
         Hello World
         <svg:svg width="100%" height="100%">
            <svg:rect width="300" height="100"
                style="fill:rgb(0,0,255);stroke:rgb(0,0,0)"/>
         </svg:svg>
      </xhtml:p>
   </xhtml:body>
</xhtml:html>
```

Figure 2.1 — Example XML Document

2.1.1 Namespaces

Although XML provides a common metasyntax for defining custom tagsets, it adds no *meaning* to the contents of the document. As a result separate tagsets may define elements with the same name, but these elements have no intrinsic commonality and may be processed in very different ways by target applications. This can cause problems when XML documents are created that contain elements from multiple tagsets such as is the case in the example document. This is a frequent occurence in variable data documents where a single document may contain parts from various sources such as raw data files, processing code fragments, output page descriptions, typesetting directives etc. In such documents it is entirely possible for there to be elements, with the same name, that originate from separate sources and hence must be separately identifiable.

The solution to this problem is the concept of *namespaces*. A namespace is defined as an International Resource Identifier [5] that uniquely identifies a tagset. Elements can be labelled as belonging to a particular namespace in two different ways — either by prefixing its name with an appropriately declared namespace prefix, or by changing the default namespace. The example shown in figure 2.1 makes use two tagsets (XHTML and SVG), with the elements being subsequently decorated with the approriate namespace prefixes.

To declare a namespace prefix, the special `xmlns` attribute is used. A declaration such as `xmlns:foo="http://example.com/namespace"` indicates that `foo` is the prefix associated to the namespace `http://example.com/namespace`. Elements can then be prefixed with `foo:` to indicate that they belong to the specified namespace. It is worth noting that the prefixes have no meaning in themselves (only the IRI used has any meaning) and can be redefined to reference a different namespace at a later point in the document.

The alternative approach of changing the default namespace also uses the `xmlns` attribute, but no prefix is specified in its declaration. For example, adding an `xmlns` declaration such as `xmlns="http://example.com/namespace"` to an element causes all elements in the subtree of that element to belong to the namespace `http://example.com/namespace` unless otherwise specified by a local prefix.

2.1.2 Working with XML

The popularity of XML stems largely from the fact that it provides a common markup for structured documents, and as a result many common tasks are trivialised through the use of standard tools. Since all XML documents are formed using a common metasyntax, anyone wanting to work with the information contained within a document can access it using a standard XML parser.

There are two main approaches that can be taken when working with XML files; processing the content as it is read from the file, or building a model of the file's contents and accessing that model.

Simple API for XML (SAX)

SAX is an event-based API which defines handlers for the various events that are generated when parsing an XML file. The lexical analysis of XML documents is made standard due to their fixed metasyntax, therefore the generated events represent the results of this tokenization process. In order to do anything useful with the generated events, custom implementations of the various event handlers must be supplied by the user. These handlers are subsequently called when the corresponding events are produced by the parser as the XML document is processed. The events that are generated relate to the logical structure of the XML — for example when a start tag has been identified or when text content of a node has been parsed. For a complete specification of the generated events, the reader is referred to the official SAX technical documentation [6].

Because SAX generates events sequentially, and with no accompanying information regarding their context, the user is responsible for storing any relevant information that may be required when handling subsequent events. This limitation of having no access to previous events (and therefore no access to previously processed parts of the input) is one that springs from the implementation and philosophy of SAX. This approach can be likened to that of traditional parsers such as those produced by YACC [7].

By processing the XML document one event at a time, the SAX processor does not need to build a complete model of the document as part of its execution (though this can certainly be achieved by the user through appropriate handlers). This leads to several benefits which satisfy the primary design goals of SAX:

- Smaller memory footprint
- Processing can begin before the document is fully parsed
- Very large documents can be processed using limited memory

Document Object Model (DOM)

The benefits of SAX come at the cost of not being able to traverse the document tree without having first built an appropriate data structure using custom event handlers. As the name suggests, the DOM takes the alternative approach by automatcally providing the user with a model of the document where each node in the tree is represented as an object. These objects are related to one another as specified by the implicit tree structure from the document and can be queried and traversed accordingly.

Obviously the construction of the DOM tree must be completed before it can be accessed by the user and herein lies its major drawback — the initial processing and the overall memory cost are much larger than those for SAX. There has been research [8] into the possibility of producing the DOM "on-demand" to avoid the cost of producing parts of the document tree that are never accessed. However, in situations where the DOM tree is wholly visited or consumed, performance gains using such techniques are not possible.

As well as the random access permitted by the DOM, another feature is that a complete model of the document is produced, and thus can be retained for future queries that would have otherwise have required a full reparsing of the document. As will be discussed later, the process of editing XML-based variable data documents requires them to be frequently re-accessed and so the persistent document model provided by the DOM is preferable to the repeated reparsing that would be required by SAX.

SAX and DOM both provide simple access to the content of an XML document from within common programming languages such as Java. However, the ubiquity of XML has led naturally to asking whether an XML tree might be traversed, or analysed and re-processed, using yet another XML-based syntax. This in turn has led to the development of other tools and technologies specifically designed for the processing and manipulation of XML documents — a particularly important example is XSLT.

2.2 Extensible Stylesheet Language Transformations (XSLT)

In the late 1990s the W3C[1] began work on bringing substantial typesetting functionality to the world of XML in the form of the Extensible Stylesheet Language (XSL). The process of transforming an XML document, with no typesetting markup, into a form that could be rendered on screen or on a printer, was one that split itself into two distinct stages. Firstly, the XML file has to be transformed into a markup that describes the visual layout of the document. This resulting markup then needs detailed rendering. Therefore, instead of XSL being an all-embracing language to handle this entire process, it was eventually split into two separate parts — namely XSLT [9] and XSL-FO [10].

XSL-FO (Formatting Objects) was the language used for specifying the visual layout of the document, whereas XSLT was developed as a general XML-based tree transformation language. One of the main reasons for the separation of XSL into XSLT and XSL-FO was the realisation that XSLT had applications other than transforming XML directly into XSL-FO. In fact, since XSLT can be used to transform an XML document into *any* output XML tagset[2], it was commonly used to generate XHTML rather than XSL-FO.

As mentioned in the introduction, XSLT has been employed in current variable data document frameworks, such as DDF, as the programing language of choice. The fact that XSLT is itself an application of XML is a major advantage when processing documents that contain embedded program code as is the case with programmatic variable data documents. Since the document structure, its content,

[1] THe World Wide Web Consortium (W3C) is an international community with the purpose of developing web standards

[2] However, XSLT is not limited to producing only XML documents — it is also capable of outputting arbitrary plain text.

and the embedded code are all written in XML (albeit in separate namespaces), the document can be processed using the same standard tools that have been previously discussed. Furthermore, because XSLT is primarily designed for transforming XML documents into other XML documents — and an XSLT stylesheet is itself an XML document — it is entirely possible for XSLT stylesheets to modify and/or generate other XSLT stylesheets. This ability is often exploited and is frequently used in later chapters of this thesis. Essentially there are big advantages with transforming documents in an "all XML" environment.

2.2.1 Language Design

Although not strictly a functional language (since functions are not treated as first-class data types), XSLT's design is heavily influenced by various aspects of functional programming. An XSLT stylesheet comprises a set of templates that are matched against the nodes found in the XML document being processed. These templates are essentially pure functions that produce a fragment of the output given a fragment of the input tree, while not producing any other side-effects. The templates are evaluated when the XPath expression (a construct specifying a set of nodes) defined in their `match` attribute matches the node in the input tree that is currently being processed. Any output produced by an executing template is appended to the current node in the result tree.

One further feature of XSLT that differentiates it from common procedural programming languages is its lack of an assignment statement. The consequence of this is that the values of all variables defined in XSLT stylesheets are fixed and cannot be changed once declared. Michael Kay[3] explains this decision as follows [11]:

> *"The XSLT language allows variables to be defined, but does not allow an existing variable to change its value — there is no assignment statement. The reason for this policy, which many new users find bewildering, is to allow style sheets to be applied incrementally. The theory is that if the language is free of side-effects, then when a small change is made to an input document it should be possible to compute the resulting change to the output document without performing the entire transformation from scratch. It has to be said that for the moment, this remains a theoretical possibility, it is not something that any existing XSLT processor achieves."*

This goal of performing a limited amount of reprocessing to produce a revised output document is one that has significant implications when editing variable data documents and is explored and expanded upon later in this thesis.

2.2.2 XPath

XSLT is dependent upon XPath for selecting items that it can subsequently process. XPath itself is not a full programming language, but rather an *expression* language that must be hosted inside a "real" language, such as XSLT or XPointer [12]. However, its abilities go beyond simply referencing nodes in an XML document, because it also supports predicates and simple programming constructs that allow filtering and other processing. To accompany the update of XSLT from version 1.0 to 2.0, XPath was also revised. The specification of version 2.0 is much larger than its predecessor and provides a richer type system that is supported by a expanded set

[3] Michael Kay is the editor of the W3C specification of the XSLT 2.0 language

of functions and operators[4]. Therefore, since the XSLT examples given throughout this thesis are expressed using XSLT 2.0, any XPath expressions will consequently also conform to the revised 2.0 version.

2.2.3 Example Stylesheet

Before discussing some of the more detailed aspects of XSLT we now show an example program (see figure 2.2) that produces a simply formatted XHTML document from an XML file containing the marked-up contact information of multiple people.

The root element `<xsl:stylesheet>` indicates that the document is an XSLT program, with the `version` attribute specifying that it should be processed by an XSLT 2.0 compliant processor. Within the stylesheet, two `<xsl:template>` elements are defined. Each of these templates has a `match` attribute specifying the pattern that a node must match for the processor to consider applying the corresponding template when processing a given node from the input tree. In our example program, one of the templates matches on "`/`" (the document root) and the other matches on any `<person>` element.

```
<xsl:stylesheet version="2.0">
   <xsl:template match="/">
      <html>
         <head>
            <title>Contact Information Summary</title>
         </head>
         <body>
            <h1>
               Number of listed people:
               <xsl:value-of select="count(//person)"/>
            </h1>
            <xsl:apply-templates/>
         </body>
      </html>
```

[4] In fact, version 2.0 of XPath is a subset of the querying language XQuery 1.0 [13]

```
    </xsl:template>

    <xsl:template match="person">
       <h2>
          <xsl:value-of select="name"/>
       </h2>
       <p>
          Tel:
          <xsl:value-of select="telephone"/>
       </p>
       <p>
          Email:
          <xsl:value-of select="email"/>
       </p>
    </xsl:template>
</xsl:stylesheet>
```

Figure 2.2 — Example XSLT stylesheet

The content of each template is a combination of XSLT instructions and data. All elements in the XSLT namespace will be interpreted by the processor with any other elements being copied to the output document. In cases where XSLT instruction elements would generate children of output elements, the instructions will be executed and any resulting output is appended as child elements of the parent element.

In our example, the first template will be interpreted when processing of the input document begins since it has a match pattern of "/" that matches the document root. The template contains a series of elements that are not in the XSLT namespace and so are copied directly to the output to produce a simple skeleton XHTML tree. The `<h1>` element that is produced contains a text string composed of some literal text concatenated with the result value of an `<xsl:value-of>` instruction. This instruction generates a text node with the value of the expression given in its `select` attribute. In this case the `select` attribute utilizes the XPath `count()` function to count the number of `person` elements within the document. The XPath expression

`//person` is used to select all `person` elements in the document irrespective of their ancestry.

The template also contains an `<xsl:apply-templates>` element that causes the processor to process all child nodes of the currently selected node. The processor therefore recursively processes the children of the document root testing each element against the patterns of the various templates specified in the stylesheet. In our example, the two `<person>` elements match the pattern specified by the second template in the stylesheet and so it is executed once for each element.

The second template simply produces a `<h2>` element that contains the name of the person in question, as well as two `<p>` elements that contain the person's telephone number and email address. Since these pieces of information are stored within appropriately named child elements of the `<person>` element (see figure 2.3), they can be retrieved by the various `<xsl:value-of>` instructions by using `select` attributes that simply specify the name of the required child element.

It should now be clear that the stylesheet produces the output given in figure 2.4, when executed with the example XML file.

```
<people>
   <person>
      <name>James Ollis</name>
      <telephone>0115 9123456</telephone>
      <email>jao@cs.nott.ac.uk</email>
   </person>
   <person>
      <name>Sherlock Holmes</name>
      <telephone>020 1234567</telephone>
      <email>sherlock.holmes@example.com</email>
   </person>
</people>
```

Figure 2.3 — Example XML input file

```
<html>
   <head>
      <title>Contact Information Summary</title>
   </head>
   <body>
      <h1>Number of listed people: 2</h1>
      <h2>James Ollis</h2>
      <p>Tel: 0115 9123456</p>
      <p>Email: jao@cs.nott.ac.uk</p>
      <h2>Sherlock Holmes</h2>
      <p>Tel: 020 1234567</p>
      <p>Email: sherlock.holmes@example.com</p>
   </body>
</html>
```

Figure 2.4 — Example XHTML output

Path Expressions

In the example stylesheet, XSLT makes heavy use of XPath expressions to select and process nodes from the input document. Although XPath has a variety of expression types, the most fundamental are the basic path expressions used to select nodes. These expressions can be either absolute (starting with a `/` indicating the document root), or relative to the current context node.

A path expression comprises a series of steps that are used to navigate the tree. Each step consists of an axis, a node test and any number of predicates, such that the syntax of a path expression is as follows:

```
axis::nodetest[predicate][predicate]...
```

Axes are used to specify the direction in which the step should operate. There are 13 axes defined in XPath, with 11 used to support navigation of ancestors, decendants, siblings etc. and the other 2 to access attributes and namespaces respectively. If no axis is declared in the path step, the default is to use the `child` axis.

The next constituent part of the step is the node test. This can be used to specify the local name, namespace or type of nodes that should be matched at this stage.

For example, a step of `child::foo` will select all children with the local name of `foo`, whereas a step of `following-sibling::text()` will select all text nodes that are siblings of the current node but appear later in the document order. To select nodes of any name or namespace, the wildcard character “`*`” can be used.

The final part of a step is a list of predicates used to filter the nodes that were returned as a result of the axis and node test. Predicates are XPath expressions themselves and are evaluated once for each node, typically returning a boolean value indicating whether the nodes should be included in the final sequence returned or not. Predicates may also return numeric values specifying the index of the item to be returned from the sequence.

2.2.4 Stylesheet Evaluation

Further to the basic execution of the stylesheet covered in the previous section, we shall now discuss some more advanced aspects that are prevalent to the discussions in later chapters.

Push vs. Pull

Most programming languages use a pull-based approach where data from the input is explicitly fetched so that it can be manipulated and some result generated. XSLT can operate in this manner, but it is more commonly used as described earlier whereby the input document is parsed and the resulting tree is walked over, with templates executing when the pattern specified within them matches the nodes found in the tree. The key instruction used to instantiate this recursive, push-based processing is the `<xsl:apply-templates>` element. Unless nodes are explicitly specified via the optional `select` attribute and an accompanying XPath expression, the default behaviour is to find and execute the highest priority matching template for each of the current node's children.

This push-based approach can also be seen in older scripting languages such as *awk*[14]. Unlike XSLT, awk operates on input records rather than tree nodes, but the output of the script is governed by matching the input against a set of templates[5] much like in XSLT.

Template Priorities

Stylesheets often contain more than one template having a pattern that satisfies the currently selected node from the input tree. In these circumstances, the default behaviour is to evaluate the template with the most specific pattern; however the priority of templates can be overriden by the user via a `priority` attribute as part of the template definition. When priority attributes are supplied, the template with the highest value (among the set of matching templates) is executed. Once a template has been selected it assumes control for the input node producing any required output and controlling the processing of any descendent nodes in the associated sub-tree.

In order for the descendants of an already matched node to be processed, the selected template must contain appropriate instructions. This can be achieved by directly referencing the descendent nodes (pull-based processing), whether that be within the current template or through calls to functions and/or other named templates, or by returning control to the processor to find approporiate template matches for through the `<xsl:apply-templates>` instruction.

[5] *awk* does not prioritise templates in the way that XSLT does, instead they are simply checked for compatability in order of their appearance in the script

Modes

There exists an optional mode attribute that can be placed on templates to indicate that they may be considered for execution only when a matching mode value has been specified on a corresponding <xsl:apply-templates> instruction. This allows templates to be grouped and it is useful when performing logically distinct operations within the stylesheet. A good example is that of generating documents with an index and/or contents pages where the input document is processed once in the normal manner to generate the actual content, but is then processed again in a different mode in order to generate the index/contents.

As well as custom mode identifiers, there are a few "special" ones that are defined within XSLT. When no mode value is given the processor runs in the default mode, which has a value of #default. Templates can also be called with the special mode value of #current, which indicates that processing should continue in the current mode. The final pre-defined value is #any, which allows the template to be called irrespective of the mode value specified.

Tunnelled Parameters

XSLT supports the passing of parameters to templates and functions through the use of <xsl:param> and <xsl:with-param> instructions. This mechanism operates in the same way as most programming languages, however an optional tunnel attribute can be added to allow a parameter to be available in all templates called from that point on without the need for explicitly passing it through.

Let us consider the code example shown in figure 2.5 where the template named a is called[6] from the initial template with a parameter named foo. This parameter has a value of 'red' and is specified as a tunnelled parameter by the tunnel attribute. The <xsl:param> instruction within template a declares the incoming parameter, but does not specify it as being tunnelled. Therefore, within the scope of template a, foo holds the specified default value (in this case 'blue').

```
<xsl:template match="/">
   <xsl:call-template name="a">
      <xsl:with-param name="foo" tunnel="yes">red</xsl:with-param>
   </xsl:call-template>
</xsl:template>

<xsl:template name="a">
   <xsl:param name="foo">blue</xsl:param>
   <!-- $foo contains the value 'blue' -->
   <xsl:call-template name="b"/>
</xsl:template>

<xsl:template name="b">
   <xsl:call-template name="c"/>
</xsl:template>

<xsl:template name="c">
   <xsl:param name="foo" tunnel="yes">green</xsl:param>
   <!-- $foo contains the value 'red' -->
</xsl:template>
```

Figure 2.5 — Tunnelled variables example

Template b is then called with no parameters and template c is subsequently called in the same manner. Template c contains an <xsl:param> instruction indicating that it can accept a parameter named foo that will take a default value of 'green' if no parameter is passed to the template. However, it also has a tunnel attribute indicating that it can obtain the value from a previously tunnelled parameter.

[6] As an alternative to specifying templates with match attributes, which are executed when a template matches the input node being processed, templates can be given a unique name via the name attribute and called by a corresponding <xsl:call-template> instruction.

Although the parameter passed from the initial template has not been explicitly passed through the other templates, it has been automatically passed through because it was declared as tunnelled. Therefore the parameter accepted by template `c` is that initially passed from the first template and hence, within the scope of template `c`, `foo` has the value '`red`'.

Chapter 3:

Authoring Variable Data Documents

The previous chapter introduced some of the common technologies for producing and processing variable data documents. In this chapter we will examine how documents in general have been traditionally authored, the aspects of variable data documents that make traditional editing paradigms less than ideal, and how we might develop new frameworks for editing.

3.1 Existing Document Editors

One of the difficulties in reviewing existing document editors is determining exactly what constitutes a document and, therefore, which applications can be classified as document editors. Clearly, there are traditional documents such as such as books, reports, magazines, etc. with which we are familiar, as well as the relatively new variable data documents that are the subject of this thesis. There are, however, a large number of other types of media that can be justifiably classified as documents, in the sense that they convey information to a consumer. Items such as TV documentaries, multimedia Web pages, and computer program interfaces can all be thought of as documents and there are associated applications that exist to aid their creation.

In the following sections we look at a variety of document types, and their editing applications, under three broad categories: traditional manuscript-type documents, variable data documents, and other miscellaneous documents. The purpose of this exercise is to evaluate the strengths and weaknesses of the techniques associated with each type of document and assess how they might be of use when looking towards a new editing framework for fully variable documents.

3.1.1 Non-variable Document Editors

To most people, the word 'document' implies a fixed-form manuscript such as a letter, article or leaflet. This association of fixed-formedness, or 'non-variability', to the word 'document' is probably a consequence of the long-term existence of written and printed documents in contrast to the relatively recent advent of other document forms. With such a long-standing history, it is not suprising that their production and consumption has been readily transferred to software. Although the most familiar graphical WYSIWYG applications were not readily available until the mid 1980s, a variety of text-based document preparation systems was available as far back as the early 1970s.

3.1.1.1 Text-Based Document Preparation

Many document preparation systems such as $\TeX$[15] and *troff*[16] were developed before the desktop publishing revolution of the mid 1980s. These systems employ an authoring model wherein the textual content of the document is interspersed with formatting and control commands and is subsequently processed by the appropriate processing engine to create a final-form document (nowadays, this is commonly in the form of a page description language (PDL) such as PostScript or PDF). This processing pipeline means that there is a separation between changes made to the source document by the author and these changes being effected in the resulting output document. This separation was largely overcome by the development of WYSIWYG publishing applications, but systems such as TEX and *troff* are still used today.

The power of their typestting abilities combined with their simple textual format means that, although they are often not used in the way originally intended, they are still used as intermediary stages in complex document workflows. In this way, they can be seen as contempories of XSL-FO (see chapter 2), however *troff* and TEX macros can also support basic programmatic constructs, hence leading them into the domain of XSLT. It is therefore not suprising to find *troff*[1] and TEX used by publishers as a target typesetting language into which abstract documents are converted, thus allowing them to use a standardized processing pipeline to achieve a consistent appearance and style.

3.1.1.2 Desktop Publishing Applications

The tedium of having to completely reprocess a document to see the output effect of any changes made by the author was largely removed with the development of WYSIWYG desktop publishing applications. By the mid 1980s, desktop computers were available with sufficient memory capacity and processing power to support interactive authoring applications such as Aldus PageMaker[2].

[1] In practice, the freely available clone *groff*[17] is often used instead

[2] Aldus Corporation was taken over by Adobe in 1994 and therefore subsequent releases of PageMaker were released under the Adobe name

Since the introduction of these first desktop publishing packages, many more have been developed offering various refinements to the model of WYSIWYG document authoring, with recent applications such as Adobe InDesign and QuarkExpress becoming ever more complex and powerful.

Although the fixed-form documents we are considering here are a subset of a wider range of document types, there is significant variation within this set. Editors such as Quill [18] and Lilac [19] deal with purely textual documents like letters, books and articles, whereas others (e.g. Juno [20]) deal with more graphically oriented documents. There are also a large number of editors that support both text and graphics within the same document, aimed at allowing the user to create reports, Web pages etc. (e.g, InDesign, DreamWeaver, FrameMaker [21], Quanta+ [22], Amaya [23], *grif*[24] etc.). Most of these editors allow for editing to be performed in more than one mode, often supporting direct editing of the source as well as via the graphical view of the resulting document. Some [24,21] further expose the structure of the document (whether it is implicit or explicit) to the user through a separate view that also supports user modifications.

This support for concurrent source, structural and graphical views of the same document is something that can be easily maintained for fixed-form documents. Each part of the document has a representable form in each of the different views and the translation between these views is fairly straightforward. For example, if a new paragraph was to be added to the source of a document being editing in Lilac, it would result in exactly one formatted and typeset paragraph being added to the corresponding graphical view. If the situation was reversed and the paragraph was added through the user's interaction with the graphical result view, it is a simple task for the editor to insert the approporiate source code to produce the paragraph at the correct point in the source view.

Impact on Variable Data Document Editing

The main advantage of WYSIWYG editing is the simple fact that any changes made to the document are done through direct interaction with the resulting graphical view. This editing process allows a user to easily create documents without the need to know the underlying markup or language, as is the case with text-based systems. This allows creative professionals to concentrate fully on designing the document without having to concern themselves with the technical aspects of the document format. Such an editing model works well for the types of document we have discussed so far because of their implicit one-to-one relationship between the marked up document content and the resulting final-form image. Every time the document is processed, it will produce the same output and every component in the output will have been directly produced from one section of the source document. A classic example of a tool that follow this paradigm, and is widely used by a large number of users, is Microsoft's 'Word' [25] word processing package. Word allows users to create predominantly textual documents through direct interaction with a view of the result document without exposing them to its underlying file formats [26]. Although basic edits, such as adding new content or changing font, are effected automatically, other, more complex, operations such as updating the table of contents of a document must be initiated manually. However, the underlying principle that the editing view presented to the user is a faithful representation of the single possible rendering of the document remains.

When working with variable data documents, the basic assumption of a one-to-one relationship between the components in the source and result documents, on which the traditional WYSIWYG desktop publishing paradigm is founded, no longer holds. Repeatedly processing the source document does *not* necessarily produce an identical result every time, but instead can produce wildly different results depending upon the input data. Furthermore, and potentially more problematically, there may be parts of the source document that are not evaluated when using certain input data. The result of this can be that some components that would otherwise have been produced are not included in the result. Since the aim is for all authoring of the document to be performed via the result view, the problem is that certain result instances can preclude the possibility of editing all aspects of the source document. As an example, consider the sample document shown in figure 1.2 of chapter 1. Many of the variable product sections are included only when certain information about the customer is included in the data instance; it is not possible for *all* sections to be included in a single result instance.

Clearly the editing model widely used for authoring non-variable documents comes up against some significant obstacles when directly applied to variable data documents. However, the editing of the source document through the final result has proven to be an intuitive and effective method of editing non-variable documents, and this remains a goal for variable document editors. Fortunately in most variable data documents that are some components that do not vary. It is upon this premise that a number of add-ons to existing desktop publishing packages have been developed, to support varying degrees of document variability. We now look at these, along with a variety of purpose-built variable data document editors.

3.1.2 Variable Data Document Editors

There are a number of products, such as CatBase [27] and uDirect [28] that work in conjunction with existing non-variable document editors (Adobe InDesign, QuarkExpress, etc.) to provide a capability for supporting some form of variability. A typical example in which this process might be used is that of adding personal details to a template document to produce a 'mail merge' of the data and the template.

CatBase works by providing the author with a wizard to select the variable content to be included in the document. This is then linked into the document by adding it as a data source for a custom component. Once this is done, a set of documents is created with the data fully bound, ready for the author to print. In the case of uDirect, a plugin allows the author to specify the variable data to be included and it then utilizes its own custom component, which is added to the document as a placeholder. A series of previews can be generated for the author to use in proofing the resulting document before a full set of result documents is generated from the supplied variable data.

Both of these products highlight one of the main drawbacks to using existing editors. Because the editor was not designed with variable data documents in mind, the placeholder components that are included to represent variable content are displayed in a static, and usually non-illustrative, way. This means that the document author is not able to view the effects of the changing variable data without generating and proofing a subset of the sample instances. For simple copy-hole and minimal-variability documents this method is viable, but when we consider documents with a high degree of variability it is clear that the number of sample instances required makes this approach impractical.

In contrast to tools designed to work alongside existing document preparation packages, a number of bespoke editors have been created specifically aimed at authoring variable data documents. Dialogue Live [29] is a commercial package that allows for extensive editing and modification control and provides support for more wide-ranging aspects of the document lifecycle. However, because it has been designed from the start with the goal of supporting variable data, and other non-standard documents, it is able to provide the user with a more complete view of the document than a combination of placeholder components and a set of generated instances otherwise could.

The work of Lumley *et al*[30] follows this same idea of creating a bespoke editor for variable data documents — in their case building upon the DDF technologies that have previously been discussed. Their proposed editing application adds support for the visual display of programmatic concepts e.g. when a group of components was generated from an iteration in the source transform and are part of a linear-flow layout. Furthermore, this editor is designed to utilize an underlying editing mechanism that allows for the document author to make changes to both the document itself and to the variable data used to generate it. This is a step toward a truly WYSIWYG variable-data editor because the author can now change the instance that is being displayed through direct interaction with the document. There is, however, still scope to further develop this idea of interactive modification of the source data at a component-by-component level, as well as dealing with the significant problem of real-time updates to the document view. These aims, and the associated practical issues, will be discussed in more detail in later sections, when we consider a refined approach to variable data document authoring.

3.1.3 Alternative Editors

Beyond traditional printable documents, there are a host of other document types that share similar characteristics to those causing the difficulties for WYSIWYG editing of variable data documents. Although an analysis of these documents, and their authoring applications, will not necessarily provide direct solutions, it will be useful in further understanding the context of the problem.

Multimedia Document Editors

There is a substantial body of research in the area of multimedia documents and how they are created and consumed [31, 32, 33, 34, 35]. This type of document extends beyond the text and images associated with traditional documents and incorporates the use of sounds, video, animations etc. The temporal aspect of these documents introduces similar problems and concerns to those we have outlined for variable data documents.

The connection can be seen when we consider multimedia documents to be those that change from state to state, or instance to instance, as a result of temporal progression, in the same way that variable data documents change from instance to instance as a result of changing input data. In both cases, the goal of a WYSIWYG editing application is to provide the user with a means of editing the document through a simple interactive view even though the composition and layout of the document may have many potential configurations.

At first glance, it appears that we are indeed attempting to solve the same problems that exist when editing multimedia documents, but there is one significant difference. When editing a multimedia document, the author sees many different 'states' that the document can be in as time progresses, however each of these states is well defined and fully bound. This is in contrast to variable data documents where the input data can potentially contain *any* values unless restrictions were applied at some stage in data preparation and the document workflow.

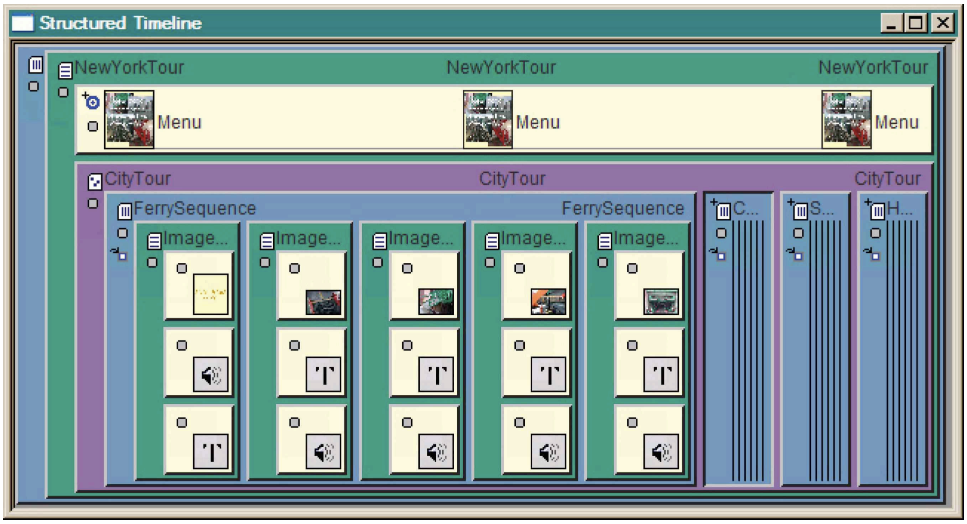


Figure 3.1 — Multimedia document authoring view

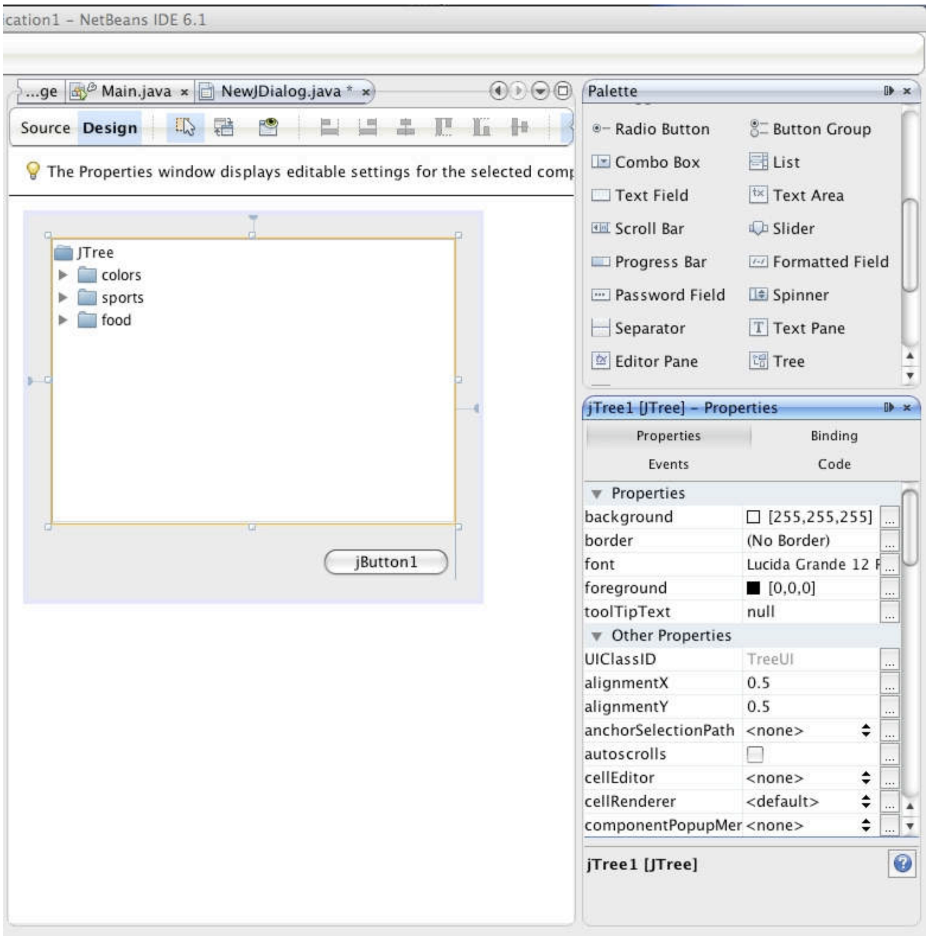
Figure 3.1 shows an interface for editing spatio-temporal multimedia documents proposed by Bulterman *et al*[36]. At first glance, such an interface does not resemble a WYSIWYG document editor. This is because it works with a graphical view of the document's *structure* rather than a final-form view. The various coloured containers represent different control structures, with the document components and associated content being shown within them.

Although this interface does not show a final-form view of the document and, by the authors' own admission, is hindered by its apparent complexity, it does show one way in which tree-structured documents can be displayed whilst allowing access to all 'branches' at the same time.

More 'conventional' looking multimedia document editors such as Adobe Flash [37] and the application described in [38] use a timeline to access the different transitional states of the document. If applied to variable data documents this approach would result in the same type of result instance enumeration that already exists in other plugins for variable data document editors that were discussed in the previous section.

Graphical User Interface (GUI) Designers

Another type of ‘document’ editor, seemingly unrelated to VDP editors, is that of user interface designers for software applications. Mainstream Integrated Development Environments (IDEs), such as Microsoft's Visual Studio and the open-source NetBeans platform, as well as bespoke designers such as TrollTech's QTDesigner all exhibit features that are potentially relevant to the problems discussed so far.

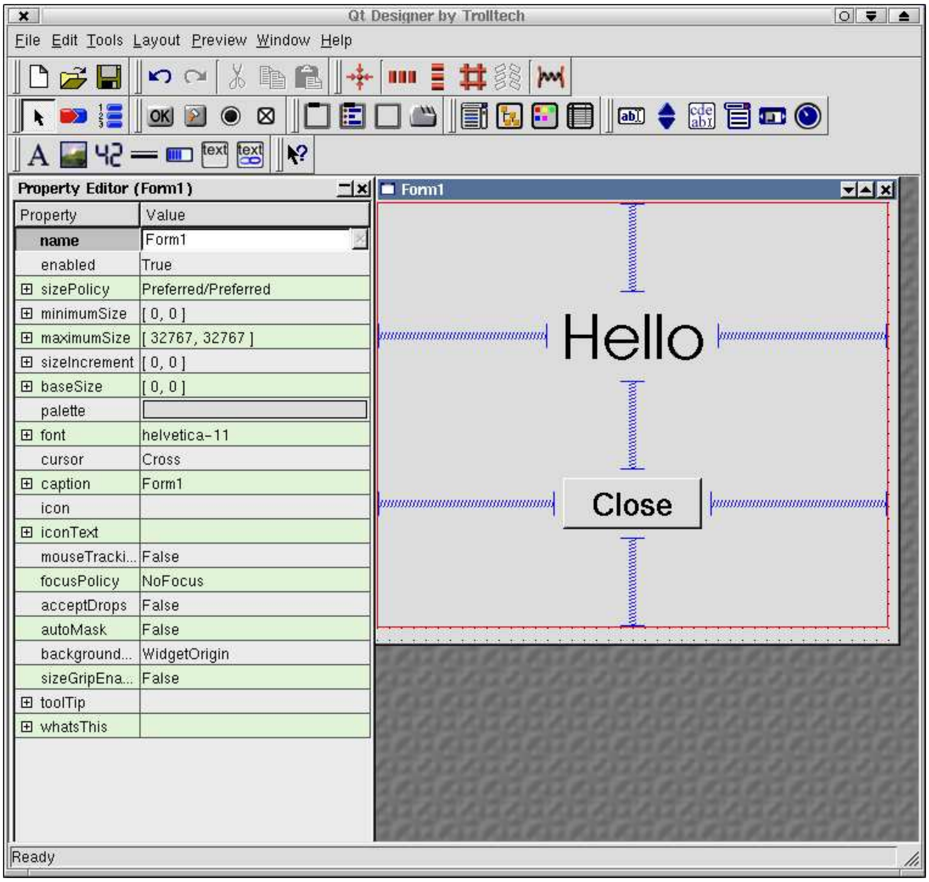


The Properties window displays editable settings for the selected comp

Properties	
background	□ [255,255,255]
border	(No Border)
font	Lucida Grande 12 F
foreground	■ [0,0,0]
toolTipText	null
Other Properties	
UIClassID	TreeUI
alignmentX	0.5
alignmentY	0.5
anchorSelectionPath	<none>
autoscrolls	□
cellEditor	<none>
cellRenderer	<default>
componentPopupMer	<none>

Figure 3.2 — User interface creation in NetBeans

The reality of GUIs is that they are the interface between the user and the content of the program with which they are interacting. In this sense, they are no different to printed documents and their programmatic underpinnings bring them closely in line with complex variable data documents. GUI designers like those shown in figures 3.2 and 3.3 often provide the user with a window canvas onto which components can be placed and organised. These are typically a combination of discrete interaction controls (buttons, menus etc.) and information input or display components such as text areas/fields, images, lists and tables. The information displayed by such components is linked to through other aspects of the GUI designer, or by direct manipulation from within the underlying source code. Changes to the currently displayed data, and/or other interactions with the user, often result in changes to the layout and composition of the interface. For example, GUI components may resize when their content changes and rules are often specified to describe the effect on the bounds of other components as a result. This process is analagous to that of changes to the layout and composition of a variable data document when changing from one input data instance to the next.



Property Editor (Form1)

Property	Value
name	Form1
enabled	True
⊞ sizePolicy	Preferred/Preferred
⊞ minimumSize	[0, 0]
⊞ maximumSize	[32767, 32767]
⊞ sizeIncrement	[0, 0]
⊞ baseSize	[0, 0]
palette	
⊞ font	helvetica-11
cursor	Cross
⊞ caption	Form1
icon	
⊞ iconText	
mouseTracki...	False
focusPolicy	NoFocus
acceptDrops	False
autoMask	False
background...	WidgetOrigin
sizeGripEna...	False
⊞ toolTip	
⊞ whatsThis	

Figure 3.3 — User interface creation in QTDesigner

As well as allowing the user to add components to a window canvas and to directly manipulate them, most GUI editors also provide access to all *properties* of the currently selected component through a separate list view. These encompass all aspects of the component, including visual properties such as size and position etc., as well as those properties that do not have a visual representation. This access to 'hidden' properties, such as the binding sources of any linked data, through a separate aspect of the editing application, allows a WYSIWYG view to be used for editing without losing the ability to control certain features of individual components.

Another interesting feature of many GUI designers is the way in which they indicate various layout constraints to the user. Figures 3.2 and 3.3 both show how the positioning of components is dependent upon the relationship with other components and the canvas boundary. These relationships can be managed through various component properties as well as through direct interaction with these 'meta-components'. It is clear that this idea of providing the user with graphical meta-components to manipulate both structure and layout could also be applied to the editing of variable data documents.

3.2 Visualizing Document Variability

In the previous sections a number of different editors have been discussed that are designed to support the creation and modification of all types of documents. Many of these work within a WYSIWYG framework, but in reality the process of editing variable data documents, using the tools described, is not truly WYSIWYG. Variable data frameworks based upon existing editors that use custom placeholder components are only able to display a complete view of the document when processed against an exhaustive set of input data. Therefore the interactive document view offered to users contains unbound variable placeholder components and is, at best, What-You-See-Is-*Part-Of*-What-You-Get. The custom variable data document editors that have been described improve this situation by giving a complete view of a single document instance, but they are unable to adequately show the full variation in the document without also producing a (potentially vast) set of example documents. In this case, we might also consider these editors to be What-You-See-Is-*One-Instance-Of-What-You-Might*-Get.

This is the crux of the problem — a true WYSIWYG editor must provide the user with a complete view of the whole document, through which they can effect any and all changes, but by their very nature, variable data documents have no single instance that can be used in such a situation.

3.2.1 Data Variability vs. Document Logic Variability

To understand how we might solve this problem it is worth analysing the nature of the variability within documents, which can broadly be classified into one of two categories: *data* variability and *document logic* variability.

Data Variability

The most common type of variability encountered is through components within the document whose content is sourced directly from the variable input data. Classic examples, such as the inclusion of a recipient's address in a document, illustrate that a variable component may be purely textual. The bounds of the text component may be changed to accommodate the variable text, but beyond that, and any consequential changes to other components, the *structure* of the document is not affected.

Document Logic Variability

In contrast to the effects caused by the changing of the content of a document component, it is the overall structure of the document (and the reasoning on which such changes are based) that we refer to here. This type of variability occurs when components are conditionally included in the document and/or the layout of such components is dependent upon some value in the variable data.

This type of variability is borne from the programmatic structures within the source document in the form of conditional branch statements (‘`if`’ statements etc.) and, in template matching languages such as XSLT, the implicit condition of whether or not a node exists in the input data that might trigger some given template.

Clearly, in situations involving simple variability, such as copy-hole documents, it is sufficient to deal only with the first of these two types. However, when working with fully variable documents, of the sort shown in the earlier examples, structural variability also has a major influence.

3.2.2 Displaying Variable Documents

Having examined how editing applications tackle the problems associated with different document types (in particular focussing on the issues specific to variable data documents) it is apparent that there is scope for improvement in the WYSIWYG editing of such documents. Two main points need attention: how we display the document to the user and how the user interacts with the given display.

3.2.2.1 Document Models

The display of the document is central to any WYSIWYG editing application — what is seen must be representative of what the document will look like when finally processed. This document view is a direct consequence of the way in which the document is modelled and the way different aspects of the document are accessed through this model. A summary of the two main approaches is given below.

Template-Based

The first option is to present the user with an abstract 'template' view of the source document where all static parts are rendered and where any variable components are rendered as fully as possible given their known property values. This is similar to the approach taken by products such as uDirect and CatBase, but it leaves the problem of satisfactorily displaying the structural variations discussed above. One solution is to initially show blank placeholders to show that there is more than one component that can be included at that point in the document. However, the individual child components of this conditional placeholder would then be accessible for editing by 'opening' the component to display its contents. This model has the advantage that all possible components that may appear in the result document can be accessed and edited through the same document representation. However, in practice, the author would frequently be presented with a document view comprising mainly blank placeholders and partially-rendered components with no content. The problem with this approach, for users with a 'design' background, is that it is too complex and programmatic and does not display the design aspects of the document sufficiently.

Instance-Based

The alternative model for representing a variable data document is to edit the result instances of the document after it has been bound to some input data and processed. This is the approach taken by the editor proposed by Lumley *et al* (see section 3.1.2). The major advantage of this approach is that the document presented to the author is a fully-rendered view of the document, therefore overcoming the main problem with the template-based model described above. It essentially treats the variable data document as a fixed-form one, meaning that in order to edit the full range of components that may be included in the result documents, numerous instance documents must be presented for editing. A further point is that once a change has been made to a particular document instance, it is not necessarily clear to the author what effect this will have on other instances without rendering and viewing them individually. For large, complex documents, this repeated proofing becomes impractical.

Both of these approaches have their benefits and shortcomings, and it is conceivable that aspects of both approaches could be incorporated into a third possibility. For example, a template-based model could be instantiated with a default data set that would result in a 'complete' document instance whilst retaining the abilities of treating the variable components as entities in their own right. The result is that components that could previously be only partially rendered can now be fully rendered in the same manner as an instance from the result document set. However, this still leaves the problems associated with displaying conditional components to the author and allowing them to be edited, as well as showing only a single document instance that will inevitably not show the complete range of variation supported. Taking these points a step further brings us to issues of the way that the author interacts with the document during the authoring process.

3.2.2.2 Interacting with the Document

Given the kind of hybrid model wherein a document is instantiated with a sample input data set to produce a fully renderable result instance, we must consider how the inherent variability of the document is exposed to the author and how they might interact with it accordingly.

This idea of exposing the variability to the author is one that is important, yet it has not necessarily been fully explored with existing editors. When working with a document instance, we must present the user with a view that shows the potential structural variability as well as the variability due to the changing data content. To do this we must show to the user, either directly or indirectly, both the programmatic transform and the input data set. As an example, consider a document with components that are conditionally included in the document as well as other components with varying textual content, such as names and addresses. It is important that the document view indicates both of these facts to the author and, furthermore, aids the author in understanding how these aspects of variability will affect the document from one instance to the next. In the case of the conditional component, the author must be able to edit the component whether or not it is included in the current result instance. Therefore, the editing application must indicate to the author that there is a component within the document that is not currently shown and allow for it to be displayed upon request. One way of achieving this is to allow the author to switch to a different input data set that results in the component being displayed. The situation involving the changing content of a persistent component can be addressed in the same way. If the author could change from one instance to the next, the effects of this change in content could be seen directly in the updated document view.

Although this might seem to be no more than emulating the process of producing a series of result instances, there is an important difference. These changes to the input data set are being instigated by the author on a component-by-component basis. Thus, rather than generating a series of somewhat random instances, the author is able to intuitively generate a specific instance that displays the required combination of input data to illustrate some desired aspect of the document. This can be taken further by guiding the author through the data selection process and providing a series of commonly occuring samples that are at the extremes of the allowable ranges. These samples could be collected from a set of sample instances or even the complete data set that will be used with the final document. For example, the textual content of the address component could be provided as a node in the input data. The pre-processing of a series of separate input data could collect the shortest, longest and most common addresses. These could then be presented to authors when they select the component generated from this variable data. In this way they can see the effects of changes to this component on the rest of the document. In contrast to the approach of cycling through a series of complete input data sets, this approach is able to generate a much larger number of document configurations, as well as selecting potentially more meaningful data.

The way in which information is presented to a document author is an area of research for Human-Computer Interaction (HCI) experts [39] and is beyond the scope of this thesis. Many of the issues relating to displaying the variability available within a document would need careful consideration and investigation in order to provide an optimal solution. Indeed, Terry *et al*[40] examine the complexities involved in designing user interfaces for creative applications and processes such as the one we describe. These considerations will therefore not be discussed further in this thesis, but the simple act of collecting and exposing this data, and the mechanisms required to support an editing process that fully utilizes it will be the subject of later chapters.

In this chapter, we have looked at the issues involved with editing documents in a WYSIWYG manner with a particular focus on the problems associated with editing variable data documents. The goal of effectively authoring such documents, through a WYSIWYG view, requires us to go beyond both the blank-placeholder 'template' and the single result 'instance' models towards a new model that encompasses aspects of both. Furthermore, user interactions with the presented view of the document must be tailored to provide access to all aspects of variabilty within the document, and it must be done in an informative way. In the next chapter, we consider the underlying processing model used to drive the editing process, as well as any implications that a change to a new editing paradigm might have.

Chapter 4:

Variable Data Document Processing

In a variable data workflow the document reprocessing required, as a result of an edit, can be substantial if each output page is fully recomputed. But this complete reprocessing is sometimes unnecessary — the effects of an edit are often localised to a subset of the components within the document. If this localisation also applies to corresponding components in the *source* document, then the changes to the programmatic transformation will also be localised. In principle, this will limit the necessary reprocessing. Therefore, the goal of providing a practical interactive editor for variable data documents requires that any reprocessing resulting from an output-based edit should be limited only to those source-level components that were affected. The clear implication here is that we can enforce a strong link between source-level components and their final appearance in the output document.

Using a representative XSLT-based document framework, this chapter examines in detail the relationship between result document components and the parts of the transform repsonsible for generating them.

4.1 Simple Document Workflow

As a foundation for the discussion that follows, we first introduce an example document workflow. The use of XML and XSLT in VDP systems has been discussed previously and we continue to concentrate on these technologies. They provide a workflow where an XML input data file is transformed by an XSLT script to directly produce the result document. This is a simplified version of the approach taken by systems such as DDF, which are more complex and produce intermediate results that are processed in several stages. The purpose of using a simple one-step transformation is to remove some of the complexities that are both unnecessary and potentially confusing. However, it should be noted that this does not preclude the techniques to be described from being extended to multi-stage processing workflows.

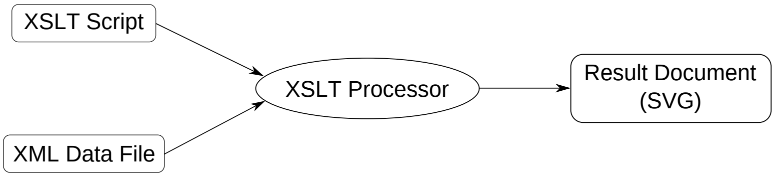


Figure 4.1 — Simple one-step document transformation

Figure 4.1 shows the process by which the input data will be transformed into a final-form result document. The variable data is provided in the form of an XML file, with the actual document structure being produced by the XSLT script. When this script is processed by an XSLT processor the variable data is bound into the result document — in this example workflow an SVG document. To produce a series of different result document instances, the XSLT script must simply be repeatedly executed with different input data used for each execution.

4.1.1 Editing the Document

Once the resulting SVG document has been produced, any changes to it must be made by amending either the XSLT transform, or the XML data, and then reprocessing to produce a new result. These edits can be made directly to the data and/or the XSLT code, but in the case of a WYSIWYG editor such edits would be made indirectly, through interactions of the user with the resulting SVG document. Methods of effecting the required changes to the input files, via edits to a final-form result document, have been proposed [30], but for the purposes of this discussion we are not so much interested in how these changes are made, but simply in the fact that they can, and do, occur.

Edits to the Transform vs. Edits to the Data

It is worth noting that changes made to the transform are different in nature from those made to the input data. Modifications made to the XSLT transform typically have the effect of changing the *structure* of the document and the properties of the components within it, whereas modifications of the data change just the *content* of these components. In an interactive editing environment, edits to the XSLT transform are a result of standard authoring operations on the document (adding/moving/resizing components etc.) whereas edits to the variable data are the result of requesting an alternative document instance as described in the previous chapter.

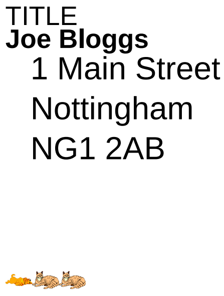
As an example of a typical edit, consider the XSLT code fragment given in figure 4.3, which would produce output similar to that shown in figure 4.4 when provided with the variable data shown in figure 4.2.

```
<person forename="Joe" surname="Bloggs">
   <address>
      1 Main Street,
      Nottingham
      NG1 2AB
   </address>
   <pets>
      <dog name="Rover"/>
      <cat name="Fluffy"/>
      <cat name="Top"/>
   </pets>
</person>
```

Figure 4.2 — Sample variable data

```
<xsl:stylesheet version="2.0">
   <xsl:template match="person">
      <svg:svg>
         <svg:text x="0" y="0">TITLE</svg:text>
         <svg:text x="0" y="10" font-weight="bold">
            <xsl:value-of select="concat(@surname, ', ', @forename)"/>
         </svg:text>
         <xsl:apply-templates/>
      </svg:svg>
   </xsl:template>

   <xsl:template match="address">
      <svg:text x="10" y="20" font-family="Helvetica" font-size="12pt">
         <xsl:value-of select="."/>
      </svg:text>
   </xsl:template>

   <xsl:template match="pets">
      <xsl:apply-templates/>
   </xsl:template>

   <xsl:template match="dog">
      <svg:image xlink:href="dog1.jpg" x="{10 + (count(preceding-
         sibling::*)*10)}" y="100" width="10" height="10"/>
   </xsl:template>

   <xsl:template match="cat">
      <svg:image xlink:href="cat1.jpg" x="{10 + (count(preceding-
         sibling::*)*10)}" y="100" width="10" height="10"/>
   </xsl:template>
</xsl:stylesheet>
```

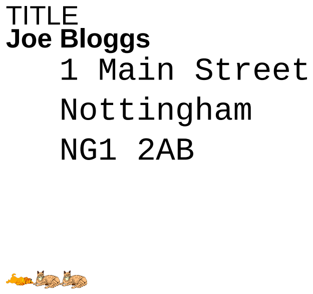
Figure 4.3 — Example XSLT transformation script



TITLE
Joe Bloggs
1 Main Street
Nottingham
NG1 2AB

Figure 4.4 — Example result document

If the user wanted to change the font in which the address was displayed, the value of the `font-family` attribute must be changed by modifying the instruction(s) responsible for generating that attribute at the the relevant point in the XSLT transform. Once this change has been made, the input data file and the XSLT script are completely reprocessed to generate the new result SVG document shown in figure 4.5.



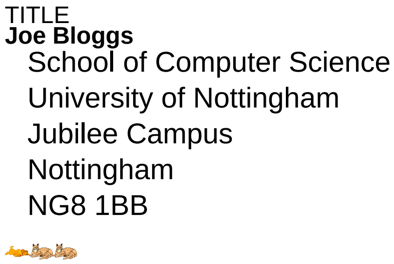
TITLE
Joe Bloggs
```
1 Main Street
Nottingham
NG1 2AB
```

Figure 4.5 — Modified result document

As an alternative *type* of edit, consider the process involved when the user wishes to see the effect on the document of some new piece of variable data. Continuing with the previous example, the user may wish to see the effect on the document when the value of a piece of variable data changes, e.g. an address. The address shown in figure 4.2 is quite short (only three lines) and it is conceivable that the addresses of other people may be much longer. Although the images in the example are *not* positioned dependent upon the position and size of the address, this is something that might occur frequently in real-world documents. In this situation the author might wish to see where the images would be positioned when a longer address is provided. Figure 4.6 shows a modified data file and figure 4.7 shows the revised document produced as a result of the change.

```
<person forename="Joe" surname="Bloggs">
   <address>
      School of Computer Science,
      University of Nottingham,
      Jubilee Campus,
      NOTTINGHAM
      NG8 1BB
   </address>
   <pets>
      <dog name="Rover"/>
      <cat name="Fluffy"/>
      <cat name="Top"/>
   </pets>
</person>
```

Figure 4.6 — Sample data with longer address field



TITLE
Joe Bloggs
School of Computer Science
University of Nottingham
Jubilee Campus
Nottingham
NG8 1BB

Figure 4.7 — Result document showing longer address

4.2 The Need for Partial Re-Evaluation

The editing process described above suffers from one serious drawback when we consider applying it to the situation that exists in an interactive editing environment. For the type of trivial documents we have used in the examples so far, the cost of reprocessing is relatively inconsequential. When considering realistic documents, which may contain hundreds of complex pages with numerous components on each page, the cost of reprocessing the entire document becomes prohibitive. This problem is not restricted simply to the document workflow presented here, but is a more general one encountered by any variable data document editing procedure that relies upon the updating and reprocessing of an interactive document instance. Indeed the variable-data document editors discussed in chapter 3 come up against this same problem, whether when regenerating the document instance as described here, or simply generating a set of example results for proofing.

To support interactive editing of such computationally expensive documents, we must follow one of two strategies: either find a way to speed up the tools used to perform the processing, or simply perform less processing.

Faster Processing

There are several ways to achieve an increase in processing speed. At the lowest level we can provide faster hardware to improve performance, however this simply raises the point at which a document becomes too complex to realistically reprocess. The current trend of an increasing number of processing cores, coupled with potential multithreading optimisations to the XSLT processing software provides another opportunity for potentially large performance gains given the functional nature of the language. Furthermore, the stylesheet itself could be statically optimised to speed up its execution [41, 42], but again these improvements can easily be nullified by simply providing a larger or more complex document. Indeed, *any* technique for speeding up processing will ultimately be overcome by attempting to process a suitably large and/or complex document.

Reducing Processor Workload

To overcome the performance problem, the cost of reprocessing must be decoupled from the size of the document by reducing the amount of the document that is reprocessed as the result of an editing operation. In the example shown in the previous section, the only altered part of the XSLT transform was that responsible for producing the address. The remainder of the transform was left unmodified and so the parts of the result document that were produced by the unmodified code are identical to those that were originally generated. This can be seen in the example of figure 4.7 as the images are unaffected by the change to the XSLT transform. Avoiding redundant reprocessing of unaffected parts of the document is central to a processing model where the computational expense is constrained by the extent and complexity of the edit operation, rather than the size and complexity of the document itself.

As discussed in chapter 2, the idea of partial XSLT evaluation is an obvious extension when processing a language such as XSLT. Indeed, Villard *et al*, have proposed methods for incrementally processing XSLT scripts [43] through analysis of the stylesheet template patterns to ascertain the execution flow through which source elements might be processed. However their techniques and goals differ from those discussed in the following sections.

Supporting Selective Re-processing

We now consider the tree-based representation, shown in figure 4.8, of the example XSLT code provided earlier in figure 4.3. In order to reprocess only the address component we must re-execute the XSLT instruction(s) contained within the subtree of the node responsible for producing that component while ignoring the rest of the transform. In the case of this example, the instructions to be re-executed are highlighted in red.

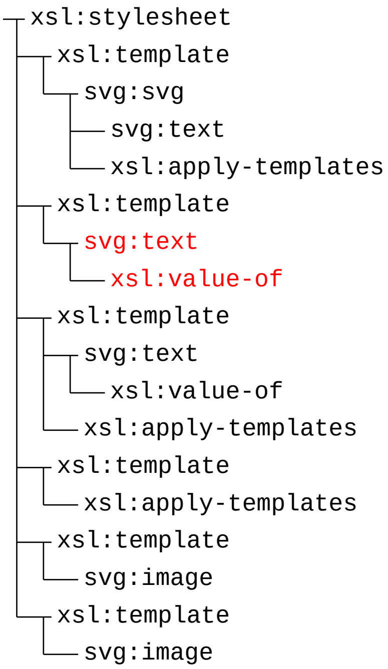


Figure 4.8 — Highlighted instructions to be re-executed

In this simple example, re-evaluating the modified part of the transform is a trivial exercise, but in more complex scripts where the modified code may make reference to previously declared variables, parameters, input data calculations etc. we must also be able to provide their original values. The method we propose for providing this required information is based on recording the state of the processor throughout the original processing stage and restoring it, as required, during reprocessing. The following chapters look in detail at the processes involved, and describe how such techniques have been applied via modifications to existing tools.

Chapter 5:

Implementation Options

The need to save and restore the state of execution during XSLT processing requires changes to be made to the processing pipeline itself. The constituent elements of the workflow can be considered to be either inputs (data files, stylesheets, intermediate results etc.) or processing tools (parsers, XSLT processors etc.). Therefore, there are two avenues to be explored when implementing the storage and restoration of processing state. Firstly, preprocessing of the inputs may be needed in order to produce the required state information during execution. Secondly, one may need to modify the processing tools to perform the necessary extra actions as they execute the input stylesheets.

Both of these approaches are considered in the following sections and the advantages and disadvantages of each are examined. In later chapters, the topics will be presented in terms of one, or both, of these different approaches, giving insight into the suitability of each.

5.1 Stylesheet Modification

Since the specifics of the document execution, in our example workflow, are based upon the processing performed by an XSLT stylesheet we need to modify that stylesheet so as to produce the required state information along with the original output. A detailed discussion of how this was achieved will be given in section 6.1.

However, a few problems arise when one tries to implement the proposed reprocessing scheme solely through the modification of the input stylesheet(s). Firstly, there is the issue of separating the original output of the stylesheet from the extra state output that is generated. At first glance, namespaces might appear to be a valid solution to this problem of 'hiding' extra information. The extra state output elements/attributes could simply be placed in a specific namespace so as to separate them from the origingal content. However there are some subtle problems with this approach — namely that these extra elements/attributes can potentially affect the subsequent execution of the stylesheet. Secondly, it is not immediately obvious how the problem of reinitializing the processor to a given state, in order to perform a partial re-evaluation, can be handled if there is access only to the input stylesheets and data. These apparent problems, along with others, are discussed in more detail as they are encountered in later chapters.

5.2 Using a Specialist XSLT Processor

XSLT is sufficiently powerful and expressive that it is entirely possible to write an XSLT program that consumes another XSLT script and executes the instructions contained within it as per the langauge specification. Clearly, a 'real' processor is required to execute the underlying 'processor' script, but any extra functionality of the sort we require, can be implemented within this extra layer of abstraction. We shall see that by working at the level 'below' the document stylesheet, we can overcome some of the problems encountered by the alternative solution of modifying the stylesheet itself.

Because such a processor is in control of the XSLT statements being executed, it is possible to add extra instructions that will allow for the current interpreter state to be recorded internally, without the normal output of the stylesheet being affected. Furthermore, the templates used to interpret the XSLT statements can be written to facilitate easy instantiation with a stored state. In this way they can be re-executed when necessary.

5.3 Modifying an Existing XSLT Processor

There are many advantages to recording the execution state from within the processor itself. The approach discussed in the previous section offers this possibility, but its performance would be restricted by the fact that we have two extra layers of XSLT running on top of the underlying XSLT processor.

Indeed, given that there is still an underlying XSLT processor, the next logical step is to remove the need for the XSLT layer built on top and to modify the processor itself. In that way we gain the following advantages:

- Full XSLT specification implementation is already available
- Execution optimizations are performed by the processor at no cost
- Speed increases should ensue, due to the removal of the extra XSLT processor layer
- Direct access to the processor's internal data structures help us to obtain, and to maintain, the required state information

This final point is arguably the most important since it allows us to avoid many of the problems of outputting state information without affecting the original execution of the XSLT script. Since we have access to the processor's data structures, and execution system, it is possible to retain copies of the required values and data structures and, more importantly, to store them within the processor without changing the output that is generated.

Although these advantages are welcome, the price to be paid is that the process of making the required modifications is potentially much more complex than creating a custom processor from scratch. However, the benefits of working with an existing processor outweigh this extra complexity and chapters 6 – 8 detail the modifications that have been made to an existing XSLT processor to support partial re-evaluation.

5.3.1 Existing XSLT Processors

XSLT processing is an integral part of many XML-based workflows. As a result, several XSLT processors have been developed. Obviously, some of these are proprietary software, but, since we must make significant modifications that require access to the source code, we are restricted to working with predominantly open-source projects. Two of the most popular such processors are Xalan [44] and Saxon [45] but the choice was Saxon for several reasons that now follow.

Firstly, Saxon offers support for version 2.0 of the XSLT language, and XPath, whereas Xalan, and many others, only fully support the outdated version 1.0. This is important since variable data documents often make use of the features and facilities introduced in the updated language specification. Saxon also employs many internal optimisations [46] during processing that provide good memory usage and execution time performance. Finally, Saxon is considered to be one of the best XSLT processors available [47] and is therefore widely used. A contributing factor to its popularity is that of confidence in the main developer, Michael Kay, who was editor of the XSLT 2.0 language specification.

As a primer for later discussions, including the changes made to Saxon, we must first explain its internal design. Michael Kay wrote an architectural overview [48] in 2005 that readers may find of interest. However it should be noted that this review refers to an older version of Saxon and that there have been substantial changes made in some areas of its architecture and implementation.

5.3.2 Saxon Architecture

Saxon is a complex piece of software that offers support for XQuery as well as the processing of XSLT stylesheets. As a result, its architecture sometimes appears more cumbersome than would be necessary solely for processing XSLT. Therefore, aspects of the design that are exclusive to the support for XQuery are omitted from the following discussions.

5.3.2.1 High Level Design

Before discussing the detailed aspects of how Saxon loads and processes stylesheets, we present a general overview of its operation. Figure 5.1 shows the major stages of processing and how they relate to one another.

The first task performed by Saxon is to load both the XML input file and the XSLT stylesheet that will be the subjects of the transformation. When Saxon is run as a standalone tool these are loaded from a file using standard parsing methodologies, however, as we shall see later, interfaces are included to support the loading of existing DOMs without the need for serialization and immediate reparsing.

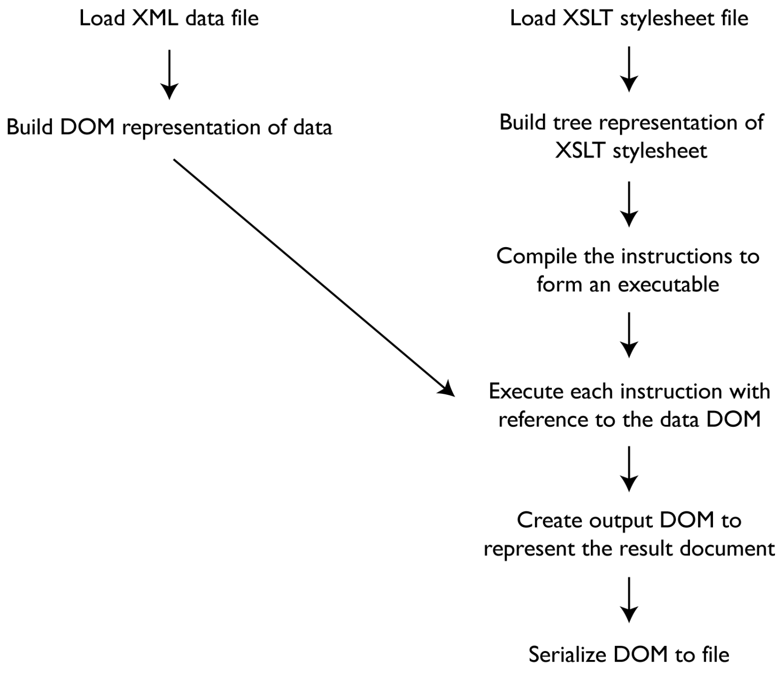

Load XML data file
Build DOM representation of data
Load XSLT stylesheet file
Build tree representation of XSLT stylesheet
Compile the instructions to form an executable
Execute each instruction with reference to the data DOM
Create output DOM to represent the result document
Serialize DOM to file

Figure 5.1 — Saxon Architectural Overview

The input XML and XSLT stylesheet documents may be loaded from disk, or from an alternative source such as an existing DOM or a SAX-based source, but in either case they are used to generate events that are handled by the appropriate builder classes to construct the data structures used within Saxon. A DOM-like structure is created to represent the input XML file, and a specialized builder constructs an internal representation of the XSLT stylesheet.

Once the tree representation of the stylesheet has been constructed, it is then compiled into an executable form. This executable contains mechanisms for selecting templates in accordance with the XSLT language specification together with executable expressions for each of the supported XSLT elements. The complete executable is then executed with the previously constructed input XML document as its data source. Those instructions that produce output content then send events to the appropriate result document builder. Depending on the configuration of Saxon, these events can be used to create a result DOM that is serialized to a file or passed to another linked tool. As we shall see in later chapters, this ability to change the builder program, responsible for creating the result DOM, is of great convenience when using Saxon as a processing engine within a document editor.

The subsequent sections discuss the details of the various stages of stylesheet and data processing that are performed during execution. To illustrate these discussions, and to give a coherent overview of the process, the following example documents are used.

```
<person forename="Joe" surname="Bloggs">
   <address>
      <line>1 Main Street</line>
      <line>Localville</line>
      <line>Mytown</line>
   </address>
   <store name="Nottingham" map_url="notts.jpg"/>
   <offers>
      <offer image="product1.jpg">
         <title>20% off selected wines</title>
         <description>Choose from our extensive selection of quality wines
             and enjoy savings of up to 20%!</description>
      </offer>
   </offers>
</person>
```

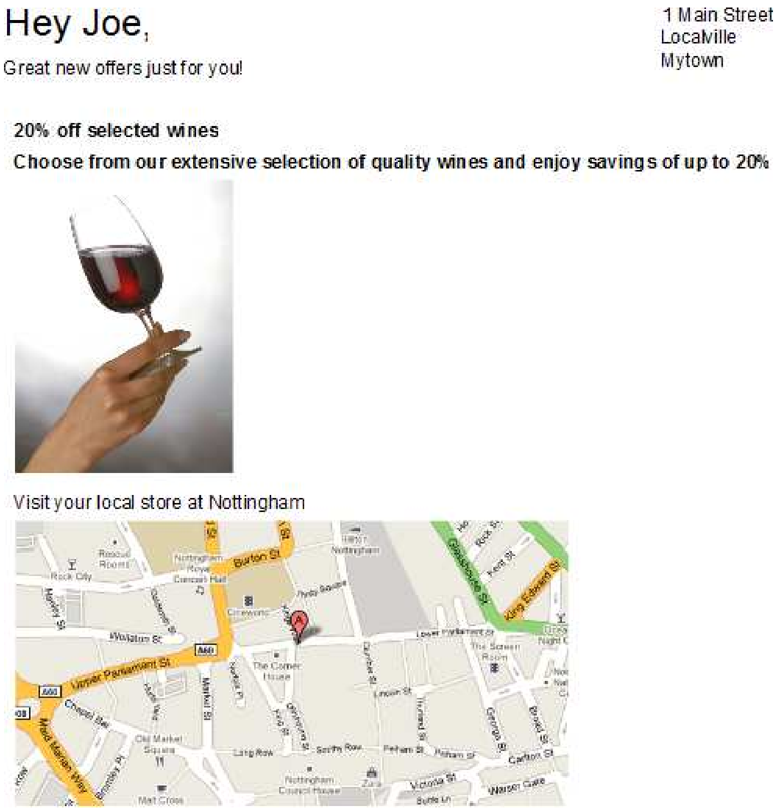
Figure 5.2 — Example input XML document

```
<xsl:template match="/">
   <svg:svg width="21cm" height="29.7cm">
      <svg:text x="10" y="10" font-family="Helvetica" font-size="24">
         <xsl:value-of select="concat('Hey ', @forename, ',')"/>
      </svg:text>
      <xsl:apply-templates/>
   </svg:svg>
</xsl:template>

<xsl:template match="address">
   <svg:text x="200" y="10">
      <xsl:apply-templates/>
   </svg:text>
</xsl:template>

<xsl:template match="line">
```

```
   <xsl:value-of select="."/>
</xsl:template>

<xsl:template match="offers">
   <svg:text x="10" y="30">Great new offers just for you!</svg:text>
   <xsl:apply-templates/>
</xsl:template>

<xsl:template match="offer">
   <svg:text x="10 + (100 * count(preceding-sibling::offer))" y="50" font-
      style="bold">
      <xsl:value-of select="title"/>
   </svg:text>
   <svg:text x="10 + (100 * count(preceding-sibling::offer))" y="70" font-
      style="bold">
      <xsl:value-of select="description"/>
   </svg:text>
   <svg:image x="10" y="100" xlink:href="{@image}"/>
</xsl:template>

<xsl:template match="store">
   <svg:text x="10" y="200">
      Visit your local store at
      <xsl:value-of select="@name"/>
   </svg:text>
   <svg:image x="10" y="220" xlink:href="{@map_url}"/>
</xsl:template>
```

Figure 5.3 — Example XSLT stylesheet

The XSLT stylesheet shown in figure 5.3 produces a simple marketing flyer based upon the contents of the input XML file shown in figure 5.2. Although the output document produced (figure 5.4) is simple, the premise of customized advertising material is frequently encountered in variable data publishing.



Hey Joe,

Great new offers just for you!

1 Main Street
Localville
Mytown

20% off selected wines

Choose from our extensive selection of quality wines and enjoy savings of up to 20%

Visit your local store at Nottingham

Figure 5.4 — SVG document produced from example inputs

We shall now look at each of processing stages that occur during the transformation of these input documents into the output shown in figure 5.4.

5.3.2.2 Input XML File

As previously mentioned, Saxon builds an internal representation of the input XML document to be used during execution of the stylesheet. This data structure is built by a subclass of `Builder`, a class that receives the events generated by the parsing of the input XML document. This document can be built from a file, or can be an existing DOM or SAX source, but whatever the source of the document, the same events are sent to the `Builder` instance. These events are similar to the events generated by a standard SAX parser and reflect the structure of the document that is being processed. For example, the builder is alerted to the start and end of element definitions, text nodes and other related events.

Saxon provides support for different `Builder` implementations, each of which is responsible for building a document instance that conforms to the DOM-like interface used internally. There are two implementations supported by default (`TreeBuilder` and `TinyTreeBuilder`), but, as discussed later, a new `Builder` implementation is required to build documents with specific properties that better support the proposed editing model. The basic `TreeBuilder` constructs a document that is implemented using objects to represent nodes in the tree, with references stored between them as is typical for implementations of DOM-like data structures. In contrast, the `TinyTreeBuilder` creates a document that makes heavy use of primitive types and arrays to reduce the memory footprint of the document, thus providing a general increase in efficiency during execution. The default configuration is for Saxon to use the `TinyTreeBuilder` because of its greater efficiency, but since all implementations expose the same interface to the processor, the behaviour and functionality is identical whichever `Builder` is used.

Figure 5.5 shows a representation of the tree structure used to model the example input XML document when built using the standard `TreeBuilder` implementation. The `ElementImpl` objects that represent the elements within the document have been labelled with the name of the element for clarity.

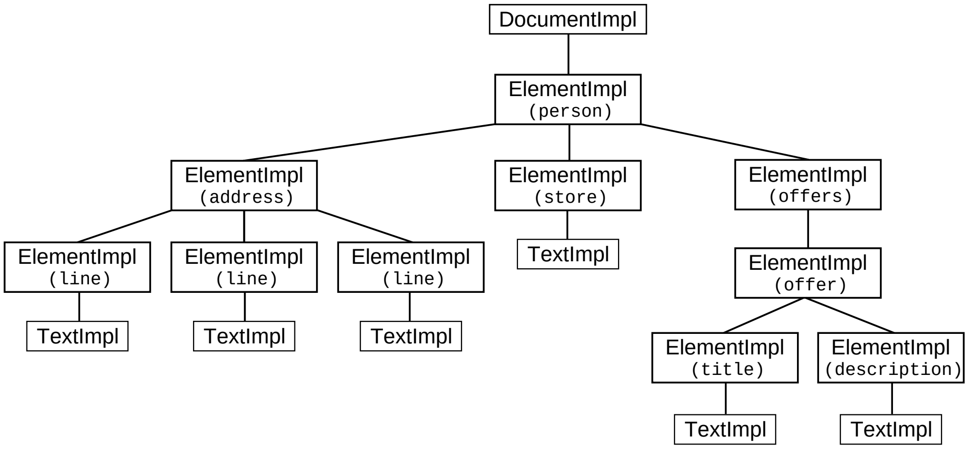


Figure 5.5 — Input XML document object structure

5.3.2.3 Parsed XSLT Representation

A similar procedure to that used for parsing the input XML data file is followed when parsing the XSLT stylesheet and, consequently, building the internal data structure used to represent it. Again, the `TreeBuilder` class is used to handle the parsing events that are generated, but instead of building the document using a generic implementation, it creates a representation that uses specific classes to represent each of the various XSLT instructions. This change in functionality is achieved by replacing the default `NodeFactory`, used by `TreeBuilder`, with a specialist one (`StyleElementFactory`).

The result of this building process is a tree structure composed of various subclasses of `StyleElement`, each responsible for providing the functionality associated with the equivalent XSLT instruction. Elements in the stylesheet that are not in the XSLT namespace, and hence are to be output to the result document, are represented by `LiteralResultElement` objects. These objects also inherit from `StyleElement`, but instead of bearing responsibility for one of the various instructions supported in the

XSLT language specification, each instance contains the details of the content to be output.

Returning to the example stylesheet introduced earlier in this chapter, figure 5.6 shows the various objects and their relations used to represent part of it. Due to size contraints, only the object representation of the first template is shown. A complete diagram is included in Appendix A. We can see that, for example, the `<xsl:value-of>` elements used to select the textual content of nodes in the input tree are represented by corresponding `XSLValueOf` objects, whereas the non-XSL elements to be copied to the output document are represented as `LiteralResultElements`.

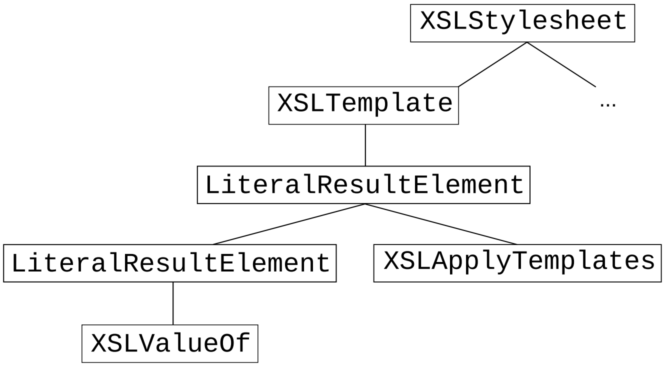


Figure 5.6 — `StyleElement` object tree represention of example template

5.3.2.4 Stylesheet Compilation

Having produced an internal representation of the stylesheet, it is then prepared for execution by compiling each `StyleElement` within the tree to create an executable form of the stylesheet. The functionality required to construct and evaluate this executable form is encapsulated within the `PreparedStylesheet` class. Methods for the construction of the XSLT tree representation, as well as for initiating the compilation process, are called from other external classes, with reference to the XSLT tree representation. The final executable is retained by the `PreparedStylesheet` instance. It is this executable that subsequently interacts with the input data tree and produces the corresponding output of the stylesheet. `XSLTemplate` nodes representing the `<xsl:template>` elements add pattern rules to an instance of a `RuleManager` (see next section) and contain a hierarchical structure of `Instruction` and `Expression` objects that are produced by the child elements in the stylesheet. These instructions and expressions are then evaluated in accordance with the XSLT and XPath language specifications. Any calls that contribute to producing the result document tree are sent to the appropriate `Receiver` object. The details of the execution of the stylesheet and the resulting output document production are discussed in the following sections.

5.3.2.4.1 The Executable

Expressions and Instructions

The base type for all compiled objects is `Expression`, with all XSLT instructions inheriting from `Instruction` (which is itself a subclass of `Expression`). XPath expressions are also compiled into a sequence of specialised `Expression` objects, the particular types depending upon the XPath expression to be compiled. With the exception of a few specific instructions, the hierarchical nature of the original stylesheet tree is preserved by maintaining expressions as children of one another. The execution methods of each instruction are therefore responsible for instigating the execution of subsequent instructions.

Templates and the `RuleManager`

One of the exceptions to hierarchical execution is that of templates. When an `XSLTemplate` object (a subclass of `StyleElement` used to represent `<xsl:template>` elements) is compiled, a corresponding `Template` object is produced. However instead of adding it directly to another object representing the stylesheet itself, it is, instead, added to a `RuleManager` instance, which is maintainted by the `Executable`.

This `RuleManager` is responsible for maintaining a list of the available templates[1] and, subsequently, providing the correct one whenever it is queried throughout the processing of the stylesheet. The 'correctness' of the template returned depends upon a number of factors, each specified by the XSLT language specification and handled by the `RuleManager`. In order for the correct template to be returned, it must meet the following requirements:

- The `Pattern`[2] against which a `Template` is stored should match the node currently being processed
- The mode in which the processor is to process the node is compatible with the `Mode` object stored alongside the `Template` in question
- The priority of the `Template` is consistent with that specified
- The `Template` has the highest precedence of all those that match the other requirements listed above

[1] In truth, the `RuleManager` maintains a series of `Mode` objects, each of which maintains a list of `Templates`, but the exposed interface hides this

[2] A `Pattern` object is built from, and thus represents, the `match` attribute specified on an `<xsl:template>` element

Therefore, these pieces of information are provided to the RuleManager when a Template is added, in order that such a query can be performed during the later operation of the Executable.

Figure 5.7 shows the Expression hierarchy built to represent the root <xsl:template> of the example stylesheet we presented earlier in the chapter. The FixedElement objects have been annotated with the name of the element that they produce, however their associated attributes have been omitted to keep the figure concise. A Block object is also included to allow for an instruction to reference multiple child instructions. Clearly, similar hierarchies are built for the other <xsl:template> elements in the stylesheet, but for clarity, only one is presented in the figure. A complete diagram of the full object hierarchy can be found in Appendix A.

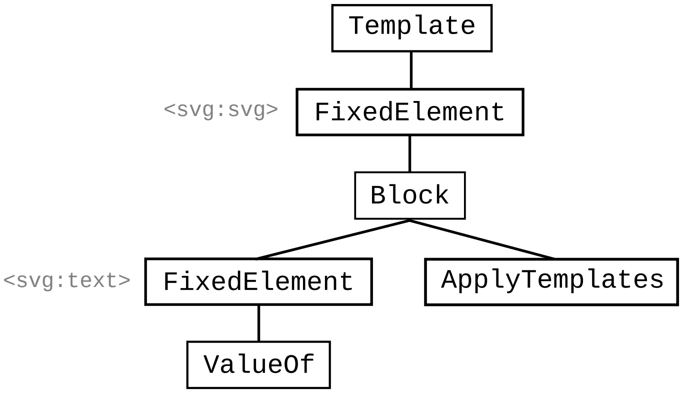


Figure 5.7 — Example Template Expression Heirarchy

5.3.3 Compiled Stylesheet Execution

Having discussed the way in which the executable is created, we now look at the process involved in executing it in order to produce the output document.

Initial Template Selection

Depending on whether or not a specific `<xsl:template>` has been indicted as the starting template, the stylesheet will begin execution by either directly processing the instructions in this template, or by processing the instructions contained within the `Template` returned by querying the `Executable`'s `RuleManager` for the most appropriate instance. Each instruction encountered is then executed through calls to either the `process()` or `processLeavingTail()` methods, depending upon the nature of the instruction in question, with child `Instruction`s and `Expression`s being processed in due course.

This execution process initially appears quite simple, but we have neglected the fact that to query the `RuleManager` it must be provided with information pertaining to the current processing state (current input node, mode, etc.) so that it can return the correct `Template` instance. In fact, this processing *context* is important for all `Instruction`s and `Expression`s that make reference to the input XML document, or to any context-sensitive data such as variables or parameters. For this reason, an object representing the current context in which `Instruction`s and `Expression`s should be processed is maintained by the processor and passed among calls to the `process()` and `processLeavingTail()` methods previously discussed. This context object, or a derivative of it, is maintained with the current details regarding the processor's state as the stylesheet is executed.

`XPathContext`

The context object described above is either an instance of `XPathContextMajor` or `XPathContextMinor`, both of which are subclasses of `XPathContext`. The

XPathContextMajor class inherits from the XPathContextMinor class and adds extra functionality to allow further changes to the dynamic context of the processor. Such a hierarchy of context objects allows for more efficient execution of the stylesheet because frequently updated values can be held within more lightweight XPathContextMinor objects.

All aspects of the dynamic context are maintained within these XPathContext objects and the values can be updated and queried by the Instruction or Expression being executed when necessary. Therefore if, for example, an instruction causes the current context node in the input XML document to change, this will be reflected in the XPathContext object.

Generating Output

One important part of the original stylesheet that has not been discussed so far is that of elements (and their associated Instructions) that produce output. Along with XSLCopy and XSLCopyOf instructions, which represent xsl:copy and xsl:copy-of nodes respectively, the most common ways to produce elements in the result document are through literal result elements and <xsl:element> nodes. As shown in previous sections, an element in the original stylesheet that is not in the xsl namespace will be represented by a LiteralResultElement in the stylesheet object tree and is subsequently compiled into a FixedElement instruction. All <xsl:element> nodes in the stylesheet are represented through XSLElement StyleElements and are also typically compiled to FixedElement instructions[3].

[3] XSLElement objects can also be compiled to ComputedElement instructions in situations where the element name and/or the namespace are not known at compile time and must be computed at runtime

All of the instruction classes discussed above inherit from a common `ElementCreator` superclass that encompasses all instructions responsible for producing elements in the result document. When executed, all `ElementCreator`s query the processor's configuration for the correct object to send the relevant element creation events to and subsequently sends all the required information through appropriate method calls.

The `Receiver` object to which the element creation events are sent is typically responsible for creating a DOM and then serializing this to a file, but this is not always the case. Saxon provides other `Receiver` implementations that can be used as SAX sources to other transformations, thus allowing stylesheets to be chained together with the output of one being used as the input to the next. Although the example workflows that are presented here are simple single step transformations, there is no reason that a series of stylesheets could not be chained together to support a multi-step processing model like that used by some variable data document frameworks such as DDF. Other `Receiver` implementations are also available, including support for custom implementations, so the results produced by the execution of the stylesheet can be used in any way desired. Later chapters will show that this facility is extremely useful when incorporating a Saxon-based processing framework into a WYSIWYG editing application.

Continuing with the example stylesheet introduced earlier, figure 5.8 shows how the `<svg:text>` element in the first template is represented at various stages of processing as well as the series of method calls invoked to produce the corresponding `<svg:text>` node in the result document.

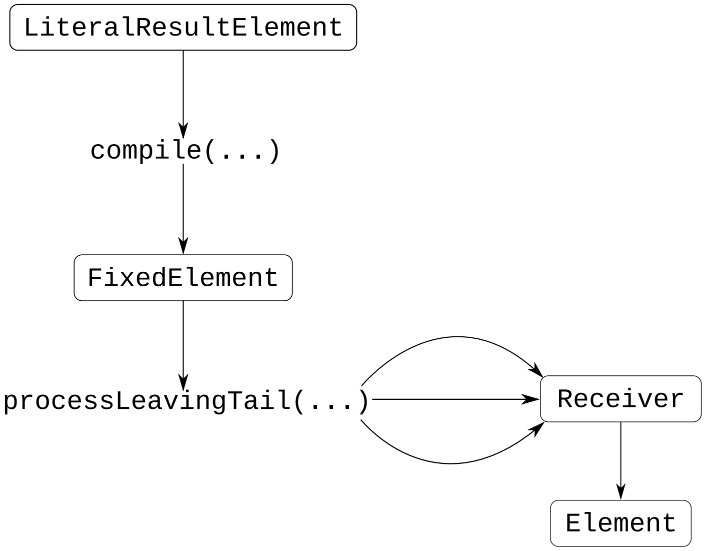


Figure 5.8 — Example output element production

Having now considered different approaches to augmenting the processing pipeline, and having subsequently looked at the implementation details of our chosen XSLT processor, we now explore how the goal of partial document re-evaluation can be realised. The following chapters discuss the different stages of partial re-evaluation and how they were achieved by following one, or another, of the implementation options proposed in this chapter.

Chapter 6:

Storing State and Component History

In this chapter we discuss a support mechanism for partially re-evaluating an XSLT transform to generate a result document. This partial re-evaluation requires the processing state of the XSLT processor to be stored during execution so that we can later restore it for reprocessing a selected part of the transform. We detail the information required for this reprocessing and then explore the ways in which we might collect and store such information, either by modifying the underlying XSLT processor or through modifications to the XSLT transform itself.

State Constituents

At any point during the processing of an XSLT script, there are several things that, together, constitute the state of the processor[1]:

- The current mode

[1] To accompany this discussion of XSLT states the reader is encouraged to refer to the overview of XSLT and XPath given in chapter 2

- The current context node
- The current position
- The values of any variables that are currently in scope
- The values of any parameters passed to the current template/function

To repeat the processing from a given point in the transform, the values of all of these items must be recorded and later restored together with a reference to the current point in the transform.

A good analogy to this process is that of CPU context switching where the execution of one process is suspended to allow another to execute. When a process is suspended, the state of the CPU (all register values, flags etc.) is stored and the state associated with the new process is loaded, thereby allowing it to execute. At a later time, the state of the original process is restored and it continues executing from the point at which it was suspended.

Calculating values

When processing an XSLT script, there are two ways to collect the required information: either modify the transform to calculate the state information and output it alongside the actual output, or output the required information from within the processor as the transform is processed.

We consider both these approaches, including their advantages and shortcomings, in the following sections.

6.1 XSLT Stylesheet Modifications

An obvious method of generating the required state information is by modifying the processing XSLT script. This is a similar approach to that used to track the transformation of document components in related work [49]. The modifications are performed through a pre-processing of the XSLT script that adds extra code to the transform. Since XSLT is XML based, this pre-processing can be done using another XSLT script. When executed, the modified XSLT script produces the same output as the original, but the added code generates extra output detailing the current state of the processor. Therefore, the resulting document is the same as that produced by the original script, but interleaved with extra state information. Figure 6.1 shows the complete two-stage processing scheme.

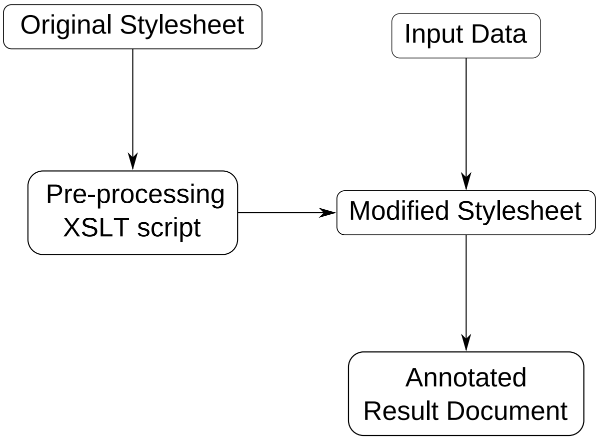


Figure 6.1 — Modified XSLT transform processing stages

Among the numerous instructions within the original transform, there are those responsible for actually generating the output elements. These instructions include:

- `<xsl:element>`
- `<xsl:copy>`
- `<xsl:copy-of>`
- Literal result elements — e.g. `<svg:rect>`

The XSLT script used to modify the original transformation script matches all such elements and inserts a section of code at the relevant points so that, when the modified version of the transform is executed, the desired state information is produced. Figure 6.2 shows an example section of the transform before and after code has been added. The various pieces of newly-added code are explained in the following sections, which also discuss how each piece of information is calculated, and stored, as part of the state of the processor.

```
<!-- Before -->
<xsl:template match="foo">
   <svg:rect width="10" height="10" x="10" y="10" fill="red"/>
</xsl:template>

<!-- After -->
<xsl:template match="foo">
   <svg:rect width="10" height="10" x="10" y="10" fill="red"/>
   <state>
      <!-- various state information nodes here -->
   </state>
</xsl:template>
```

Figure 6.2 — Adding state producing code to the transform

Eager vs. Delayed Calculation

The code fragments to be added to the transform for each piece of state information fall broadly into two categories: those that contain information obtained through calculations performed by the modifying script (eager calculation) and those that contain further code to dynamically generate the relevant values when the modified transform is executed (delayed calculation). However, as we shall see in the following sections, variables and parameters must be processed using both of these methods.

Some of the required information can be eagerly calculated by simply analysing the XSLT transform before it is ever executed. Values such as the path of the XSLT instruction and the names of the (non-tunnelled) parameter/variables that are in scope at the corresponding point in the code can be calculated and added to the transform as literal strings. Other values, such as the actual contents of the variables/

parameters and the current context node, can only be ascertained during execution of the transform and so the calculation of these must be delayed until the transform is processed.

The way in which each piece of state information is obtained is now described.

6.1.1 XSLT Instruction Path

The path to the currently executing instruction is stored so that we have a point of reference to which the rest of the state information relates. This value is necessarily calculated during the script modification process and the literal string value is added to the transform, so that it can be copied onto the output elements when the transform is executed. For efficiency purposes, this literal string value is a truncated version of the complete path such that the relevant node at each level in the tree is simply represented by an integer specifying its position. Therefore, the highlighted node shown in figure 6.3 is represented as `/1/3/2`.

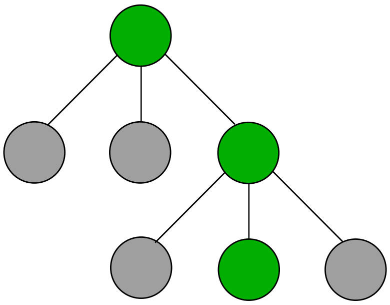


Figure 6.3 — Example of encoding XSLT instruction path

Since we are calculating the appropriate value during the modification of the existing transform, the process by which we create the literal string is straightforward. In this instance, the original transform is the input document to our modifying transform. Therefore, the current context node is the instruction node in the original transform to which we are adding the state producing code. We can simply build a string by recursively querying the parent node (until we reach the document root node) of the current node's position within its children. This process is performed by the user defined `buildInstructionPath` function shown in figure 6.4.

```
<xsl:function name="f:buildInstructionPath">
   <xsl:param name="node"/>
   <xsl:choose>
      <xsl:when test="$node/parent::*">
         <xsl:value-of select="concat(f:buildInstructionPath($node/
            parent::*), '/', count($node/preceding-sibling::*) + 1)"/>
      </xsl:when>
      <xsl:otherwise>
         <xsl:text>/1</xsl:text>
      </xsl:otherwise>
   </xsl:choose>
</xsl:function>
```

Figure 6.4 — Anciallary function for building stylesheet instruction path

Figure 6.6 shows an example of the code added to the transform, as a result of the code shown in figure 6.5 being executed.

```
<path>
   <!-- user defined function to recursively build the required string -->
   <xsl:value-of select="f:buildInstructionPath(.)"/>
</path>
```

Figure 6.5 — Transform instruction path calculating code fragment

```
<state>
   <!-- ... -->
   <path>/1/3/5/8/1</path>
   <!-- ... -->
</state>
```

Figure 6.6 — Instruction path code added to original transform

6.1.2 Position

Calculating the current position during execution is made simple by the XPath `position()` function. This returns an integer representing the index position of the node that is currently being processed. Since we need to call this function when the transform is being executed, the code shown in figure 6.7 is added so that the position value is calculated at the time when the state information is produced.

```
<state>
   <!-- ... -->
   <position>
      <xsl:value-of select="position()"/>
   </position>
   <!-- ... -->
</state>
```

Figure 6.7 — Position value calculating code fragment

6.1.3 Variables

Storing the currently available variables can be treated as two separate exercises — firstly identifying the variables that are in scope at the current point in the transform, and secondly calculating and storing the values of each variable.

The process of identifying in-scope variables can be performed during the modification of the original XSLT script, whereas calculating and storing their values must be done during execution of the modified script. A list of relevant variables can be obtained by searching up the input tree from the current node and, for each ancestor, checking whether any of the nodes on the preceding-sibling axis are variable elements. For each variable that is found, and which is therefore in scope, we can add an element to the output state tree that holds the name of the variable as well as code that will calculate its type, and value, when executed. The type and value calculations are required since a variable may hold different values ranging from integers, to sequences and input tree node references, each of which must be stored differently. For example, integers and sequences can be copied simply by using an `xsl:copy-of` instruction, but this is insufficient for input tree node references since any ancestral information is then lost. Therefore, input tree node references must be identified and a path to the node generated and stored in the state output.

Figure 6.8 shows the code that is added to the original XSLT script. The ancillary functions that are used within this code are summarised in figures 6.9 and 6.10.

```
<vars>
   <var name="foo" type="{f:getValueType($foo)}">
      <xsl:value-of select="f:serializeVariableValue($foo)"/>
   </var>
</vars>
```

Figure 6.8 — Variable/Parameter name calculation code fragment

```
<xsl:function name="f:getCurrentMode">
   <xsl:param name="node"/>
   <xsl:value-of select="($node/ancestor-or-self::mode, '#default')[1]"/>
</xsl:function>
<xsl:function name="f:serializeVariable" as="xs:string">
   <xsl:param name="var"/>
   <xsl:choose>
      <xsl:when test="$var instance of xs:integer">
         <xsl:value-of select="$var"/>
      </xsl:when>
      <xsl:when test="$var instance of xs:string">
         <xsl:value-of select="$var"/>
      </xsl:when>
      <xsl:when test="$var instance of element()">
         <xsl:variable name="atts" select="[ommited for simplicity]"/>
         <xsl:variable name="children" select="f:serializeVariable(*)"/>
         <xsl:value-of select="concat('<', name($var), ' ', $atts, '>',
             $children, '</', name($var), '>')"/>
      </xsl:when>
      <!-- ... more tests ... -->
      <xsl:otherwise>
         <xsl:text>Error</xsl:text>
      </xsl:otherwise>
   </xsl:choose>
</xsl:function>
```

Figure 6.9 — Variable value calculation function

```
<xsl:function name="f:getValueType" as="xs:string">
   <xsl:param name="var"/>
   <xsl:choose>
      <xsl:when test="$var instance of xs:decimal">
         <xsl:text>xs:decimal</xsl:text>
      </xsl:when>
      <xsl:when test="$var instance of xs:string">
         <xsl:text>xs:string</xsl:text>
      </xsl:when>
```

```
        <xsl:when test="$var instance of element()">
            <xsl:text>xs:element</xsl:text>
        </xsl:when>
        <!-- ... more tests ... -->
        <xsl:otherwise>
            <xsl:text>[unknown]</xsl:text>
        </xsl:otherwise>
    </xsl:choose>
</xsl:function>
```

Figure 6.10 — Value type calculation function

6.1.4 Parameters

Parameters are handled in much the same way as variables since they are simply variables that are defined at the top of a template or function, with their values being set when the template/function is executed. As described in chapter 2.2, parameters can be tunnelled through template/function calls, therefore each entry in the generated state is decorated with a `tunnel` attribute signifying whether it is tunnelled or not. Figure 6.9 shows an example piece of code that might be added.

```
<state>
    <!-- ... -->
    <params>
        <param name="foo" tunnel="no" type="{f:getValueType($foo)}">
            <xsl:value-of select="f:serializeVariableValue($foo)"/>
        </param>
        <param name="bar" tunnel="yes" type="{f:getValueType($bar)}">
            <xsl:value-of select="f:serializeVariableValue($bar)"/>
        </param>
    </params>
    <!-- ... -->
</state>
```

Figure 6.11 — Parameter records producing code fragment

Each `param` entry is created by searching up the input tree to find any `xsl:params` of the template/function that the current instruction is a descendant of. The name is stored at this point, with the value calculation being deferred to the actual stylesheet execution by adding the call to an ancillary function. As with variables, the serialized form of the value is built by the `serializeVariableValue` function with the `type` attribute indicating the serialized format being built by the `getValueType` function.

6.1.5 Tunneled Parameters

Tunnelled parameters present a problem when trying to store the current processing state. At any given point in the stylesheet processing it is not possible to access any tunnelled parameters that have not been declared as being used in the current template or function. For example, it is not possible in the code sample shown in figure 6.12 to access the tunnelled parameter `foo` from template `b` (or even be aware that it exists). Therefore, if we were to save the state of the processor at any point during the execution of template `b` we would need to store the value of `foo` (which in this case is `red`, having been tunnelled from the initial template), but would be unable to.

```
<xsl:template match="/">
   <xsl:call-template name="a">
      <xsl:with-param tunnel="yes" name="foo">red</xsl:with-param>
   </xsl:call-template>
</xsl:template>

<xsl:template name="a">
   <xsl:param name="foo">blue</xsl:param>
   <xsl:call-template name="b"/>
</xsl:template>

<xsl:template name="b">
   <xsl:call-template name="c"/>
</xsl:template>

<xsl:template name="c">
   <xsl:param name="foo" tunnel="yes">green</xsl:param>
</xsl:template>
```

Figure 6.12 — Tunneled variables example

A solution to this problem is to build a list of all tunnelled parameters in the entire stylesheet and to output this, along with any values that are currently available, as part of the stored state information. As well as the inherent inefficiency of storing every tunnelled parameter, even when it may not be in scope, this requires each template in the stylesheet to be re-written so as to allow for the parameters to be accepted and subsequently accessed. There is also a further requirement of renaming all of the tunnelled parameters to avoid conflicts with existing parameter declarations.

6.1.6 Context Node

Obtaining the context node is a relatively trivial exercise since it can be referenced using the '.' (dot) operator. However, to generate a meaningful entry in the output state requires a similar approach to that taken when dealing with variables and parameters. The context node is often a node in the input tree, but it can also be some other object such as a node in a constructed sequence. Therefore, simply copying the node to the output is not useful; instead we must build a path that can be used to reference the node. This path can then be output as a literal string as part of the state tree that accompanies the original result element.

Figure 6.13 shows the inline code that is added to the transform which, in turn, calls the function shown in figure 6.14 that is also added to the modified XSLT script.

```
<state>
   <!-- ... -->
   <context-node>
      <xsl:value-of select="f:buildContextPath(.)"/>
   </context-node>
   <!-- ... -->
</state>
```

Figure 6.13 — Context node producing code fragment

```
<xsl:function name="f:buildContextPath" as="xs:string">
   <xsl:param name="node"/>
   <xsl:choose>
      <xsl:when test="$node/parent::*">
         <xsl:value-of select="concat(f:buildContextPath($node/parent::*),
             '/', count($node/preceding-sibling::*) + 1)"/>
      </xsl:when>
      <xsl:otherwise>
         <xsl:text>/1</xsl:text>
      </xsl:otherwise>
   </xsl:choose>
</xsl:function>
```

Figure 6.14 — Context node path-building function

6.1.7 Mode

The mode in which the processor is currently executing must be stored because the resulting execution path followed by the processor may vary depending on the mode. Since modes apply to functions and templates, it might seem trivial to store the value of the mode attribute on the ancestor template/function and to store the `#default` mode value when none is present. Unfortunately, this approach does not take into consideration the possibility of a template or function that specifies `any` as its mode attribute value. The `any` value allows the template/function to be called irrespective of any specified mode. Therefore, finding the current mode at a given point in the script is not simply a case of copying an attribute value, since we may be in a *specific* mode but within a template/function that indicates it can be called in *any* mode. As a result, we must obtain the value of the current mode at execution time, and there is no easy way to query the mode value within XSLT or XPath. We must therefore rely on extension functions that are supported by the processor to provide us with this information. This kind of support is analogous to making modifications to the processor itself and so will not be discussed any further here since the process is covered in more detail later in the chapter.

```
<mode>
   <xsl:value-of select="'#default'"/>
</mode>
```

Figure 6.15 — Mode value annotating code fragment

An example of the code needed to produce the `mode` element in the state tree is given in figure 6.15.

Possible Problems

One of the problems with storing state as output elements is ensuring that the extra state information stored alongside the original output, does not alter the execution of the script. This problem manifests itself in the example code fragment shown in figure 6.16.

```
<xsl:template match="/">
   <xsl:variable name="foo">
      <xsl:apply-templates/>
   </xsl:variable>
   <xsl:choose>
      <xsl:when test="count($foo) eq 1">
         <svg:rect fill="green"/>
      </xsl:when>
      <xsl:otherwise>
         <svg:rect fill="blue"/>
      </xsl:otherwise>
   </xsl:choose>
</xsl:template>

<xsl:template match="*">
   <xsl:copy select="*"/>
</xsl:template>
```

Figure 6.16 — Example XSLT code fragment before modification

Consider what would happen if extra code, to generate state information, were added to the `xsl:copy` element in the second template as shown in figure 6.17. The sequence returned by the second template will now contain state elements as well as any child elements produced as standard output. Since we capture the result in a variable, `foo`, and conditionally branch depending on the number of items in that sequence, we can no longer produce the green rectangle because there is *always* an even number of items returned since the elements output are paired with state elements.

```
<xsl:template match="/">
   <xsl:variable name="foo">
      <xsl:apply-templates/>
   </xsl:variable>
   <xsl:choose>
      <xsl:when test="count($foo) eq 1">
         <svg:rect fill="green"/>
         <state>
            <mode>#default</mode>
            <position>
               <xsl:value-of select="position()"/>
            </position>
            <!-- etc. -->
         </state>
      </xsl:when>
      <xsl:otherwise>
         <svg:rect fill="blue"/>
         <state>
            <mode>#default</mode>
            <position>
               <xsl:value-of select="position()"/>
            </position>
            <!-- etc. -->
         </state>
      </xsl:otherwise>
   </xsl:choose>
</xsl:template>

<xsl:template match="*">
   <xsl:copy select="*"/>
   <state>
      <mode>#default</mode>
      <position>
         <xsl:value-of select="position()"/>
      </position>
      <!-- etc. -->
   </state>
</xsl:template>
```

Figure 6.17 — Generating state information as elements

Storing State Information as Attributes

One way to limit this type of problem is to return the state information as attributes on the original result elements rather than producing entirely new state elements. This reduces the number of situations where we may cause problems by outputting extra information, since the guarantees made by the XML specification are less restrictive for attributes than elements. For example, the order of attributes is not guaranteed unlike with elements, therefore any code in our original transform that relies on properties such as this can be considered 'incorrect' and ignored. There is, however, nothing preventing the author of an XSLT script, for example, counting the number of attributes on an element and, although this is generally considered bad practice, it is entirely legal. Therefore, following such a scheme allows us to reduce the number of situations where problems arise, but not to eliminate them entirely.

```
<xsl:template match="/">
   <xsl:variable name="foo">
      <xsl:apply-templates/>
   </xsl:variable>
   <xsl:choose>
      <xsl:when test="count($foo) eq 1">
         <svg:rect fill="green">
            <xsl:attribute name="mode" select="'#default'"/>
            <xsl:attribute name="position" select="position()"/>
            <!-- other state attributes -->
         </svg:rect>
      </xsl:when>
      <xsl:otherwise>
         <svg:rect fill="blue">
            <xsl:attribute name="mode" select="'#default'"/>
            <xsl:attribute name="position" select="position()"/>
            <!-- other state attributes -->
         </svg:rect>
      </xsl:otherwise>
   </xsl:choose>
</xsl:template>

<xsl:template match="*">
   <xsl:copy select="*">
      <xsl:attribute name="mode" select="'#default'"/>
      <xsl:attribute name="position" select="position()"/>
      <!-- other state attributes -->
   </xsl:copy>
</xsl:template>
```

Figure 6.18 — Generating state information as attributes

Storing the state information as attributes on result elements brings its own set of problems — most notably the issues arising from having to store all values as textual strings. Some pieces of information, such as the XSLT instruction path and current mode, can easily be represented as string attribute values. However things are not so simple when dealing with variables and parameters.

Firstly, when outputting variables and parameters, there may be multiple values to be stored. For example, when storing the current mode we can simply name the attribute `mode` and store the appropriate value alongside it. To avoid conflict with any exisiting attribute called `mode` we can place the new attribute in our new namespace[2]. However, when dealing with variables and parameters there are often more than one of them that is in scope and therefore needs recording. Namespaces allow us to avoid conflict with existing attributes, but they do not help when we have conflicts among our own added attributes. Therefore, we cannot simply use multiple attributes called `variable` or `parameter` (the XML specfication does not allow it[3]) and so we must devise an alternative strategy.

Since there are two pieces of information stored in each variable/parameter (its name and its value), there is a possibility of using the name of each variable/parameter as the name of the attribute. At first this might seem a sensible solution, but there are some problems with it. Consider what happens if we have a variable called `mode` — this would again cause a conflict with our existing state attributes. The conflict could be solved, for example, by prepending '`var_`' to each attribute name, but there remains another issue. We have no information about the number of variables/parameters, so we must assume that any unknown attribute must be treated as a variable/parameter. If, for whatever reason, we were to add further information at a later stage this could lead to severe problems.

Given the issues discussed above, the following scheme was devised for storing variables, parameters and tunnelled parameters:

[2] Namespaces have not been included in the examples for clarity.

[3] Trying to generate an element with multiple atrtibutes with the same name does not fail, but instead results in an element with a single attribute whose value was overwritten mutliple times.

- There will be three attributes, `varcount`, `paramcount` and `tunnelcount`, each of which will hold the number of values to be stored.
- Variables/parameters will be stored in three parts — their names, types and values will be stored separately.
- Variables/parameters will be numbered sequentially to avoid conflicts

Therefore, when storing a state where two variables, `foo` and `bar`, were in scope with the values `42` and `"a_string"` respectively, the following attributes would be generated:

```
<svg:rect varcount="2" var1_name="foo" var2_name="bar"
    var1_type="xs:integer" var2_type="xs:string" var1_value="42"
    var2_value="a_string"/>
```

Figure 6.19 — State attributes example (variables)

This scheme works well in the type of situation shown in figure 6.19, where the values held in the variables/parameters are simple numerical values or strings. However, as mentioned earlier, further problems, such as tree fragments and references to nodes in the input tree, arise when variables/parameters contain more complex structures that are not easily stored as attribute value strings.

Serializing Values

Clearly, all variable and parameter values need to be serialized to a string, however how this should be done is not immediately clear. As shown in figure 6.19, literal string values and numerical values are trivial to serialize. Serializing references to nodes within the input tree can be done by constructing the path of the referenced node within the input and storing that as the attribute value. Variables/parameters holding tree fragments, such as that shown in figure 6.20 are serialized to standard XML markup. Figure 6.21 shows the serilized version of the tree shown in figure 6.20.

```
<xsl:variable name="foo">
   <lut>
      <bar key="1">January</bar>
      <bar key="2">February</bar>
      <bar key="3">March</bar>
      <!-- etc -->
   </lut>
</xsl:variable>
```

Figure 6.20 — Example variable holding tree fragment

```
<result-element var_count="1" var1_name="foo" var1_type="literaltree"
    var1_value="&lt;lut&gt;&lt;bar key=&quot;1&quot;&gt;January&lt;/
   bar&gt; ..."/>
```

Figure 6.21 — Serialized form of variable holding tree fragment

Although we have discussed the different ways in which the various types of variable/parameter values can be stored, there remains the question of initially indentifying the type of a given variable/parameter. Fortunately we can easily test for the primitive value types such as strings and integers etc. using the XPath `instance of` keyword and the value types defined by the XML Schema specification [50]. An example of such a test can be seen in figure 6.22.

```
<xsl:if test="$foo instance of xs:string">
   <!-- return the type of the variable/parameter as a string -->
   xs:string
</xsl:if>
<!-- ... more type tests ... -->
```

Figure 6.22 — Example variable/parameter test for a primitive type

Having handled the primitive variable/parameter values, we are left with differentiating between references to nodes in the input tree and other stored structures such as temporary tree fragments. A simple solution to this problem is to test whether the stored node has a parent node. All nodes in the input XML document will have a parent node (except the document node which will be handled separately), whereas temporary tree fragments will not. Having differentiated between these types of variable, they must be serialized as previously described.

The approach described here is one that immediately stands out as a possibility because of the fact that it can be implemented entriely within the language that is the subject of the transformation. Therefore, it is important that its suitability is evaluated, but given the now-apparent shortcomings of modifying the transform in order to generate the required state information, we now look at the alternative approach of modifying the XSLT processor itself.

6.2 Saxon Modifications

In contrast to modifying the stylesheet, making modifications to the processor itself provides us with a much wider range of possibilities. Working within the processor has two major benefits: we have direct access to the processor's internal data structures and we can control how the result document is produced. The first of these allows for a much more optimal, and simple, way of obtaining the state of the processor as the stylesheet is executed, and the latter allows us to overcome the issues raised in relation to outputting the infomation that has been gathered. We now look in detial at the different solutions made possible by this low-level access, as well as the necessary modifications to support them.

6.2.1 Gathering State Information

Since we are free to make whatever changes are necessary within Saxon, we are able to delve directly into the stylesheet execution code to gain access to the various pieces of information we require, as and when the stylesheet is executed. As described in chapter 5, the execution of the stylesheet is performed by cascading calls to either of the `process` or `procesLeavingTail` methods available on all

`Instruction` and `Expression` objects that are used to represent the nodes within the stylesheet. Both of these methods require an `XPathContext` object to be passed to them, which is responsible for providing, and maintaining, the dynamic execution context of the processor. Therefore, many of the pieces of information that constitute the execution state of the processor as a whole can be obtained from this single source.

There are two approaches that can be taken when collecting the required information from the `XPathContext` object (as well as other sources as we shall see shortly). Firstly, and perhaps rather naïvely, it is possible to request each piece of information separately, as and when it is required to be output or stored. This is a simple solution since the majority of the data is made available in a suitable way by the `XPathContext` object as part of its normal functionality. Only minor modifications are necessary to allow access to that data that is not readily available. The second, and slightly more complex, solution is to further modify the inner workings of the `XPathContext` object so that each time a value is explicitly set or otherwise changed, an external agent is notified of that change. This agent can then be queried for the data as in the first solution. Although this adds another layer of abstraction, and in this case it actually makes the retrieval of data *less* efficient (because extra work is being done, but the same queries are being made and serviced), we shall soon see how it can be used as part of a more general mechanism that is both coherent and also ultimately *more* efficient.

In addition to the data maintained by the `XPathContext` object, there are some other pieces of information that must also be stored in order to support partial re-evaluation of the stylesheet. The most obvious of these is a reference to the

`Instruction` currently being evaluated. Clearly, this information is essential when re-initialising the processor at a particular point in the stylesheet and so must be included in the information that is stored, however it is not known to, or maintained by, the `XPathContext` object and therefore must be found by some other means. The solution is to augment the execution of the individual `Instruction`s that make up Saxon's internal (compiled) representation of the stylesheet. This is an attractive solution since it is references to these `Instruction`s that are required, however they are compiled forms of the `StyleElement` objects in the original stylesheet representation and have no information pertaining to their position in the stylesheet. To provide access to this information, the compilation code for each `StyleElement` that prodcues an `Instruction` is augmented to provide the newly compiled `Instruction` with a reference to its corresponding `StyleElement`. This reference can then be used to calculate the location path of the node in the stylesheet by querying its ancestor nodes. In this way the required reference can be calculated from within the `Instruction` as is required.

6.2.2 Outputting State Information

Efficient retrieval of the information needed to support partial re-evaluation of the stylesheet is another challenge. For this information to be useful during the re-evaulation process, it must be easily accessible not only to re-configure the processor, but also to be linked to all relevant parts of the result document. Therefore, once the state information has been obtained, it must be stored in a sensible way.

Two alternative approaches to storing the data have been discussed earlier in this chapter when we considered augmentation of the stylesheet. Both of these (storing the data as elements or as attributes on the result document nodes) suffer from the same shortcomings irrespective of the way they are generated. However, looking at how we might implement storage from within the processor is a worthwhile exercise since it inevitably has some commonality with any other approaches that might otherise be considered.

The obvious place to start for changing the output produced by the processor is in the `Instruction`s responsible for producing the output from the stylesheet. There are a limited number of elements in an XSLT stylesheet that can generate elements in the result document, namely:

- `<xsl:element>`
- `<xsl:copy>`
- `<xsl:copy-of>`
- Literal result elements, such as `<svg:rect>`

Therefore, it is the `Instruction` objects that correspond to these elements that will now be the subject of our attentions. Unsurprisingly, the commonality in function between these instructions has resulted in a common relationship in the object-oriented design of the relevant Saxon classes. All of the corresponding `Instructions`s (`ComputedElement`, `Copy`, `CopyOf` and `FixedElement`) are sub-classes of the abstract `ElementCreator` class. This means that there is a common point from which all output to the result document is generated, and it is at this point that the output can easily be augmented with whichever elements or attributes are deemed necessary. Also, given the fact that this common point of execution within the `ElementCreator` class is the `processLeavingTail` method, we can directly generate the extra output as described above. Supplementary code can be added within the method such that each piece of information is retrieved from the

XPathContext object, and/or other sources, as necessary, and the extra elements or attributes are directly added to the result element currently being produced. This augmentation is illustrated in the truncated code example shown in figure 6.23 of the processLeavingTail method within ElementCreator.

```
public TailCall processLeavingTail(XPathContext
    context) throws XPathException {
    //selection and preparation of output destination
    //selected destination is called 'out'

    //begin outputting the current result element
    out.startElement(...);

    //state output code inserted here

    //process any child Instruction/Expression(s)
    content.process(context);

    //end output of current result element
    out.endElement();
}
```

Figure 6.23 — State augmentation of truncated `processLeavingTail` method

This type of modification to the compiled styleheet instructions allows for other solutions to the problem of outputting state information beyond what has been discussed so far. An alternative to annotating the elements in the result document with numerous extra elements and/or attributes is to create a separate output steam for the state information and to put a single 'ID' attribute on each element produced, which links it to the correct position in the stored state. By taking this approach we can minimise, but not entirely eliminate, the effect on the original execution of the stylesheet as is the case with the original solution, However, the possibility of storing the state information separately to the stylesheet's output and then referencing it from the result document elements is one that can be achieved by going beyond the modifications presented here.

In the code example given in figure 6.23, methods are called on an object (‘out’ in the example) to indicate the structure and content of the result document to be produced. Under the default configuration of Saxon, this object is an instance of the `DOMWriter` class and it produces a standard DOM structure using the Xerces DOM implementation [51]. However, this behaviour can be altered through changes to the configuration of Saxon (as discussed in chapter 5) or, alternatively, can be modified by directly implementing changes to the existing classes. By providing an alternative implementation for constructing the result document, we are able to store the required state information without any potentially adverse effects on the execution of the stylesheet. The modifications made in order to support this soultion are the subject of the next section.

6.2.3 An Alternative Output DOM Implementation

The default Xerces DOM implementation, used by Saxon, conforms to the W3C DOM specification through the implmentation of a set of Java interfaces produced by the W3C. Furthermore, although the Xerces implementation is used to construct the result document, this fact is not guaranteed, nor even advertised, by Saxon since it is only made available to subsequent internal or external tools as an implementation of the abstact W3C interface. Therefore, we are free to replace the Xerces implementation with our own, so long as it conforms to the W3C interfaces as advertised by Saxon.

At first, this may appear to be irrelevant since any tools that operated on the result document could not do anything with it beyond what was specified in the W3C interfaces. However, since we are intending this result document to be partially re-consumed by the tool that created it (i.e. partial re-evaluation performed internally within Saxon), knowledge of the underlying implementation can be utilized internally whilst still exposing the standard W3C interfaces for any other tools that may use the final output. Therefore, we can modify the previous, ID-based approach by no longer storing the ID as an attribute on the result element (thereby removing the possible cause of problems when executing the stylesheet), but rather storing a reference to the state information as part of the class description used to represent elements in our DOM implementation. Because this reference is a property of the implementing class, and not the W3C interface, it can be accessed by any object that has knowledge of its concrete implementation (i.e. the modified objects internal to Saxon), yet it is entirely invisible to external tools that operate only on the DOM according to the methods defined in the W3C interfaces.

As a basis upon which to create the annotation-supporting DOM, the Xerces class hierarchy is a good model to follow and so the types and functionality of the objects created, as well as their relations, are similar to those used within Xerces. However, to allow annotations to be added and subsequently accessed, a new annotated element object is required. This is a subclass of the standard element object implementation and extends it by providing methods (and associated internal fields) to store and retrieve references to all of the state information that is relevant to the element. The overall structure of the DOM, and the relations between the objects, are discussed in more detail in chapter 8, since the design implications that result from the re-evaluation selection strategy discussed therein have a significant effect on these issues. For the discussions presented here, regarding the production and storage of state information, it is possible to concentrate solely on the object representing the annotated element and the objects and other data that constitutes the processor's state, without the need to discuss other aspects of the DOM implementation.

Given that we now have a mechanism for annotating the result document elements with information, without affecting the evaluation of the stylesheet, it makes sense to revisit the way we gather this information and, thus, how it is stored within the annotated element object. In the initial approach, each piece of information would be stored either as descendant elements of the result element, or as serialized string values of attributes on that element. As such, each piece of information was accessed every time the state information was output alongside a result element. However, since the information being stored can be *referenced by*, rather than *copied to*, the result element, it needs only to be updated as and when a value is changed. The class used to fascilitate this process is the `ProcessorState` class.

6.2.4 Optimised State Storage

Simply copying the value of every piece of state data onto each result element, with the correct value at the relevant point of execution, is not only computationally expensive, but it is also unnecessary and redundant. In many cases, the values change only infrequently, and rarely all at once, so the naïveity of producing multiple copies for every result element is clear. As an example, consider the execution of the sample template shown in figure 6.24.

```
<xsl:template match="foo" mode="bar">
   <xsl:variable name="cnt" select="count(*)"/>
   <svg:text>Some example text</svg:text>
   <svg:rect x="10" y="10" width="{100 * $cnt}" height="100"/>
</xsl:template>
```

Figure 6.24 — Example template with minimally changing execution state

The result of the execution of this template is that two result elements will be produced; an `<svg:text>` element containing the text ‘Some example text’ and an `<svg:rect>` with a width proportional to the number of child elements found on the foo element that the template has matched on. The state informaton stored on each of these elements will be mostly identical, with the obvious exception of the reference to the instruction that created it. For example, the variable declared before the two result elements is in scope for the entire template, the mode in which the instructions execute does not change within the template, and the context node does not change from the foo element that the template matches on. However, each of these pieces of information must be calculated or otherwise obtained for *both* of the result elements.

The idea of limiting the calculation of state values to when they are set or changed was introduced in section 6.2.1, and it is that approach we return to here. Instead of *fetching* the data every time a result element is generated, it is more efficient to notify an specialist class (ProcessorState) when a value is set or changed and then to place a reference to the correct point within the data structure that is created onto the relevant result element object. Therefore, the subclasses of Instruction and XPathContext are modified to alert the ProcessorState class of changes to the processor's state through static method calls, as well as when a new Instruction is executed so that a 'state-tree' can be built and maintained. Each time a value is changed (i.e. the current execution mode changes, a variable is declared, etc.) the state tree is updated either by storing the value on the current ProcessorState node or, if necessary, creating a new node as a child of the current one. In this way, the complete state of the processor can be obtained by concatenating the current state with all of its ancestors, whilst minimizing the amount of memory required to hold the entire tree.

The final part of the solution is to simply link the result element object to the StyleElement reference corresponding to the Instruction that created it, as well as a reference to the current ProcessorState object. Because the default DOMWriter implementation is unaware of the need to store state information, it constructs the result elements as the result of method calls that do not take the StyleElement or ProcessorState references as parameters. Therefore, the methods were altered to accept these parameters, and the code contained within them was changed to create and work with annotated element objects, as opposed to the standard non-annotated objects that would otherwise have been used. The end result of this process is, therefore, a result DOM that complies with the standard W3C interfaces, yet contains references to all of the information regarding the state of the processor when each element was created.

As we shall see in later chapters, there is more data that must also be stored in order to support the techniques use for triggering the re-evaluation of Instructions. The details of this extra data, as well as how its storage is incorporated into the mechanisms and data structures described in this chapter, are presented alongside the relevant discussions. We now consider the implications of user edits to the document, how these are reflected in the input data and/or stylesheet, and how such changes are effected within the data structures within the processor.

Chapter 7:

Effecting Document Edits

The previous chapter has detailed the modifications made to the document processing framework to support the storing of state information during document evaluation. Although this state can easily be used to re-initialise the processor and re-execute specific instructions, support for effecting edits to the document must also be provided to avoid simply producing identical output to that already generated.

In an editing environment, the need for re-evaluation is a consequence of edits made to the document. In the case of the variable data documents presented earlier, these edits can be either changes to the stylesheet or to the variable input data. This present chapter considers both of these and how such changes affect the re-evaluation process.

7.1 Changes to the Data

The “variable” nature of a variable-data document is a consequence of the fact that the input data changes from one instance to the next. Therefore, when editing such a document, it is important that the author is able to alter the input data instance so that the full variability of the document can be explored and its effect on the result document observed. As discussed in chapter 3, this can either be achieved by generating a series of example documents using sample input data or, alternatively, it can be done in a piecemeal fashion through manual manipulation of the data. Such manual manipulation requires a mechanism to allow changes to the input data tree to be made, whilst remaining compatible with the state storage scheme described in the previous chapter.

7.1.1 Modifying the Input Data DOM

The default implementation used by Saxon to store the input data tree is a custom DOM-like structure that can be manipulated in much the same way as a standard DOM would allow. Therefore, a simple solution is to build a mechanism through which authors can change the data via some aspect of the editor's interface. This would culminate in nodes within the tree being added, removed or otherwise altered. There is, however, a variation on this approach that allows for a more convenient, and potentially more useful, solution.

Rather than having a data structure to represent a single input data instance that can be arbitrarily changed or replaced, it is possible to construct a *switchable* structure that encompasses a range of related input instances, which at any given time presents itself as a single instance. As an example, consider the data instances shown in figure 7.1, where each instance is a child of the root node of the document.

```
<data>
   <instance>
      <surname>Rubble</surname>
      <forename>Joe</forename>
      <address>
         <line>Room A01</line>
         <line>School of Computer Science</line>
         <line>University of Nottingham</line>
         <line>Jubilee Campus</line>
         <line>Nottingham</line>
         <line>NG8 1BB</line>
         <line>UK</line>
      </address>
   </instance>
   <instance>
      <surname>Higginbotham</surname>
      <forename>Barney</forename>
      <address>
         <line>3 Priory Mews</line>
         <line>Nottingham</line>
         <line>NG7 1AB</line>
      </address>
   </instance>
   <instance>
      <surname>Smith</surname>
      <forename>Samantha</forename>
      <address>
         <line>1 Aldridge Close</line>
         <line>Toton</line>
         <line>Beeston</line>
         <line>Nottingham</line>
         <line>NG9 9YZ</line>
      </address>
   </instance>
   <!-- more instances -->
</data>
```

Figure 7.1 — Example data instances

Each of the three instances contains contact information pertaining to an individual who is the subject of the document being authored. Ordinarily, variable data document editors (such as those described in chapter 3) would produce a result document instance based upon the data provided for each input data instance. Evidently, this would result in three documents being produced. However, by merging the three instances into a single structure, we can produce a template for a wider variety of instances. Upon closer inspection it can be seen that the constituent pieces of data exhibit a range of values across the instances — the length of the names of the individuals ranges from very short to long, as do the addresses, which range from three lines to seven lines in length. Therefore, from these three simple instances, it is possible to construct an 'artificial' instance that combines a short name with a short address, or a long name with a long address, without the need for any more raw data instances.

There are two ways in which such combinations can be generated; either a new document can be created from the original instances each and every time a variation is requested, or *all three* instances can be combined into a single structure that can be morphed into any combination as needed. The first option of creating a new data instance, or even simply replacing nodes within it, can cause problems when tracking the usage of the data nodes, which is necessary for the automatic re-evaluation mechanism discussed in the following chapter. Therefore, the construction of a single, final-form structure containing all aspects of the available data instances is the preferred solution. However, even this solution is not without its problems, most notably that building such a structure requires a sufficiently diverse, yet manageable, set of data instances to be collated to make the alternative values useful.

7.1.2 Collating Sample Data Instances

Having a large set of data instances upon which a document will be based, is not necessarily a common situation for a document author. Often, documents are created ahead of time with the variable data being provided at some later point prior to printing. A good example of this is would be advertising leaflets based on the current shopping habits of supermarket loyalty card holders. The types of products that particular customers are to be presented with, as well as the details of the products themselves, are likely to be collected as close to the time of printing as possible, with the overall document design being completed by the document author well in advance.

In such circumstances, the *exact* details of each customer are not necessary, but rather a *representative* data set is all that is required. This can be produced in one of two ways:

- Use an existing data set that encompasses similar types of data
- Use a specially constructed sample data set that exhibits the expected range of variable data

Clearly, using or adapting real data is preferred, but a carefully constructed sample data set can have its own advantages. In fact, simply merging numerous real data records together into one amorphous structure is not necessarily useful when it comes to selecting alternative data values. Having too many choices reduces the effectiveness of such an editing paradigm and suffers from a similar problem to that of existing editors generating numerous proof documents as decribed in chapter 3. Therefore, an ideal solution would be a simplified sample data set that is automatically constructed from a set of real data instances, including metadata annotations calculated during the construction process. However, before we consider how such a data set might be created, we first look at the process of simply merging data instances into a single structure.

Combining Data Instances

Combining multiple raw data instances, such as those shown in figure 7.1, into a single switchable structure might seem straightforward, but that is not always the case. Combining instances with identical tree structures allows the structure of the combined 'instance' simply to be copied from any of the raw instances, with the textual content of each node being added to a list of alternatives at the corresponding points in the tree. In the example given, this is the case with the name elements, but when we consider the address part of the data we have a more complex problem. Since the number of `<line>` elements changes from one instance to the next, the way in which this should be represented in the combined structure is not immediately obvious. In more complex data sets, where parts of the tree structure (including attributes) may be optionally omitted in any given instance, merging the instances into a single structure becomes yet harder.

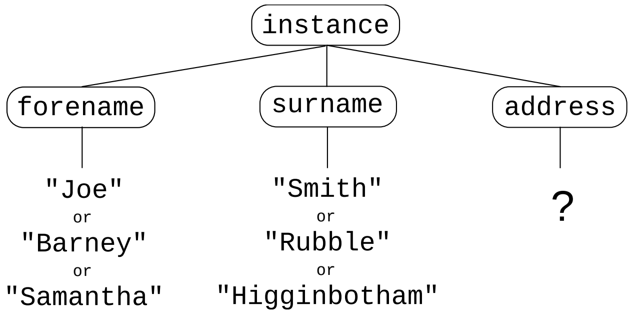


Figure 7.2 — Combining structurally dissimilar data instances

Access to a Document Type Definition (DTD) or schema[1], would provide us with more information regarding the allowable structure of the instances. This would potentially make the problem easier to tackle, however a suitable DTD or schema is not always available. The work of Chidlovskii on extraction of schemas from collections of XML documents [54] and XML document creation using structural suggestions [55] offer further possibilites towards this goal. The task of merging documents with similar, or near-identical, structure is also central to research aimed at efficient and effective version control and merging of XML documents and there is a wide range of existing literature relating to the subject [56, 57]. Lindholm [58] proposes a three-way merging algorithm in which a document instance and two derivative instances are compared and merged to produce a single

[1] This could be a traditional XML schema as defined by the W3C specification [52] or an alternative such as the specifications defined by RelaxNG [53]

instance containing the variations from both of the derived instances, and Rönnau *et al*[59,60] propose a solution of 'context fingerprinting' to facilitate the merging of two related XML documents. Inspiration could be taken from these methodologies when developing a robust solution to the problem, but a complete solution is outside the scope of this thesis. Therefore we shall not consider this problem any further, but instead we introduce a simplified model, and an accompanying tagset for its definition, that supports the common aspects of a merged data set. This, less complex, model still provides sufficient descriptive power to create combined instances from data sets that exhibit a range of types of variation, while not becoming obfuscated by the details of a totally comprehensive solution.

7.1.3 Switchable Data Description

We have already discussed the most basic of situations where the structure of all the raw instances is identical and fixed, and it is only the textual content of the leaf nodes that varies. All other situations must, therefore, involve some variation in the structure of the document, and so a way of handling this in the instance definition is necessary. Specific cases that the model must support include:

- Elements that may optionally be omitted from the instance
- Elements that are mutually exclusive (only one element from a set is included)
- Attributes that may be omitted from the instance
- Attributes that have varying values across different instances

Clearly, the model must also support the declaration of fixed elements, attributes and textual values for parts of the tree that do not change from one instance to the next.

We now consider each aspect of the proposed model, including the markup used for its declaration, and discuss how they relate to supporting the cases described above. The tagset that is presented represents a meta-description of the switchable instance rather than a direct declaration of it. To avoid issues with conflicting names when handling and processing the switchable instance description, the constituent elements are defined in our own (`alt`) namespace.

Elements

One of the most basic requirements for the model is that it allows fixed elements (those that are known not to change across the various instances) to be included. Because of the fixed nature of the element, it can simply be declared as an `<alt:element>` element with details regarding its name, namespace URI and namespace prefix defined as appropriate attributes[2]. Any 'real' attributes that are to be present on the fixed element are declared as child descriptor elements that are described in the following section.

An example of how this type of descriptor element is used is shown in figure 7.3. The example shows the meta-description of a `<foo:bar>` element to be inlcuded in the switchable instance where the `foo` prefix is bound the namespace `http://www.example.com`.

```
<alt:element localname="bar" uri="http://www.example.com" prefix="foo">
   <!-- ... -->
</alt:element>
```

Figure 7.3 — Example instance descriptor for a fixed element

[2] The `prefix` and `uri` atrtributes can be ommitted if the element belongs to the default namespace.

Attributes

As mentioned in the previous section, attributes are not declared on the `<alt:element>` to which they belong, but rather they are declared as child elements of it. Each attribute is represented by an `<alt:attribute>` element, with the details of its name, prefix and namespace URI being provided as attributes in the same way that they are specified on an `<alt:element>` element. The attribute's value is also provided as an attribute, called `value`, on the `<alt:attribute>` element. Therefore, the fixed element (with fixed attributes)

```
<book publication_date="1st Jan 2010" revision="1.0"/>
```

would be represented as shown in figure 7.4

```
<alt:element localname="book">
   <alt:attribute localname="publication_date" value="1st Jan 2010"/>
   <alt:attribute localname="revision" value="1.0"/>
</alt:element>
```

Figure 7.4 — Example instance descriptor for fixed attributes

Text

It is entirely plausible to include invariant textual content of an element as a direct child of the `<alt:element>` as shown in figure 7.5, but this approach can cause ambiguity when we try and represent the numerous alternative text values that might exist due to variation among the data instances.

```
<alt:element localname="book">
   <alt:attribute localname="publication_date" value="1st Jan 2010"/>
   <alt:attribute localname="revision" value="1.0"/>
   Alice in Wonderland
</alt:element>
```

Figure 7.5 — Text content as a direct child of an instance element

The details of how these are represented are covered in the following sections, but we introduce the notion of declaring text content within an `<alt:value>` element here. By ‘wrapping’ the text with an `<alt:value>` element we remove the problem of ambiguity with multiple text values because each is separated within its own element rather than being siblings of one another (which could be normalized to a single text node, hence the ambiguity). Therefore, all text nodes (and alternative attribute values) that are to be created in the switchable instance are declared as `<alt:value>` elements in the description as shown in the updated example given in figure 7.6.

```
<alt:element localname="book">
   <alt:attribute localname="publication_date" value="1st Jan 2010"/>
   <alt:attribute localname="revision" value="1.0"/>
   <alt:value>Alice in Wonderland</alt:value>
</alt:element>
```

Figure 7.6 — Example instance descriptor for textual content

It is worth noting that the elements described so far (`<alt:element>`, `<alt:attribute>` and `<alt:value>`) closely resemble the XSLT `<xsl:element>`, `<xsl:attribute>` and `<xsl:text>` instructions respectively, both in terms of their syntactic definition and their semantics. This is unsuprising, since the purpose of both groups is very similar — the XSLT elements describe the nodes to be produced as part of the result of the stylesheet's execution, and the instance descriptor elements describe the nodes to be produced when creating the switchable instance tree. Having discussed the elements necessary to describe the fixed parts of the switchable instance definition, we now turn to those used to declare the variable aspects of it.

Alternative Nodes

Unlike the situation involving a standard data instance, the switchable instance must maintain relationships with nodes that are not currently part of the document, as well as those that are. A simple example of such circumstances is shown in figure 7.7 where each data instance contains a different type of element as a child of the `<offer>` element.

```
<data>
   <instance>
      <offer>
         <wine>
            <name>Blossom Hill Chardonnay</name>
            <year>2008</year>
            <price>5.99</price>
         </wine>
      </offer>
   </instance>
   <instance>
      <offer>
         <clothing>
            <name>Men's cotton shirt</name>
            <price>8.99</price>
         </clothing>
      </offer>
   </instance>
   <instance>
      <offer>
         <music type="CD">
            <artist>Stereophonics</artist>
            <name>Word Gets Around</name>
            <price>12.99</price>
         </music>
      </offer>
   </instance>
</data>
```

Figure 7.7 — Example instances containing mutually exclusive elements

The example instances show how there are a variety of alternative offers available to be included in the switchable instance, but also that they are mutually exclusive. Clearly, we wish to include each of these tree fragments within the switchable instance so that each one can be selected by the document author when the document is being edited, but it would be incorrect to include *all* of the alternatives in a single instance.

Therefore, the solution to this problem is the idea of a node that acts as a parent to each of the possible alternatives, and can change the selected node upon request. Although all of the alternative child nodes exist within the switchable instance, only the selected node is made available to any external tool that accesses it, making it appear as a single instance. The example shown in figure 7.7 can therefore be modelled as is in figure 7.8, which shows how the nodes are conceptually related.

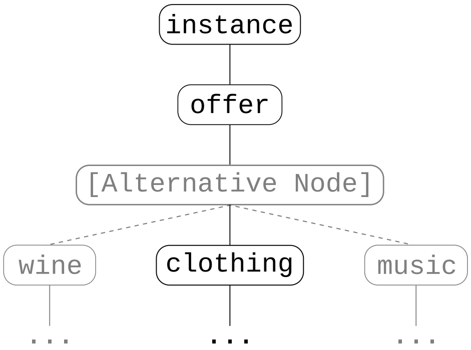


Figure 7.8 — Conceptual relations involving alternative nodes

As with the other aspects of the switchable instance, alternative nodes are declared using a specialist element in the instance description. The `<alt:alternative_node>` element needs no attributes to be specified and can be included wherever needed. Figure 7.9 shows the use of an `<alt:alternative_node>` element when describing the switchable instance created from figure 7.7.

```
<alt:element localname="instance">
   <alt:element localname="offer">
      <alt:alternative_node>
         <alt:element localname="wine">
            <alt:element name="name">
               <alt:value>Blossom Hill Chardonnay</alt:value>
            </alt:element>
            <alt:element name="year">
               <alt:value>2008</alt:value>
            </alt:element>
            <alt:element name="price">
               <alt:value>5.99</alt:value>
            </alt:element>
         </alt:element>
         <alt:element localname="clothing">
            <alt:element localname="name">
               <alt:value>Men's cotton shirt</alt:value>
            </alt:element>
            <alt:element localname="price">
               <alt:value>8.99</alt:value>
            </alt:element>
         </alt:element>
         <alt:element localname="music">
            <alt:attribute localname="type" value="CD"/>
            <alt:element localname="atrist">
               <alt:value>Stereophonics</alt:value>
            </alt:element>
            <alt:element localname="name">
               <alt:value>Word Gets Around</alt:value>
            </alt:element>
            <alt:element localname="price">
               <alt:value>12.99</alt:value>
            </alt:element>
         </alt:element>
      </alt:alternative_node>
   </alt:element>
</alt:element>
```

Figure 7.9 — Example instance descriptor of an alternative node

Although the example presented here involves different elements (and their subtrees) as alternative children to the `<offer>` element, the children of an alternative node need not be elements, but can be any type of XML node. By allowing *all* types of node to be provided as alternatives (even other alternative nodes) we not only prevent unnecessary restrictions, but we also gain a simple solution to the problem of fixed elements with varying textual context. Since text is represented as a distinct type of node in XML, we can easily support alternative textual values by placing each of the alternative text nodes as children to an alternative node. An example of this is shown in figure 7.10, where the element `<foo>` can contain any of the three text nodes as its content.

```
<alt:element localname="foo">
   <alt:alternative_node>
      <alt:value>red</alt:value>
      <alt:value>yellow</alt:value>
      <alt:value>turquoise</alt:value>
   </alt:alternative_node>
</alt:element>
```

Figure 7.10 — Text nodes as children of an alternative node

Alternative Attributes

The final type of variability to be supported is that of attributes with associated values that change from one instance to the next. An alternative attribute behaves like a standard attribute in that it has a name and belongs to a parent element, but it also behaves like an alternative node by supporting a variety of different values. In the same way that an alternative node controls which of the child nodes is included in the current instance, an alternative attribute maintains all of its possible values, but only ever presents the currently selected one. Thus we see that the way in which an alternative attribute is declared within the document description resembles both a normal attribute and an alternative node. Figure 7.11 shows an example declaration of an attribute named `bar` that can take any of the three values shown.

```
<alt:element localname="foo">
   <alt:alternative_attribute localname="bar">
      <alt:value>Monday</alt:value>
      <alt:value>Wednesday</alt:value>
      <alt:value>Saturday</alt:value>
   </alt:alternative_attribute>
</alt:element>
```

Figure 7.11 — Example instance descriptor for an alternative attribute

Instead of the attribute value being declared by a `value` attribute specified on the `<alt:alternative_attribute>` element, a series of `<alt:value>` child elements are specified, with each one declaring an alternative value that the attribute can take.

7.1.4 Metadata Annotations

In addition to the elements and attributes presented so far, we allow for the possiblity of adding metadata relating to the instance data through the use of additional attributes on each of the descriptor elements. The purpose of such metadata is not to aid the generation of, or to add functionality to, the instance produced, but rather it is made available to the editing application, in which the switchable instance is being used, to facilitate the informed selection of alternative data. Obviously, the type information that can be provided through metadata is, in principle, unrestricted, but there has to be a way of collecting the data in the first instance, and the consuming application must be aware of its existance.

The problem of collecting the data depends upon the way in which the switchable instance description is created. If it is constructed by hand, then, clearly, it must also be provided manually. However, the idea behind annontating the instance with metadata comes into its own when we consider the production of the switchable instance from a set of sample data instances. Earlier in the chapter, we discussed the problem that simply collecting many sample instances into a single one can result in a large number of alternative values for each piece of data. We also mentioned how a slimmed-down tree structure, with fewer alternatives, would be more suitable when being used in an editing application. Therefore, if the number of alternatives is to be reduced, they should be selected so as to give a representative indication of the sample data set as a whole. For example, if a set of data instances contain a common address section, which has textual content with a varying length, we might choose to limit the alternatives to the those with the shortest, longest, average and most frequent number of lines[3]. The annotations we advocate would indicate this information along with the relative frequencies of each alternative value. The result of this is that a large number of alternatives can be reduced to a much smaller one, but without losing all of the information contained within the larger set. Figure 7.12 shows an example of fabricated metadata for the address situation described above.

```
<alt:element localname="address">
   <alt:alternative_node>
      <alt:value minimum="yes">
         1 Main St.
         Nottingham
         NG1 2AB
      </alt:value>
      <alt:value frequency="0.4">
         123 Brampton Drive
         Stapleford
         Nottingham
         NG9 7YZ
      </alt:value>
      <alt:value frequency="0.25">
         54 Aldridge Close
         Toton
         Beeston
         Nottingham
         NG9 6MN
      </alt:value>
      <alt:value maximum="yes">
         Room C53
         School of Computer Science
         Jubilee Campus
         University of Nottingham
         Nottingham
         NOTTINGHAMSHIRE
         NG8 1BB
      </alt:value>
   </alt:alternative_node>
</alt:element>
```

Figure 7.12 — Example instance showing annotating metadata

Although we have not discussed the details of how a series of sample data instances might be collated into a single switchable instance, it is clear that some extra

[3] In order to keep the example simple, the address sections are constructed as single blocks of text with embedded newline characters, rather than a series of `<line>` elements as in the earlier example in figure 7.1.

processing during this procedure can generate a large amount of useful metadata. The example presented in figure 7.12 is concerned only with textual variable data where the metadata can be easily calculated. However, there are often circumstances involving other types of data, for example, when the sample data instances include URIs to images, as shown in figure 7.13. In this case, for the metadata to be useful, it must be generated by a tool that is aware of the types of data contained within the sample instances. For variable images, metadata regarding the maximum and minimum bounds, aspect ratios and colour information might be more appropriate than simply providing information relating to the URI string provided. For this reason, the process of producing a correctly annotated switchable instance description from a set of sample data instances can be difficult and, more importantly, situation-specific, and so it is not considered here further. Therefore, for the remainder of this thesis it is assumed that an annotated switchable instance is available for consumption, without further consideration of how it was produced.

```
<data>
   <instance>
      <!-- ... -->
      <image src="http://www.example.com/image.jpg"/>
      <!-- ... -->
   </instance>
   <instance>
      <!-- ... -->
      <image src="foo.jpg"/>
      <!-- ... -->
   </instance>
   <instance>
      <!-- ... -->
      <image src="resources/images/image1.jpg"/>
      <!-- ... -->
   </instance>
</data>
```

Figure 7.13 — Sample instance data containing image URIs

The details of how the metadata can be made available through an editing application, and how such information can be utilized in specific circumstances are discussed in chapter 9. A complete example of a switchable data instance, which exhibits the full range of capabilities discussed, can be found in Appendix B. This example will be the basis for the switchable instance used in later chapters when we consider the execution performance of the modified toolchain.

7.1.5 Implementation of the Switchable Data Instance

So far, we have discussed the model used to represent the switchable data instance, as well as the tagset used to define it, but it is also important to consider how it is implemented and, therefore, how it interacts with the modified version of Saxon described in the previous chapter. Clearly, the standard input tree implementation used by Saxon does not provide support for alternative nodes or attributes with a range of possible values. Therefore a new, custom, implementation is needed to support the model as well as providing functionality to allow for the instance to be 'switched' from one configuration to the next.

The classes used to support the standard fixed elements, attributes and text nodes implement the same interfaces used by the default Saxon implementations, so that our switchable input document can simply be used as a direct replacement without requiring any further modifications to other parts of the processor.

Beyond the basic classes, there must also be classes to provide the functionality required of the alternative nodes and alternative attributes. This is achieved through the `AlternativeNodeImpl` and `AlternativeAttributeImpl` classes, but the implementation of the instance must remain fully consistent with the standard Saxon interfaces, and so explicit knowledge of these must not be necessary when processing the document. Therefore, the classes that implement the externally visible interfaces, which represent the standard document nodes, are written with full knowledge of the variable aspects of the instance structure so that they can operate as expected by external tools, which expect a single immutable instance, without exposing any implementation details. However, for external tools that are aware of the nature of the input data instance, such as the editing application presented later in chapter 9, the various classes also provide access to the alternative nodes and attributes, thus allowing their currently selected alternatives to be changed, as well as any related metadata to be queried.

In order for Saxon to work with this new implementation, a new `Builder` implementation, `AlternativeTreeBuilder`, must also be provided that constructs the switchable instance object structure from the XML-based description presented in the preceding sections. The `Configuration` object used by Saxon to control its operation must also be modified to make it aware of the availability of the new `Builder` implementation.

We have now considered the problem of supporting changes to the input data instance, and have presented the idea of a switchable instance that can be directly used within Saxon, whilst also allowing compatible tools to effect the changes needed. We now turn our attention to the issues surrounding edits that are made to the XSLT stylesheet itself.

7.2 Changes to the Transform

In the previous section we considered the process of changing the input data instance and the mechanisms needed to support these changes. The basis of such changes stem from the need for the document author to view different result instances that might be generated, in order to see the effect of the variable data on the document as a whole. However, this type of edit is a specialist one that results from the variable nature of the document and there remains the need to make more 'traditional' edits, such as adding components to the document or changing the properties of existing components. These edits require changes to be made to the underlying XSLT stylesheet and it is the way in which these changes are made that we now consider.

7.2.1 Modifying the Stylesheet DOM

All edits that can be made to the XSLT stylesheet can be broken down into a series of actions, each of which can be assigned to one of three categories: adding new instructions, editing existing instructions, or removing existing instructions. In fact, these categories can be further reduced to just two if we treat the editing of an existing instruction as a compound action of removing the instruction and replacing it with a new instruction that exhibits the necessary changes.

Since the XSLT stylesheet is represented as a DOM-like structure within Saxon, and therefore supports direct manipulation along the same lines as a standard DOM, the process of modifying the `StyleElement`-based representation of the stylesheet is relatively straightforward. Removing instructions or expressions from the tree can be achieved by removing the corresponding `StyleElement` objects from their parent node. This can be effected by calling a method on the parent `StyleElement` object, with a reference to an instruction/expression, that requests that the child `StyleElement` object be removed. Equally, when adding a new instruction to the stylesheet tree, a method an be called on the parent `StyleElement` object requesting that the instruction be added as one of its children. Clearly, the new `StyleElement` object must first be created before it can be added, but this is a simple extra task for the `StyleElementFactory` class, which is used during the initial constrcution of the stylesheet.

The process of making a change to an existing `StyleElement` object can be handled by firstly removing the existing `StyleElement` object and then replacing it with a newly constructed one. This approach reduces the number of distinct editing tasks to be supported, while also allowing all edits to be handled through common procedures. There is, however, an extra processing stage that must be performed when dealing with the editing of existing `StyleElement` objects, which is illustrated in figure 7.14.

```
<xsl:template match="/">
   <xsl:for-each select="1 to 10">
      <svg:text x="0" y="{10 * position()}">
         <xsl:text>Line:</xsl:text>
         <xsl:value-of select="."/>
      </svg:text>
   </xsl:for-each>
</xsl:template>
```

Figure 7.14 — `StyleElement` replacement example

The result of the example code is to print out the strings “Line: 1” through to “Line: 10” beneath one another, but let us consider the changes required if we wanted to alter the number of lines that are printed. Clearly, this can be achieved by modifying the `select` attribute placed upon the `<xsl:for-each>` instruction. However, by breaking this edit down into the removal of the affected `StyleElement` object, followed by the addition of a new, modified, one, we encounter a problem. By removing the original `StyleElement` object, we also remove the child nodes attached to it, and so when we add the newly created `StyleElement` object, we must also transfer the child `StyleElement` objects of the original node to the new one.

This process is depicted in figure 7.15. Once again, this can be achieved very simply by using the standard manipulation methods provided by the `StyleElement` class, but it is essential to the integrity of the stylesheet that this is done.

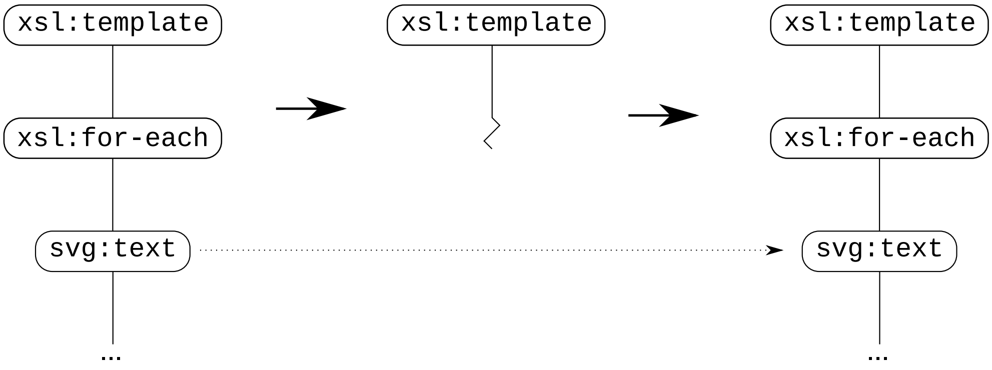


Figure 7.15 — Transfer of child `StyleElement` objects during edit

7.2.2 Reflecting Changes in the Transform Executable

Making the necessary changes to the `StyleElement` representation of the stylesheet to reflect an edit to the document is relatively simple, but these changes alone are not sufficient for the edit to have any effect. In chapter 5, we discussed the way in which Saxon compiles this `StyleElement` tree into an executable form of the stylesheet. Therefore, the changes made to the `StyleElement` representation must be reflected in the compiled form if the edit is to be realised when the stylesheet is re-evaluated.

The naïve approach to this problem is to make the necessary changes to the `StyleElement` representation and then to recompile the entire stylesheet, as would be done when the stylesheet is first loaded. This is a bad approach for two reasons:

- It is inefficient, because those parts of the stylesheet that have not been changed are recompiled unnecessarily.
- It renders all of the stored state information useless since it is linked to existing `Instruction`s that will be replaced by the recompilation.

The first of these points is self-evident, but the second point requires further explanation. The previous chapter, explained how each `ProcessorState` object is referenced by the relevant result DOM element, alongside a reference to the `Instruction` that was executed when the result element was generated. If the entire stylesheet is recompiled, *all* of the `StyleElement` elements produce a new `Instruction` object, which are then combined to create a new executable form of the stylesheet. The result of this is that the internal configuration of the processor is updated to reference the newly created executable form, and yet the numerous `ProcessorState` objects, which were created during the initial processing of the stylesheet, still reference the *old* executable form. As we shall see in chapter 8, the automatic reprocessing mechanism that is employed relies on checking whether or not the generating `Instruction` is still referenced within the executable, and re-evaluating the relevant parts of the stylesheet if it is not. Clearly, if we were to completely recompile the stylesheet after every edit, this would not be possible.

Given that complete recompilation of the stylesheet is both inefficient and problematic, we must effect the changes to the existing executable form in a much more targetted way. Since we create the new `StyleElement` objects that we then add to the DOM-like representation of the stylesheet, we can also individually compile them without having to go through the full compilation procedure that is instigated from the root node of the stylesheet. This allows us to genertate the new

`Instruction` objects individually, but simply calling the `compile(...)` method on a `StyleElement` object is only part of the procedure that must be performed. Firstly, the old `Instruction` object that is part of the `Executable` must be removed so that it can be replaced (or simply removed if that is the edit to be effected). Once this has been done, or if we are just adding a new element to the stylesheet, the new `Instruction` object must be incorporated into the `Executable`.

Removing Existing Expressions

In the majority of cases, removing an existing `Expression` object (`Expression` being the superclass of `Instruction`) can be achieved through a single call to the parent `Instruction` requesting that the `Expression` in question be removed as its child. However, in other cases the procedure is much more involved. For example, if an `XSLTemplate` node (the subclass of `StyleElement` that represents an `<xsl:template>` element) is removed, any changes made to the `RuleManager` object used by the stylesheet must be reversed. This is a much more complex process in which any `Pattern` rules that were added when the `XSLTemplate` object was first compiled must be removed, as well as any related changes that affect the operation of the `RuleManager` as discussed in chapter 5.

The variety of actions perfomed when a `StyleElement` is first compiled is necessarily diverse, and the variety of procedures required to reverse this compilation is equally so. Therefore, a new method, `uncompile(...)`, is added to the abstract `StyleElement` class such that it an be called to reverse the effects of the initial compilation and effectively remove the `Expression` from the `Executable`. Because the actions required to achieve this are dependent upon the `StyleElement` in question, the `uncompile(...)` method is not implemented in the `StyleElement` class, but rather as specialised implementations in the various subclasses.

Adding New Expressions

The process of adding a new `Expression` object to the stylesheet is relatively simple. A call made to the `compile(...)` method of a `StyleElement` object will either return a corresponding `Expression` object, or will make direct modifications to the executable (as is the case with `XSLTemplate`). Therefore, in some circumstances, the call to `compile(..)` is all that is necessary to effect the required changes to the `Executable`. However, when an `Expression` object is returned, it must be added as a child of the relevant parent `Expression` object. The way in which this is achieved is dependent upon the specific subclass of `Expression` in question, but is ultimately a simple procedure.

Having considered the implications of how edits are made to both the input data instance and the various stylesheet representations in this chapter, as well as having considered the issues surrounding how state information is stored during execution of the stylesheet in the previous chapter, we now turn to the problem of incorporating these into a partial re-evaluation mechanism.

Chapter 8:

Working with Stored State

In the previous chapters, we have introduced the idea of recording the state of the processor during its initial execution so that, at a later point, it can be restored and the processing repeated. The exact details of the information that is needed, how it can be obtained, and what format it might be stored in, have been dicussed in chapter 6. We now look at the ways in which this information is used to restore the execution state so that, after the types of edits discussed in chapter 7 have been performed, we can subsequently partially re-evaluate the transformation script to produce an updated result document. We also discuss a strategy that can be used to perform the necessary re-evaluation automatically.

8.1 Partial Re-evaluation

Re-evaluation of a document by using stored execution state depends on the mechanism employed to gather and store that state in the first instance. As was discussed in chapter 5, two approaches to this problem were proposed — a pre-processing XSLT script to augment the original stylesheet, and modifications made directly to the evaluating XSLT processor — but the latter was chosen as the preferred choice for a number of reasons. As well as the problems previously discussed, there is also the difficulty of devising a viable re-evaluation strategy. The process of stylesheet augmentation provides us with a method, albeit imperfect, of gathering and storing the execution state as the elements in the result document are produced. Once the initial processing pass has been completed, we are left with a modified stylesheet and a result document annotated with state information. For a particular piece of the original stylesheet to be re-evaluated, we must perform a number of steps.

Firstly, the node at the root of the subtree requiring re-evaluation must be identified. This is a simple process, since the path to that node is part of the state information stored. However, there is a significant challenge in isolating the subtree and re-evaluating it in the processing context in which it was originally evaluated. In order to maintain the premise of making any required alterations, to the pre- or post-processing stages, at the XSLT level, a number of potentially complex transfomations must be completed through a series of newly created XSLT stylesheets. Things are further complicated when we consider how the execution state might be recreated and how the parts of the stylesheet requiring re-evaluation are identified.

Some aspects of the execution state, such as the current mode, are easy to determine during re-evaluation, but others, implicitly maintained by the underlying processor, are much more problematic. As an example of this, consider the problem of setting the current context position, as would be returned by the `position()` XPath function. Re-evaluating specific instructions in an identical execution state requires that this implicit ‘variable’ is reset to the correct value, but an XSLT-based solution would require extra code to achieve this, thus adding to the cost and complexity of re-evaluation.

The problem of identifying the instructions that require re-evaluation once an edit has been made is another area where an XSLT-only solution is not well suited. Consider the simple code example shown in figure 8.1 where the text content of an input data element is stored in a variable and then later referenced.

```
<!-- ... -->
<xsl:variable name="foo">
   <xsl:value-of select="/bar"/>
</xsl:variable>
<svg:text>
   <xsl:value-of select="$foo"/>
</svg:text>
<!-- ... -->
```

Figure 8.1 — Re-evaluation caused by an edit to a referenced variable

If the input data instance is edited such that the text content of the `bar` element in question is changed, the `<svg:text>` element must be re-evaluated because it utilises the variable, `foo`, that is affected by the edit. In order to indentify this re-evaluation requirement, via another XSLT stylesheet, a complex analysis of the original stylesheet must be performed in which all expressions are parsed and checked for references to the variable in question. This process, as with the others described, is ultimately computationally expensive and, along with the other problems discussed in chapter 5, renders a solution based on stylesheet augmentation to the problem unsuitable.

The alternative approach of making modifications to the underlying processor can provide solutions to these problems, and it is these solutions that we now discuss.

8.2 Re-initializing the Processor

As discussed in chapter 6, all the pieces of data regarding the execution state at a given time are stored within a `ProcessorState` object. In order to re-initialize the process to this state, we must reset the various variables and constructs within the processor to the values and/or references stored within the `ProcessorState` object. In contrast to the problems of the XSLT-based solution, the procedure is much more simple when we have direct access to the internal objects and data structures within the processor. In a similar way to that used to store the state in the first instance, *restoring* the state can be achieved through some simple modifications to the subclasses of `XPathContext`. Clearly, much of the required functionality already exists since it is needed when setting and updating values during the normal processing of the stylesheet. However, methods for setting or updating some of the implicit values, such as the context position discussed in the previous section, are not provided by default within Saxon because, under normal operation, such changes are handled internally. Therefore, changes have been made to allow for the setting and/or updating of such values as necessary. This is made possible by the fact that the data stored within the `ProcessorState` objects is the same data used in the initial processing. In the case of primitive values, the stored values are simply copied, but more complex objects are retained by storing references to them within the `ProcessorState` object. Thus, setting them within the new `XPathContext` object is achieved either by simply copying a primitive value or by replacing an internal reference with a stored reference and is therefore computationally inexpensive.

In conjunction with the minor changes made to the `XPathContext`-based classes, the procedure used to restore the execution state utilises existing functionality within Saxon, which supports the creation of a new XPathContext object. The buildXPathContext() method contained with the ProcessorState class creates a new XPathContext object and uses the newly added, and previously existing, methods to set all values and references held by the object to those stored within the ProcessorState object. The result of this call to the buildXPathContext(...) method is an XPathContext object that is in an identical state that of the XPathContext object used during the initial processing.

We recall from chapter 5 that the processing of instructions is performed through calls to either the process(...) or processLeavingTail(...) methods (depending on the Instruction or Expression in question) and that both of these methods take a reference to an XPathContext object as a parameter. It is this object that is used to access the dynamic context in which the Instruction or Expression should be evaluated. Therefore, re-evaluation of a given instruction can easily be achieved by calling the correct method on the appropriate Instruction object, while passing it a reference to the newly constructed XPathContext object.

8.3 Capturing the Generated Tree Fragment

Although the re-evaluation of individual instructions can be performed as described, the nodes produced as a result of such re-evaluation need to be captured and used to replace the corresponding nodes in the original result document. In general, the destination for all result nodes that are produced is set before the initial execution of the stylesheet begins. Thereafter, all appropriate method calls, such as those emitted from the various subclasses of ElementCreator, are sent to the Receiver object that was registered within the processor. Clearly, the correct tree structure of the result document is achieved because of the order in which these calls are made, and since we are not reproducing the whole tree (and thus not making all the necessary calls), the nodes produced due to the re-evaluation process will not automatically replace the required nodes.

Instead, we provide the processor with a new destination for the result document fragment that is to be produced by the partial re-evaluation, and we perform the replacement in the original result document manually once the re-evaluation has been completed. The first stage in this process is to create a new DOM `Document` to which the newly created result nodes can be added and to wrap it within a `DOMDestination` object that implements the `Destination` interface expected by the `Controller` object, which is responsible for managing the execution of the stylesheet. This object, along with references to the instruction to be re-evaluated and the corresponding `ProcessorState` object, is then passed to a newly created method within the `Controller` class that performs the partial re-evaluation. This new method, `transformFragment(...)`, handles the configuration of the processor to use the new destination when creating result document nodes; it also constructs a new `XPathContext` object as described in the previous section, and begins the partial re-evaluation of the selected instruction.

Once the re-evaluation has been completed, the DOM `Document` contained within the `DOMDestination` provided, has the generated tree fragment as a child of its document node. Therefore, the object from which this entire process was instigated now has a reference to the DOM `Document` containing the new tree fragment, as well as a reference to the original result document. With access to both these structures, it is a trivial operation to remove the old nodes from the result document and replace them with those generated as a result of the partial re-evaluation. This process is depicted in figure 8.2.

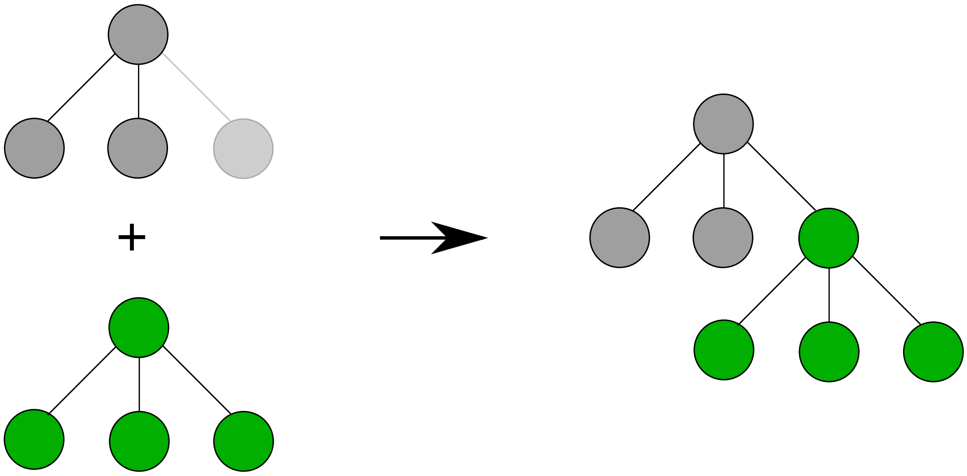


Figure 8.2 — Tree fragment replacement after partial re-evaluation

8.4 Automatic Re-Evaluation of Document Components

So far, we have discussed the ways in which processor state can be saved, restored, and instructions re-evaluated with their new results being reflected in the complete result document. However, the procedure in which this process has been performed is a manual one and it is not a practical approach if we consider the full implications of the potential re-evaluation needed when an edit is made. Consider a complex document where a piece of data in the input instance represents a person's address, as might be the case in a typical advertising leaflet. The view of the document seen by the author is the result of the execution of the XSLT stylesheet with the input data, and so any interaction with the document, in terms of editing, must be performed through that result instance. Also, let us assume that, for whatever reason, the recipient's address is required to be included in more than one place within the document. If the author requests a change to the input data instance as described in chapter 7, so that he/she can view the effect of a different data instance, such a request would be initiated through interaction with one of the text components that references the address data. Clearly, it is a trivial task to identify the instruction associated with the selected component as being one in need of re-evaluation (since a reference to the corresponding `StyleNode` object is stored on the result document element). It is also the case that the other component(s) that reference the address data require re-evaluation. Such a situation would require the author to select each and every component that depends on the address data *in any way* and initiate the re-evaluation of the relevant instruction(s). This becomes further complicated when we consider inter-component dependencies (e.g. component A should be the same width as component B), together with the issues surrounding the use of variables by various components within the XSLT stylesheet, and changes to the XSLT stylesheet itself.

Clearly a mechanism is needed for automatically performing the necessary re-evaluation, as and when required. The two sources used to create the result document instance, which is displayed to the user, are the input data instance and the XSLT stylesheet. Any edits made by the author must be reflected in one or the other of these, and it is only changes to these structures that have any effect on the result instance. Therefore, any mechanism to initiate re-evaluation of the document as a reaction to such edits, must be aware of the fact that they have occurred, as well as having a method of identifying the parts of the stylesheet affected.

Chapter 6 briefly mentioned the need for further data to be stored alongside the constituent pieces of state information in order to support such a re-evaluation mechanism. Chapter 7 also alluded to a similar situation in which information relating to the recompilation of `StyleNode` objects should be maintained to support automatic re-evaluation. We now discuss the extra functionality needed in both of these cases.

8.4.1 Recording Input Data Instance Usage

In summary, a change to the input data instance can result in a number of separate parts of the stylesheet requiring re-evaluation, each possibly in a number of different execution states. Therefore, when indentifying the parts of the result document that are affected by the change to the input data, we must have some knowledge of which parts of the input data instance were used in the process of generating the result document elements. This problem can be approached in one of two ways — either analyse the stylesheet to deduce which parts of the input data instance are used at each point throughout its execution, or simply record the usage as the stylesheet is initially executed. The difference in these approaches mirrors that between the different methods of storing the execution state that were discussed in chapter 6, and similar reasoning as to why we might choose the latter solution applies here also. Analysis of the entire stylesheet after every edit to the input data instance is expensive in the extreme, and a complete analysis may require a simulated execution of the entire stylesheet. Therefore, we favour the latter approach of recording the usage of the input data instance during the execution of the stylesheet. This can be achieved in two separate ways; a list of result elements can be recorded in a dictionary-like structure alongside each input data node that was used in their production, or, alternatively, a list of the input data nodes used in the production of each result element can be stored alongside each element. As we shall see in later sections, aspects of both these approaches are combined in the final solution that is presented, but the actual process of recording the use of input data nodes utilizes the latter approach. Therefore, whenever a node in the input data tree is accessed, this fact is recorded in the current ProcessorState object, which is eventually stored alongside an element in the result document tree. Such a solution is convenient since this mechanism for storing information within the result elements (the ProcessorState objects) is already implemented, as described in chapter 6.

The question now remains of ascertaining when individual nodes in the input data instance are referenced from the XSLT stylesheet. A simple solution would be to augment the Expression objects within the compiled form of the stylesheet such that they report when they have accessed particular input data nodes. However, this becomes increasingly difficult when we consider the range of permitted XPath expressions that may utilize different parts of the data instance. Simple node selection XPath expressions such as the select expression shown in figure 8.3 are represented within Saxon by a series of embedded objects of a particular sublcasses of Expression.

```
<xsl:template match="foo">
   <!-- ... -->
</xsl:template>
```

Figure 8.3 — Simple node selection XPath expression

This subclass can be modified to emit a notification that it has been evaluated, and hence that a particular node in the input data instance has been used, during execution of the stylesheet, but when we consider other XPath expressions, in particular those utilizing functions, things become more complicated. The number of modifications required to support such a mechanism within the various expression types and functions supported by the XPath classes, as exemplified by the `match` expression shown in figure 8.4[1], forces us to consider an alternative solution.

```
<xsl:template match="//*[count(starts-with(./text(), 'The')) gt 1]">
   <!-- ... -->
</xsl:template>
```

Figure 8.4 — Complex XPath expression utilizing functions

There are two 'actors' in the execution process — the `Instructions` and `Expressions` within the compiled form of the stylesheet and the input data instance. We have just discovered that the necessary modifications to the compiled `Expressions` are numerous and ultimately impractical, and so we turn to the possibility of making modifications to the input data instance.

We recall from chapter 7 that the input data instance utilises the *switchable* implementation, thereby maintaining each allowable node in the instance within its structure. Each node is implemented as an object in its own right in contrast to the `TinyTree` implementation offered as the default by Saxon. Therefore, if a node in the data instance is accessed or operated on, a corresponding method will be called on the object that represents it. This provides an ideal point from which we can notify any interested objects of the fact that a particular node has been used in some way. When executing the stylesheet, any such use of the data instance indicates that it is the subject of the `Instruction` or `Expression` being evaluated and so any changes to the node could result in the `Instruction` or `Expression` requiring re-evaluation.

[1] For completeness, the `match` expression shown selects all elements within the document that have at least one text node child whose content starts with the string 'The'

The notification that a node has been accessed is achieved by adding a call to an external entity from each of the methods supported by the various data node implementation classes. Since we already have a external entity (the `ProcessorState` class), that is notified of changes to the internal state of the processor, it is only sensible that we extend its functionality to include the ability to record references to the input data nodes as and when they are used. Therefore, the `ProcessorState` objects referenced by the result document elements contain all the necessary information to determine whether or not they require re-evaluation when the input data instance is edited.

8.4.2 Compiled Expression Versioning

The other aspect of the document that can be edited is the underlying XSLT stylesheet, and the ways in which such editing is supported has been previously discussed in chapter 7. There is, however, an important point to make regarding the process of replacing `StyleElement` objects in the stylesheet representation and their subsequent reflection in the compiled executable form. Chapter 6 explained how a reference to the relevant `StyleElement` object is stored alongside the appropriate `ProcessorState` object within the annotated result elements, but there has been no discussion of the implications of the re-compilation or replacement of `StyleElement`s on this stored reference. Clearly, if a `StyleElement` object is re-compiled or replaced, any result element that was produced as a consequence of the execution of its old compiled `Instruction` should be either re-evaluated or removed completely. However, the reference to the old `StyleElement`, which was initially stored on the result element, remains unchanged and only a process of checking and updating every element can rectify this.

In order to support this checking procedure, which we discuss further later in the chapter, we provide a method of versioning `Instruction`s that are produced by their respective `StyleElement`s. The crux of the problem is that there can be stored references to `StyleElement`s that are either no longer part of the stylesheet, or have produced multiple `Instruction` objects as a consequence of their re-compilation. Therefore, the process of removing a `StyleElement` object from the stylesheet is modified to indicate that the `Instruction` object it currently references should not be used. This is achieved through the use of a versioning variable within the `StyleElement` object that can be queried by other objects. As part of the initial compilation of a `StyleElement`, the version variable is set to the value of `0`, thus indicating that it is the first `Instruction` produced and referenced by this object. Subsequent compilations of the *same* `StyleElement` increase the value of this variable to indicate the increased number of `Instructions` produced, and the removal of a `StyleElement` from the stylesheet is indicated by setting the versioning variable to the value of `-1`.

Therefore, when the stylesheet is executed, the version of the compiled object referenced by the `StyleElement`, which is stored within the annotated result element, is also stored within that element. By having this information, as well as a reference to the `StyleElement` object, it is possible to compare the version used during the previous evaluation with the current version of the `Instruction`, thus determining if the node requires re-evaluation because of a change to the stylesheet.

8.4.3 Variables and Parameters

A further complication to the re-evaluation process is introduced when we consider the role of variables and parameters in the stylesheet. In situations where instructions make use of previously declared variables or parameters, they are clearly dependent upon their values. Therefore, it is important that any edits made to the definition of these variables/parameters is noticed by the instructions that reference them. To support this propagation, the `ProcessorState` class is further extended to allow for the storage references to variable and parameter definitions as they are utilized during the execution of the stylesheet.

It is also necessary to check for affected variable definitions when making edits to the stylesheet. Let us consider the affect on an `<xsl:variable>` instruction when one or more of its descendent nodes is edited as shown in the example code given in figure 8.5.

```
<xsl:variable name="foo">
   <xsl:text>bar</xsl:text>
</xsl:variable>
<svg:text>
   <xsl:value-of select="$foo"/>
</svg:text>
```

Figure 8.5 — Propagation of an edit to a variable

Let us assume that the textual content of the variable ‘`foo`’ is changed from the value ‘`bar`’ to some other text string, thus causing the following `<svg:text>` element to require re-evaluation. The previous section has discussed how the version number of an `Instruction` is updated when its associated `StyleElement` object is recompiled, and it is this same mechanism that we employ here. When the edit is made to the `<xsl:text>` element, its ancestors must be searched to determine whether or not it is a child element of an `<xsl:variable>` element. If such an ancestor element is found, it is flagged as being outdated by increasing its compilation version.

8.4.4 Identifying the Need for Re-evaluation

The previous sections have discussed the ways in which we have augmented the state storage mechanisms to support the storage of other information for determining whether partial re-evaluation of the stylesheet is required. We now consider how, and where, the need for partial re-evaluation is determined and subsequently initiated.

A Self-Regenerating DOM

Rather than producing a series of external mechanisms designed to check for any nodes in the result document that reference obsolete `StyleElements`, or have `ProcessorState` objects that describe values that have been altered or updated, we introduce the idea of the result nodes *themselves* initiating any necessary re-evaulation.

Although this may seem an odd solution at first, it is ultimately one that yields further reductions in potential re-processing costs. By making the result elements responsible for initiating any partial re-evaluation that may be necessary, this will only occur when they perform the checking procedure. In turn, this check must be performed when the node is accessed so that it can respond in the correct way with regard to the current state of the stylesheet and input data instance. The advantage to this approach, as opposed to eagerly performing any re-evaluation of the stylesheet directly as part of the editing process, is that the re-evaluation is only performed as and when it is required. In terms of the result document, this requirement stems from its consumption by another entity. In the case presented here, this entity would be the editing application, which would access the result document in order to draw it and display the result to the user. Therefore, if an edit made to the document by the author affects elements in the result instance that are not currently being displayed by the editing application, their re-evaluation might not be performed until they are displayed. In cases where multiple edits are made that affect such an element, the intermediate states will never be computed because they were never requested. In large, multi-page documents, this has the potential to offer large savings in terms of processing cost.

Update Notifications

As we have already discussed, calls to the `ProcessorState` class are made whenever parts of the input data tree or stylesheet variables/parameters are used. References to these objects are then stored so as to associate the stored execution state with the parts of the document used. A simple solution to the problem of identifying outdated result elements would be to check the status of these stored references and, if they have been altered by an edit to the document, initiate the necessary re-evaluation. However, this repeated 'polling' of data regarding the status of the stored objects is an expensive one. When we consider that each of the elements in the result document will retrieve and check this information every time they are accessed, the costs can become considerable. Therefore, a process of *notification*, as opposed to polling, is presented as a more efficient solution.

The way in which this is realised has already been alluded to earlier in the chapter when we considered the process by which the usage of the input data tree is recorded. This recording process associates every result element with the input data nodes, and stylesheet variables/parameters, that were used during its creation. In order to be able to notify these elements of changes to these dependencies, relationships in the opposite direction must also be maintained. Therefore, each time a result element is created, it is registered with the input data nodes and stylesheet variables/ parameters that were stored in the corresponding `ProcessorState` object. Thus, whenever an edit is effected on the stylesheet or input data tree, any result elements that were registered with the object in question can be notified of the change. This is implemented by simply setting a boolean flag on the result element, which is then checked as part of its procedure for determining if any re-evaluation is necessary. By following this process, only the affected result elements are notified of an edit, and unnecessary polling checks are not needed on every method call to the result elements.

The notification process itself is done through a simple method call to the relevant result element. Depending upon on the nature of the edit, this method call is made from either the input data node object, or a stylesheet `StyleElement` object. In the case of a change being made to the input data instance, the notification is emitted from the method used to select an alternative node/value in the switchable data instance. In the case of changes to variables or parameters within the stylesheet, the notification is sent when the relevant `StyleElement` object is compiled to create a new `Instruction` object.

Variable/Parameter Re-evaluation

When a result element is re-evaluated because of a change to the input data tree, the relevant instruction can simply be re-executed with a rebuilt `XPathContext` object to generate a replacement element. This is possible because the input data nodes are accessed as the instruction is executed, and so any changes are reflected during this execution. The situation involving variables and parameters is, unfortunately, not as straightforward. Variables and parameters are declared and defined by one instruction, but then accessed via other instructions. Therefore, whenever an `Instruction` is executed, the values of these variables/parameters have already been evaluated and stored within the current `XPathContext` object. Clearly, in situations where re-evaluation is required because of changes to a variable or parameter declaration, simply re-evaluating the relevant instruction with a rebuilt `XPathContext` object will not yield an updated result element, but rather one identical to the original. The reason for this is simple — although the result element has been notified of a change to the variable/parameter definition, this change has not been propagated to the calculated values used during execution of the instruction(s).

The problem can be understood more easily by considering the true nature of a variable or parameter. In reality, they are produced as the result of executing an `Instruction` in the same way that result elements are generated by element creation `Instructions`. Therefore, a re-evaluation mechanism similar to that implemented for result elements can be used to automatically re-evaluate variables and parameters. Because all parameters and variables are set in, and retrieved from, the `XPathContext` object, we can perform a check as to whether or not they require re-evaluating before returning the correct value. To be able to perform these checks, information regarding the variable/parameter compilation version is required. Therfore, the process through which variables/parameters are set is modified such that references to the appropriate declaration `StyleElement` and current compilation versions are stored. By checking these values when a variable/

parameter value is requested, the variable/parameter can be re-evaluated if necessary and the correct values returned.

Two-Tier Result Nodes

In order to allow the replacement of result tree nodes with newly generated document fragments 'on-the-fly' the implementation of the result document nodes must be modified. The modification in question entails the division of each node into two parts — the actual node and a handle to that node in which the re-evaluation check is performed. Therefore, for each of the different node classes supported as part of our custom result DOM implementation, we also produce a corresponding handle class. The names of these classes mirror the name of the object that they are a handle to, with the string 'Handle' appended to the node class name (e.g. `ElementHandle` is the associated handle class for the `ElementImpl` class).

Each handle class implements the same interfaces supported by the relevant node class and can therefore be referenced directly by anything that expects a standard DOM interface (such as the internal workings of Saxon). The methods supported by the handle classes are identical to those supported by their corresponding node classes, with each of the incoming method calls being passed on to the actual node object referenced by the handle. The process by which a node is accessed, and subsequently checked for need for re-evaluation is as summarised in figure 8.6.

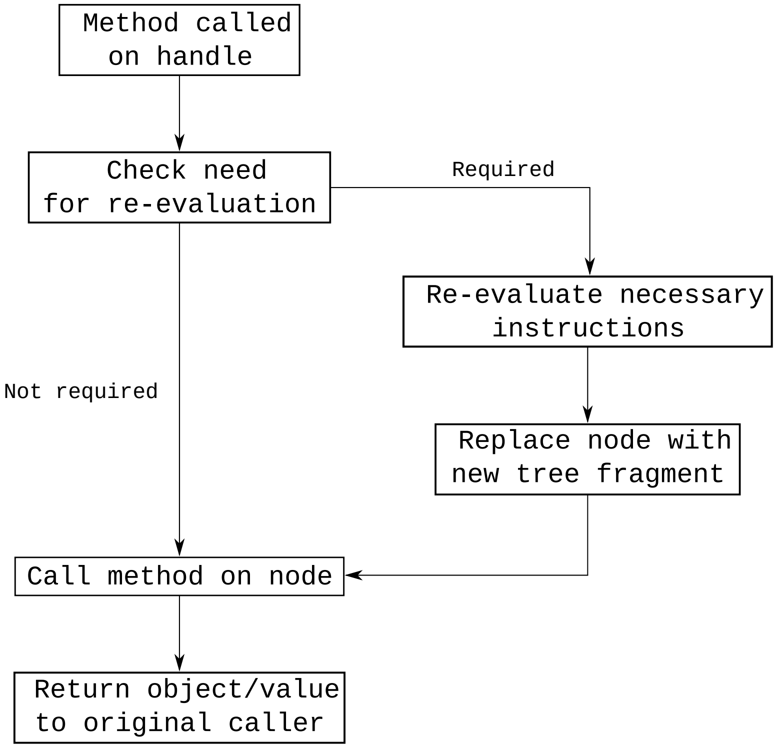

Method called on handle
Check need for re-evaluation
Required
Re-evaluate necessary instructions
Not required
Replace node with new tree fragment
Call method on node
Return object/value to original caller

Figure 8.6 — Flowchart showing re-evaluation identification process

The first stage of the process is for a method to be called on the node's handle, for example, querying the name or type of the node. The handle then checks whether or not it has been notified that data on which it is dependent has been changed. If re-evaluation is necessary, the handle requests this be performed through a call to the necessary method on the underlying `Controller` object and retrieves the newly generated result document fragment. The old node is then replaced by the root node of the new tree fragment and a method call matching that which was originally called on the handle is called on the new node. Any object or values returned by the node are finally returned by the handle's method. From the viewpoint of the object that made the original method call on the handle, the handle has behaved exactly as a 'standard' node would have and the update check and (possible) re-evaluation has been performed trasparently so that the method is called on the up-to-date result node.

Handle and Node Relationships

To complete the discussion of the result DOM structure, we must discuss the relationships maintained between the nodes and their handles. It is clear that each handle holds an internal reference to the node that it 'shadows', however it is also important that the actual node objects are never made available through the public interface. If they were then the update checks will not be performed and the handle mechanism might become inconsistent and corrupted. Therefore, the internal implementations of the various node classes must be modified such that they are aware of the existence of the handles and only return references to the relevant handles as opposed to references to the actual nodes. This is particularly the case when we consider methods supported by the nodes that provide access to parent and child nodes. Clearly, these must return references to the handles rather than to the nodes themselves and this is exactly the behaviour exhibited by the modified node classes. In summary, all references that are returned are references to handles, *not* actual nodes, but the inter-node relationships are maintained by the nodes rather than the handles. A diagram showing the abstract parent-child relationships between two nodes is given in figure 8.7.

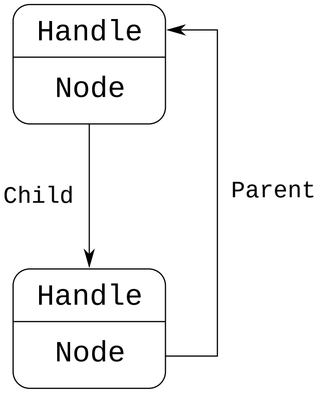


Figure 8.7 — Abstract node/handle relationships

Summary

This chapter has discussed the mechanisms employed to automatically identify and perform the necessary re-evaluation of the document. This is achieved by utilizing the information that is stored as a result of the extra functionality described in chapter 6 and the editing processes and supporting structures detailed in chapter 7. We now turn our attention to incorporating the methodologies and modified tools previously discussed into a working WYSIWYG document editor that can be used in the performance analysis of such an editing framework.

Chapter 9:

Editor Integration

The preceding chapters have shown how partial document re-evaluation can be achieved and how a processing framework can be constructed to support interactive editing of the underlying stylesheet and its input data document. In this chapter we consider how this framework can be incorporated into a WYSIWYG editor, as well as the implications of such integration on the construction of the document, and interaction with it.

9.1 Building a Usable Editor

An important aspect of a WYSIWYG editor is that of displaying the document to the user. Since our example document workflow culminates in the creation of an SVG document, we must provide support for the direct rendering of SVG. One possible option is to design and write a bespoke SVG rendering engine. However, this is a large undertaking and one that has already been tackled by many people. There are a number of different rendering engines, but it was decided to use the open source

Batik [61] suite produced by the Apache Foundation. This selection was motivated by the following reasons:

- It is open source and so any source code changes that need to be made can easily be incorporated
- It is written in Java and so integration with the other Java-based tools is relatively straightforward
- It is widely recognized as having good support for the SVG standards with good supporting documentation

As part of the libraries supplied with Batik there exists a Swing[1] component, `JSVGCanvas` that can directly handle the rendering of an SVG document around which an editor can be constructed. Therefore, the current document instance created through interaction with other parts of the editor (via edits and subsequent processing passes performed by our modified Saxon processing framework) can be passed to this component to be rendered and then displayed to the user.

Although the `JSVGCanvas` faithfully renders the document provided to it, the internal workings of the Batik libraries operate on a specific SVG DOM implementation rather than any generic DOM implementation. This means that the first stage of rendering the document is to produce a clone of it using the internal SVG DOM classes, which can then be rendered. As we see in the next section, this can cause problems when adding functionality for user interactions with the document since all interactions are performed with the cloned DOM and not the annotated one produced by our modified toolchain. One solution to this problem is to change the output DOM implementation used by the modified version of

[1] Swing is the default widget toolkit provided as part of Sun Microsystems' Java Foundation Classes (JFC)

Saxon so that it uses the Batik classes to produce a compatible DOM. Unfortunately this leads to further problems. Firstly, the Batik DOM classes would need further augmentation to allow for the annotations and `ProcessorState` references to be stored on them, thus requiring a large amount of work to be done within the Batik libraries themselves. Secondly, since the premise of partial re-evaluation of the document is not dependent upon an SVG-based workflow, it seems undesirable to alter the XSLT processor to support only a specific XML tagset.

The alternative approach is to make modifications to Batik such that it can operate on a generic DOM implementation. However, there are technical reasons why Batik works with a specific DOM implementation and attempting to make changes to alter this behaviour is impractical without attempting to rewrite large parts of its functionality. Therefore, the Batik libraries are used in their unmodified form and all interactions with the cloned document are translated to the original result DOM by simply traversing to the corresponding node in the original result DOM. Obviously this approach is not optimal, most notably because the result DOM is cloned every time the `JSVGCanvas` needs to be repainted, however this setup is simply a test demonstration of the workflow methodology presented and therefore such a compromise was deemed acceptable.

9.2 Document Interaction

As well as simply rendering the supplied document, the Batik libraries offer a variety of extensions that allow for interaction with the document through mouse and keyboard events generated by the `JSVGCanvas` object. A series of default `Interactors` and `Overlays` are provided to handle common tasks such as zooming and translating the document, but these can be removed and custom implementations added if needed.

By adding our own custom `Interactor` implementations, we can handle events generated by user interaction with the document instance and react accordingly. For example, when the user selects an SVG component we can retrieve references to any variable input data used in the creation of that component and provide the user with the option to choose an alternative value in the switchable input DOM. The use of custom interactors is also essential in supporting the click-and-drag editing support typically expected of WYSIWYG editing applications.

9.3 Abstract Edit Decomposition

As well as supporting component selection, and changes to the input data instance, there are a number of common, yet potentially complex, edits typically performed through direct user interaction with the result document. Changes such as those to the position and size of components can be handled through the use of custom `Interactors`, with supporting graphical cues (e.g. resizing handles) being displayed through custom `Overlays`. Other, component-specific, edits such as changes to fonts, colours etc. can be handled through other aspects of the editor, but whatever the nature of these edits, they must all be effected as changes to the XSLT stylesheet and/or the input XML document.

The specific changes that must be made to the stylesheet or input data are clearly dependent upon the action performed by the user, but they can generally be categorised as either actual edits to the document or as changes to the data instance being shown. This distinction was made in chapter 4, but it is worth revisiting in the context of how the user will interact with the document during editing.

In situations where the user selects a result document component that was constructed using a piece of variable input data, the editing application should provide the user with the ability to change the value of that piece of data to another one, thus creating a modified data instance. But, more generally, other types of edit will result in various changes to the stylesheet that must be reflected in changes to the `Executable` in preparation for re-evaluation. In both cases, the re-evaluation of the document is required to propagate the changes made into the result document instance presented to the user. As discussed in chapter 8, such re-evaluation is instigated as necessary by the result document whenever methods are called on it (i.e. whenever it is accessed). Therefore, all the editing application has to do, in order to refresh the result document view once an edit has been performed, is to simply repaint the result document, which will, in turn, cause various method calls to be made.

9.4 Document Composition

Although we have discussed the process by which changes are made to the stylesheet and/or input data instance, this has been based on the assumption that the document already contains all of the necessary components and that all edits will simply be changes to exisiting components. Clearly, a WYSIWYG editing application is not of much use to creative professionals if they are unable to add or remove components and, in particular, to build new documents starting from a blank canvas.

Adding components to the document requires new instructions to be added to the stylesheet such that, when executed, they will produce the required component at the correct point in the result document. At first, this might seem relatively simple — user interactions with the document view can be handled by an appropriate `Interactor` and the editor can then construct the required stylsheet nodes depending on the component that is to be added, and add these new nodes to the stylesheet where necessary. As an example, if an author wished to add an image to the document, the element shown in figure 9.1 could be added by constructing the correct StyleElement objects and compiling it as previously discussed.

```
<svg:image xlink:href="image.jpg" x="10" y="10" width="200" height="200"/>
```

Figure 9.1 — Stylesheet instruction to produce an output image

The problem then arises of where this instruction should be added to the stylesheet. In fact, this is just one aspect of the larger problem of how the stylesheet should be constructed so that interaction with the editing application is made simple and efficient. As with most editing applications that use an underlying programming and/or markup language, it is not realistic to expect the code produced by the application to be of the same level of complexity that could be achieved via hand-coding by an intelligent programmer. Therefore, although the editor and its associated tools are able to process and display any valid XSLT and XML, the code that is produced as a result of user interaction with the editor may typically seem simplistic and verbose.

One of the most fundamental decisions to be taken is whether to work in a ‘pull’ or ‘push’ based way as previously described in chapter 2. A pull-based approach is arguably more suited to these circumstances where the component creating instructions can be specified in a given order rather than relying on the order of nodes in the input XML document.

Another requirement is for an ‘empty’ document to produce a valid result document, even if it contains no content. The solution to this problem is for the editor to create ‘blank’ documents with a single template that matches the root node of the input document and produces the necessary SVG container nodes in the result document. An example of such a template is given in figure 9.2, which includes nodes for pagination support in the result document.

```
<xsl:template match="/">
   <svg:svg>
      <svg:pageSet>
         <svg:page width="210mm" height="297mm">
            <!-- page content to be added here -->
         </svg:page>
      </svg:pageSet>
   </svg:svg>
</xsl:template>
```

Figure 9.2 — Root Node matching template for ‘blank’ documents

As indicated in the figure, the insertion point for instructions, when adding new components to the document, is within the `<svg:page>` container element. Under a push-based processing model, this entails inserting an `<xsl:apply-templates>` instruction, as well as adding the necessary templates to match the input data nodes. However, a pull-based approach requires that the relevant instructions be added as direct children of this node resulting in a more imperative page description.

The functional nature of XSLT is a major benefit in such circumstances i.e. where numerous instructions are added to the root template to produce multiple output components. Since XSLT instructions are side-effect free, it is possible to add new instructions when a new component is requested without having to be concerned about their effect on other instructions. For example, if instructions to produce a textual component using a specific font were added, there is no danger of affecting other components because of any change of font, as might be the case with other page description languages such as PostScript and PDF [62].

9.5 Component Properties

A typical document will, more often than not, contain more than one type of component (e.g. text, images, shapes) with each of these different components having its own set of properties. For example, a text component might support changes to the typeface and font size used when rendering its contents, but these are not relevant to other components such as images or shapes. For the editing application to allow changes to these disparate properties, it must have knowledge of *what* these properties are and *how* they can be edited. Without such information, the editing application cannot offer a graphically interactive solution to the problem and any changes would need to be made via direct edits to the stylesheet and/or input data instance.

A solution is to provide 'templates' for each of the component types supported by the editor, with details relating to the available properties, and their editing mechanisms, being included within the template itself. These component templates are discussed in more detail in the following section, but it is sufficient to understand that the editing application has knowledge of the various properties supported by the different components, and that it provides interface widgets to edit their values.

In this situation, the editing application behaves in a very similar manner to the way in which 'normal' IDEs, such as Visual Studio, work when designing and building user interfaces for graphical windowed applications. When a component in the document —an interface widget in the case of an IDE —is selected, a series of properties, together with their values, and functionality to edit those values, is presented to the user by the editing application or the IDE.

Computed Properties & Component Relations

As well as supporting appropriate changes to specific property *types*, the editing application must also allow the user to set computed property values or values that reference the input data instance. Therefore, irrespective of the component or property type, the user must always be permitted to specify a computed expression to replace an existing property value. Such an expression may simply be a reference to the input data instance, or it may reference other components within the document. The way in which this expression can be specified by the user is discussed later in this chapter, but the resulting construct would typically be a valid XPath expression. As an example, let us consider the XSLT code shown in figure 9.3, which results in the creation of a block of text in the result document.

```
<svg:text x="10" y="10" color="#000000" font-size="10pt" font-
   family="Helvetica">Some Example Text</svg:text>
```

Figure 9.3 — Example instruction exhibiting non-computed properties

Clearly the editing application must allow for edits to be made to each of the properties specified through attributes, as well as a separate property relating to the textual content of the component. Let us assume that the input data instance has information regarding the style to be used within the document, and that part of that style sets out the typeface to be used. As with a change to any of the available properties, the code shown in figure 9.3 would need to be altered to replace the affected attribute values. For example, the code might be changed as shown in figure 9.4.

```
<svg:text x="10" y="10" color="#000000" font-size="10pt" font-family="{/
   @fontname}">Some Example Text</svg:text>
```

Figure 9.4 — Example instruction exhibiting computer properties

The way in which the edit is made to the XSLT instruction element is discussed in the following sections, but the nature of the computed expressions remains the same.

As well as references to parts of the input data instance, or computed values based upon it, we must also allow properties to reference other components. This type of relationship is illustrated by the grouping or layout of components. A container object that lays out a list of child components, requires that the components in question can be specified. If we return to the previous example, the piece of text described might be one such child. In order for a layout container to be able to reference it, the text component must be given an identifier. The way in which this is achieved is to wrap the component in an `<xsl:variable>` element, with a unique name, as shown in figure 9.5. The name of this variable can then be provided to the layout component as would the case with any other variable expression.

```
<xsl:variable name="foo">
   <svg:text x="10" y="10" color="#000000" font-size="10pt" font-family="{/
      @fontname}">Some Example Text</svg:text>
</xsl:variable>
```

Figure 9.5 — Supporting component interdependencies

9.6 Document Component Templates

We have now highlighted the various problems posed by getting the editing application to interact with the underlying XSLT stylesheet and processing mechanisms. To solve these problems, we introduce the concept of a *document component template*.

Many of the problems stem from the fact that an XSLT stylesheet can produce arbitrary output that need not be valid SVG. Therefore, the underlying XSLT processor also has no limitations on the ways in which the stylesheet is constructed, nor the result produced. However, for editing variable data documents some restrictions are necessary. The editing application works with abstract components, such as text blocks and images, but the processor works with elements, attributes and string values. We therefore require a mapping between the high-level concept of a document component and the XSLT instructions and attributes that are used to produce it.

Figure 9.6 shows an example template for an image component. The template has a unique name, as well as a descriptive name to be displayed to the user by the editing application. The main content of the template can be split in two; the configurable properties of the component, and the XSLT code required to produce the desired effect.

```
<template name="component-e02e64c8-a203-4ee7-a7f3-ce8103f9d701"
    displayname="Image">
   <param name="x" displayname="X"
        editor="testingeditor.property_editors.FloatPropertyEditor"
        default_value="0.0"/>
   <param name="y" displayname="Y"
        editor="testingeditor.property_editors.FloatPropertyEditor"
        default_value="0.0"/>
   <param name="width" displayname="Width"
        editor="testingeditor.property_editors.FloatPropertyEditor"
        default_value="40.0"/>
   <param name="height" displayname="Height"
        editor="testingeditor.property_editors.FloatPropertyEditor"
        default_value="40.0"/>
   <param name="src" displayname="Source"
        editor="testingeditor.property_editors.FilePathPropertyEditor"
        default_value="'file://localhost/Users/james/Desktop/Uni/Research/
       PhD/ThesisProcessor/figures/placeholder.jpg'"/>
   <body>
       <svg:image x="{$x}" y="{$y}" width="{$y}" height="{$height}"
           xlink:href="{$src}"/>
   </body>
</template>
```

Figure 9.6 — Image document component template definition

Each property is specified as a parameter to the template and can be edited using the Java class specified by the `editor` attribute. As an example, consider the ‘x’ property that can be edited by the `FloatPropertyEditor` widget within the editing application, and which has a default value of '0.0'. The <body> element within the template contains all of the XSLT code necessary to produce the required result elements. In the case of the image template shown, this is simply a <svg:image> element that binds the parameter values to specific attributes, although other template types can contain much more complex code.

The result of using templates to specify the link between the editing application and the underlying stylesheet is that stylesheet is constructed in the form shown in figure 9.7.

```
<xsl:template match="/">
   <svg:svg>
      <svg:pageSet>
         <svg:page width="210mm" height="297mm">
            <xsl:call-template name="component-e02e64c8-a203-4ee7-a7f3-
               ce8103f9d701">
               <xsl:with-param name="x" select="100"/>
               <xsl:with-param name="y" select="100"/>
               <xsl:with-param name="width" select="10"/>
               <xsl:with-param name="height" select="10"/>
               <xsl:with-param name="src" select="'photo.jpg'"/>
            </xsl:call-template>
         </svg:page>
      </svg:pageSet>
   </svg:svg>
</xsl:template>
```

Figure 9.7 — Root template in component-based document

Here, a call to the desired template component is added to the <svg:page> element, and any parameters are specified. In the example shown, the actual image file is called photo.jpg, and its bounding box in the result document will be 10x10 pixels, with the top left at co-ordinates (100, 100).

This is not the only step that is required if a component of the required type is not already included in the stylesheet. Because the <xsl:call-template> instruction calls a named template, then a template with that name needs to exist. Such a named template can be created from the component template definition and added to the stylesheet. Once it has been included it can be referenced multiple times from within the stylesheet. The process is as follows:

1. Create an `<xsl:template>` with the name specified by the component template
2. Add an `<xsl:param>` element to the `<xsl:template>` for each of the properties specified
3. Copy the code from within the `<body>` element of the component template to the `<xsl:template>`

Therefore, the named XSLT template corresponding to the Image component template would be defined as shown in figure 9.8.

```
<xsl:template name="component-e02e64c8-a203-4ee7-a7f3-ce8103f9d701">
   <xsl:param name="x">0.0</xsl:param>
   <xsl:param name="y">0.0</xsl:param>
   <xsl:param name="width">40.0</xsl:param>
   <xsl:param name="height">40.0</xsl:param>
   <xsl:param name="src">'file://localhost/Users/james/Desktop/Uni/
      Research/PhD/ThesisProcessor/figures/placeholder.jpg'</xsl:param>
   <svg:image x="{$x}" y="{$y}" width="{$y}" height="{$height}"
       xlink:href="{$src}"/>
</xsl:template>
```

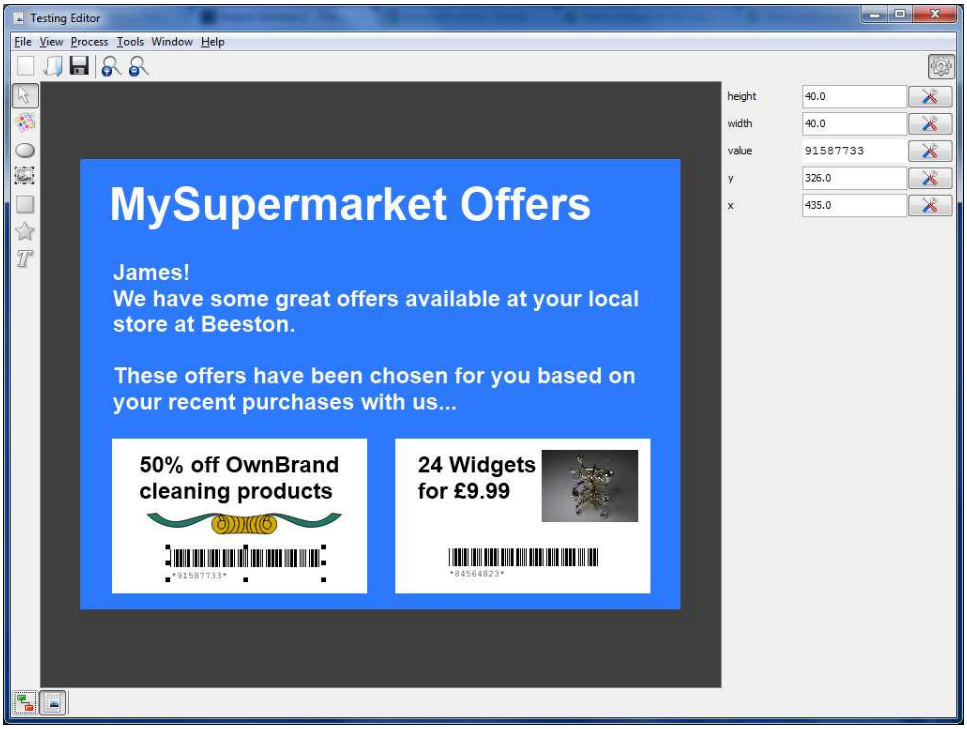
Figure 9.8 — XSLT name template built from Image component template

As we shall see in the next section, this mechanism of component templates allows for easy integration with the user interface provided by the editing application.

9.7 Processor-Integrated Editor

Document View

Figure 9.9 shows a screenshot of the editor with a simple example document being edited. The SVG result document produced by evaluation of the underlying stylesheet is displayed in the central window, and a list of property editors relevant to the currently selected component is shown on the right hand side of the editor window. Direct manipulation of the result document is supported through mouse interactions that allow the movement and resizing of components, as well as other related operations.



height 40.0
width 40.0
value 91587733
y 326.0
x 435.0

MySupermarket Offers

James!
We have some great offers available at your local store at Beeston.

These offers have been chosen for you based on your recent purchases with us...

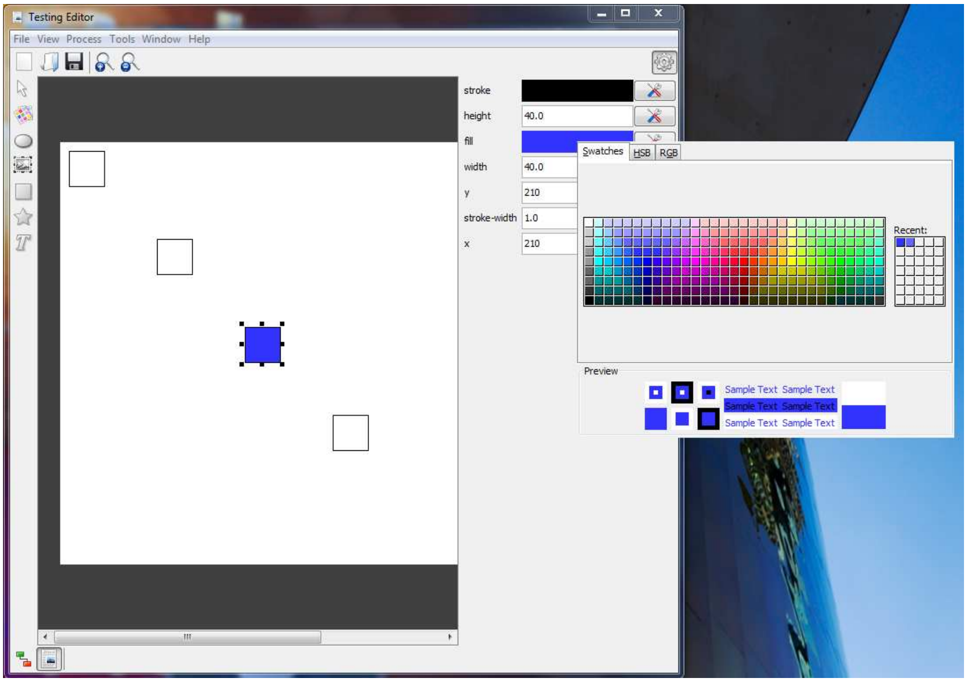
50% off OwnBrand cleaning products
91587733

24 Widgets for £9.99
84564823

Figure 9.9 — Editor view of simple example document

Property Editors

To change properties other than the dimension and position of a component on the page, the document author must make used of the property editing panel to the right of the document view panel. This list of properties is obtained by querying the selected SVG result element for a reference to the component template definition used in its creation. Once this reference is obtained, the editing application constructs the appropriate editing widget as defined in the component template definition, and populates it with the values that were passed as parameters to the named template in the XSLT stylesheet. The different property editor types handle the different values that are relevant to the property in question and, upon alteration, reflect the new value in the correct `<xsl:with-param>` element in the corresponding `<xsl:call-template>` instruction in the stylesheet. The `XSLWithParam` object (a subclass of `StyleElement`) that was changed is then recompiled, as discussed in chapter 7, and the document redrawn (thus triggering the automatic re-evaluation mechanism discussed in chapter 8). An example of a property editor in use (in the case of the example it is a colour-based property that is the subject of the edit) can be seen in figure 9.10.



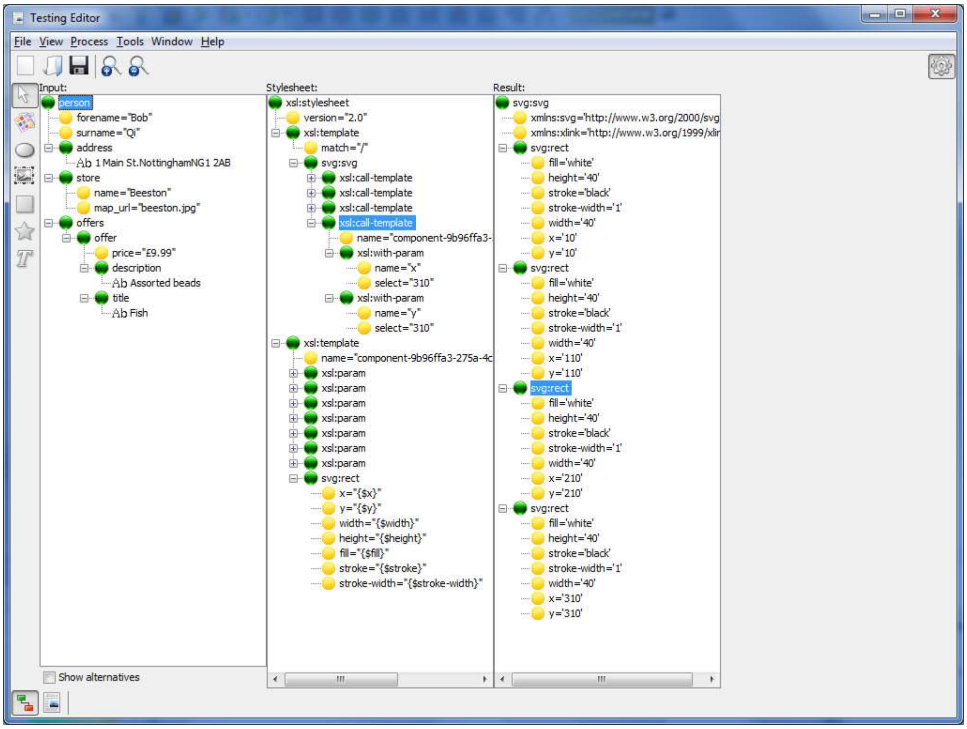
stroke
height 40.0
fill
width 40.0
y 210
stroke-width 1.0
x 210

Figure 9.10 — Colour-based property editor in use

In addition to the specific property editors displayed to the user, each property also has access to a variable expression builder via the button to the right of the property editing widget. In the demonstration editor presented here, this simply allows the user to enter a valid XPath expression as a string, but a more advanced editor could provide a wizard to help build the expression in a more user-friendly manner.

Underlying Tree View

In an actual editing application, the document view discussed above might be the only one available to the document author. However, since the editing application presented here is designed for demonstrating and testing the underlying processing model, it is convenient to have a second view that displays the underlying tree structures of the input data, XSLT stylesheet and SVG result documents. A screenshot of the editor in this mode is shown in figure 9.11.



Input:

```
person
    forename="Bob"
    surname="Qi"
    address
        Ab 1 Main St.NottinghamNG1 2AB
    store
        name="Beeston"
        map_url="beeston.jpg"
    offers
        offer
            price="£9.99"
            description
                Ab Assorted beads
            title
                Ab Fish
```

Stylesheet:

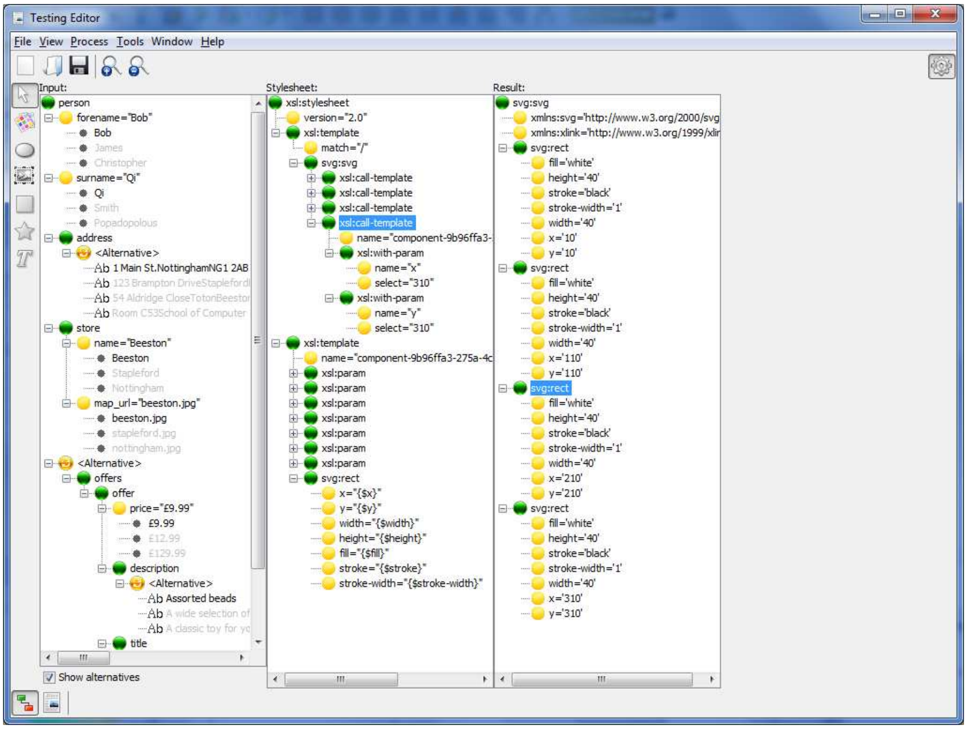
```
xsl:stylesheet
    version="2.0"
    xsl:template
        match="/"
        svg:svg
            xsl:call-template
            xsl:call-template
            xsl:call-template
            xsl:call-template
                name="component-9b96ffa3-
                xsl:with-param
                    name="x"
                    select="310"
                xsl:with-param
                    name="y"
                    select="310"
    xsl:template
        name="component-9b96ffa3-275a-4c
        xsl:param
        xsl:param
        xsl:param
        xsl:param
        xsl:param
        xsl:param
        xsl:param
        svg:rect
            x="{$x}"
            y="{$y}"
            width="{$width}"
            height="{$height}"
            fill="{$fill}"
            stroke="{$stroke}"
            stroke-width="{$stroke-width}"
```

Result:

```
svg:svg
    xmlns:svg='http://www.w3.org/2000/svg
    xmlns:xlink='http://www.w3.org/1999/xlir
    svg:rect
        fill='white'
        height='40'
        stroke='black'
        stroke-width='1'
        width='40'
        x='10'
        y='10'
    svg:rect
        fill='white'
        height='40'
        stroke='black'
        stroke-width='1'
        width='40'
        x='110'
        y='110'
    svg:rect
        fill='white'
        height='40'
        stroke='black'
        stroke-width='1'
        width='40'
        x='210'
        y='210'
    svg:rect
        fill='white'
        height='40'
        stroke='black'
        stroke-width='1'
        width='40'
        x='310'
        y='310'
```

Figure 9.11 — Editor tree view of underlying document structures

Each of the tree views shows the underlying structure of one of the documents held within the processor, and allows us direct access to them. As an example of this, the input data instance, shown in the far left view, can be manually altered to produce a new data instance. The default view is to show the data instance as it is seen by the processor (i.e. it has no knowledge of alternative nodes/values and looks like a standard document), but this behaviour can be changed by selecting the checkbox below the view. With this checkbox checked, the complete structure of the switchable instance is exposed and the currently selected node at each of the alternative points in the instance can be replaced with any of the other supported nodes/values by right-clicking on the desired alternative and selecting the appropriate command from the resulting popup menu.. A screenshot of the editor displaying the full structure of the switchable instance is shown in figure 9.12.



```
person
  forename="Bob"
    Bob
    James
    Christopher
  surname="Qi"
    Qi
    Smith
    Popadopolous
  address
    <Alternative>
      Ab 1 Main St.NottinghamNG1 2AB
      Ab 123 Brampton DriveStapleford
      Ab 54 Aldridge CloseTotonBeestor
      Ab Room C53School of Computer
  store
    name="Beeston"
      Beeston
      Stapleford
      Nottingham
    map_url="beeston.jpg"
      beeston.jpg
      stapleford.jpg
      nottingham.jpg
  <Alternative>
    offers
      offer
        price="£9.99"
          £9.99
          £12.99
          £129.99
        description
          <Alternative>
            Ab Assorted beads
            Ab A wide selection of
            Ab A classic toy for yo
        title
```

```
xsl:stylesheet
  version="2.0"
  xsl:template
    match="/"
    svg:svg
      xsl:call-template
      xsl:call-template
      xsl:call-template
      xsl:call-template
        name="component-9b96ffa3-
        xsl:with-param
          name="x"
          select="310"
        xsl:with-param
          name="y"
          select="310"
  xsl:template
    name="component-9b96ffa3-275a-4c
    xsl:param
    xsl:param
    xsl:param
    xsl:param
    xsl:param
    xsl:param
    xsl:param
    svg:rect
      x="{$x}"
      y="{$y}"
      width="{$width}"
      height="{$height}"
      fill="{$fill}"
      stroke="{$stroke}"
      stroke-width="{$stroke-width}"
```

```
svg:svg
  xmlns:svg='http://www.w3.org/2000/svg
  xmlns:xlink='http://www.w3.org/1999/xlir
  svg:rect
    fill='white'
    height='40'
    stroke='black'
    stroke-width='1'
    width='40'
    x='10'
    y='10'
  svg:rect
    fill='white'
    height='40'
    stroke='black'
    stroke-width='1'
    width='40'
    x='110'
    y='110'
  svg:rect
    fill='white'
    height='40'
    stroke='black'
    stroke-width='1'
    width='40'
    x='210'
    y='210'
  svg:rect
    fill='white'
    height='40'
    stroke='black'
    stroke-width='1'
    width='40'
    x='310'
    y='310'
```

Figure 9.12 — Editor tree view showing alternative input data values

An additional feature — of great benefit when testing and performing analysis of the processing framework — is provided through the result document tree view. Each node in the tree can be selected and, when this occurs, the various state information stored alongside that node is presented to the user. As well as viewing the internal state information, we can also force re-evaluation of that node outside of the automatic re-evaluation mechanism. This provides us with manual control over the parts of the stylesheet that are re-evaluated which can be used to analyse specific re-evaluation situations in isolation.

Although the editor is a prototype, it has proved sufficient to demonstrate the techniques discussed. The addition of the document structure view and the possibility of manually controlling the (re-)processing of the document makes the editor a good basis for the performance analysis of the partially re-evaluating toolchain described in the following chapter.

Chapter 10:

Performance Analysis

Previous chapters have discussed the ways in which the overheads of repeated re-processing of document components can be reduced and, hence, how documents with localised changes can be optimally re-evaluated. In this chapter we explore the benefits of these techniques by analysing the performance of the tools developed when re-processing a range of representative variable data documents. The performance analysis will be done using the editing application presented in the preceding chapter and a corresponding toolchain based on unmodified processing tools.

10.1 Methodology

Before we consider measuring performance, we first consider what aspects of 'performance' we are interested in measuring. The aim of this work is to provide techniques for increasing the speed at which documents can be re-processed to support a more responsive interactive editing paradigm. Hence the primary performance metric that we are interested in measuring is the reduction in re-processing time that results from an edit to the document. However, the proposed techniques of storing and restoring processing state introduce some computational overhead as well as a larger memory load, both of which must also be considered.

The potential reduction in re-processing is directly related to the complexity and composition of the document in question as well as the change resulting from an editing operation. Situations involving simple localized changes to large, complex documents are candidates for the largest optimizations. But, in the worst case, documents where everything changes must of necessity, be completely re-processed. In such cases, the amount of processing performed is necessarily *larger* with the modified processing tools due to the overhead of storing state information during re-processing. One of the aims of the following analysis is to determine the extent to which such processing overhead affects processing speed in such circumstances.

Representative Documents and Edits

The potential performance gains obtained by using the processing tools already developed are heavily dependent on the complexity and composition of the document as well as the nature of the edit that has been performed. To obtain representative results, a range of different documents and associated edits must be large enough to encompass the variety of situations that might arise during a typical editing session.

Therefore test cases covering the following scenarios will be evaluated:

- Adding components to the document without affecting other existing document components
- Changing properties of existing document components
- Reprocessing the document as a result of an edit

Each test case scenario will be evaluated across a range of documents that differ in the number and type of components that they contain.

Testing Platform

All of the tools used as the basis for this work are written in the Java programming language, and so must be executed by a Java Virtual Machine (JVM). For this performance analysis, we use the default JVM supplied by Oracle [63] running on the Windows 7 operating system from Microsoft. It should be noted that this analysis would be equally valid using other JVMs on other operating system platforms, however due to the different system architectures and implementations, the performance across platforms would not be consistent. Therefore, all test cases will be run on the same operating system/JVM installation.

In the following section, we examine how to measure the time taken to execute various parts of the modified and unmodified toolchain. Then we propose ways in which the measurements can be made while minimising the effect of, or eliminating, these issues. These measurements will be used to indicate the relative performance increase/decrease as a result of the optimizations (and associated overhead) discussed in this thesis.

10.2 Sources of Error

As with all experimental measurement, there are a number of ways that errors can be introduced. The following subsections detail the anticipated sources of error and the precautions taken to avoid them.

10.2.1 Experimental Error

One of the forseeable problems when running test cases on our modified toolchain is that of classloading and caching. On the first reference to a given type of

Java object, the class definition must be loaded by a `ClassLoader` before the object can be instantiated. Once the class definition has been loaded it will remain so, and thereafter it can be used to create subsequent objects of that type, until it is explicitly unloaded or the associated `ClassLoader` is destroyed. In the case of the default JVM `ClassLoader`, class definitions cannot be unloaded and therefore they will remain loaded until the JVM terminates execution of the program.

The effect of this process is that the time taken to instantiate an object of a given type takes much longer the first time the class is referenced (due to the overhead of loading the class definition) than on subsequent instantiations. This overhead might potentially affect the measurements we make and so each test case will be repeated a number of times, with the lowest value being used. It might naively be assumed that the *average* time would provide a more reliable mesurement, but the argument can be clearly made for using the minumum value instead. Any given test case is an abstract representation of a series of operations that must be performed to carry out the required task. Because these operations must always be performed to complete the task, and it is only these operations that we wish to time, *any* other operations performed by the operating system etc. are superfluous and take additional time, thus adding to the measured value. Therefore, the test iteration performed in the shortest time is the one with the least interference from other sources and hence is the closest to the real value attributable to the test case itself. The repetition of measurements should discard any results affected by such overhead as well as any anomalous results that occur for other unforseen reasons.

10.2.2 Measurement Resolution

The aim of these performance tests is to ascertain the amount of time required for our modified and unmodified toolchains to reprocess a series of documents that have been subject to specific edits. To do this we must accurately record the time taken to execute the reprocessing routines inside the processor. At first glance, a system by which we record the absolute time at the start of the process and then compare it to the absolute time after execution should provide the correct value. There are however two major issues with this approach — timing resolution and multithreading/multitasking.

Firstly, we are limited to the resolution provided by the various time-access methods provided by Java. There are two main methods provided which can return time values in milliseconds and nanoseconds respectively: `System.systemTimeInMillis()` and `System.nanoTime()`. However, although the values returned will be of the correct order, the resolution of these values is implementation dependent and not guaranteed to match the 'advertised' precision. `System.nanoTime()` uses mechanisms only available on certain platforms and resorts to scaling the value returned by `System.systemTimeInMillis()` when these mechanisms are not available. Furthermore, the value returned by `System.systemTimeInMillis()` is dependent upon the JVM implementation and underlying operating system and, in some cases, can only provide values with a resolution of 10ms.

Secondly, and more problematically, running an inherently multithreaded program on a multitasking operating system means that we can never be sure how much of the time that elapsed between starting and ending the reprocessing routines was actually used performing the task we are trying to measure. It is entirely possible (even likely) that after we have recorded the start time, but before we finish reprocessing, the JVM may block the current thread to allow other threads to run. This second thread may then run for some time (for example, redrawing the user inferface) before our original thread is allowed to continue executing.

However, not only can the JVM block the current thread, but the OS may block the JVM process itself to allow other processes to execute. Preventing OS process switching can only be guaranteed by using a non-multitasking operating system or a real-time operating system with custom scheduling algorithms, however its effects can be minimised by stopping all unnecessary programs and services from running. This, coupled with the repetition of tests, will decrease any such errors introduced when performing the tests.

10.2.3 JVM Issues

Although we have specified a common platform on which all our tests will be performed, the JVM itself is a final point of possible error. As well as the problems dscribed above, there are a number of other JVM operations that could affect our timing measurements. The pertinent issues are discussed in the following subsections, with suggested remedial action where appropriate. A more detailed discussion of general JVM issues with respect to program benchmarking can be found in [64].

Hotspot Compiler

Unlike traditional languages such as C, Java source code is compiled to an intermediate bytecode rather than directly to machine instructions. This bytecode is then converted to machine code by the JVM, either by simple interpretation or

Just-in-time (JIT) compilation. Since JIT compilation has a relatively high initial cost, the JVM defaults to interpreting *all* parts of the program to avoid long start-up times. However, bytecodes compiled to native machine instructions execute much more quickly than when they are simply interpreted. Therefore, the JVM analyzes the code as it runs and records statistics based on the amount of time spent executing particular pieces of code. When a 'hotspot' is detected, the relavent bytecodes are JIT compiled and executed to increase performance. Other optimizations such as method inlining are also performed by the JIT compiler at this stage to further improve performance.

Although this mechanism of interpretation and compilation works well in striking a balance between speed and responsiveness, it can cause serious problems when attempting to analyze the performance of our modified processing tools. Because the point at which a section of the program's bytecodes will be JIT compiled is determined by the JVM, it is entirely possible that one invocation of the code will be interpreted and the next compiled. This would cause a large spike owing to the time taken to compile the bytecode, followed by consistently faster execution, which could seriously affect our measurements. In our set of test cases, it is quite probable that those involving larger, more complex documents will cause parts of the program to run enough times that it will trigger the JVM's hotspot compiler.

We could attempt to negate this problem by running the tests numerous times before we start taking measurements in an attempt to *force* the JVM to JIT-compile all the relevant sections, however this is time-consuming and there are better solutions. Firstly, there exists a JVM option called `-XX:CompileThreshold` that can be specified when starting the JVM. This allows us to set the number of times a piece of code is interpreted before it is JIT compiled, but this will still cause a delay when that limit is reached.

Secondly, there are another set of JVM options that can be used to control its behaviour with respect to hotspot compilation — `-Xint`, `-Xcomp` and `-Xmixed`. These instruct the JVM to always interpret the bytecodes (without any JIT compilation), to always JIT compile the bytecodes before execution, or to use a mixture of the two approaches as described above. If none of these options is specified, the JVM will default to `mixed` mode. Specifying that the compiler should always JIT compile (`-Xcomp`) suffers from the same problem of initial compilation delay as lowering the hotspot threshold using `-XX:CompileThreshold`. This should not be a suprise since setting the compile threshold to 1 has the effect of JIT compiling each section of the program when it is first encountered. By forcing the JVM to *never* JIT compile the bytecodes, our modified processing tools will potentially execute more slowly, however the execution will progress in a much more uniform and predictable manner. This is essential when performing the sort of timing tests presented here, and so the problem of hotspot compilation will be avoided by passing the `-Xint` flag to the JVM.

Garbage Collection

In contrast to languages like C and C++, Java hides much of the task of memory allocation and reclamation from the programmer. Memory is automatically allocated when a new object is created and it cannot be explicitly released by the programmer; instead the destruction of objects is performed by the garbage collecter implemented within the JVM once the object is no longer referenced. The programmer can request that the garbage collector be run via the `System.gc()` method call, but this is merely a *request* and the Java specification makes no guarantees as to when, or if, the JVM will service it. This has the potential to cause problems when running timing tests since the execution of the processor may or may not be paused while the garbage collector runs.

One solution to minimizing this problem is to redundantly maintain references to every object created so that none are released during the execution of the code that we are measuring. Although the garbage collector will still run, no objects will be eligible for collection and so the time spent by the JVM would be negligible. This is, however, a false economy since the process of creating and destroying objects is an integral part of the processor's execution. Any increase or decrease in the number of objects created (and therefore time spent managing them) is intrinsic to the way in which the modified processor operates and so artificially removing this aspect of its execution would unfairly skew the results. Furthermore, holding such references would provide a non-representative memory load during execution, preventing any analysis of this aspect of the modified toolchain's performance. In extreme cases, the resulting large memory load could also cause the operating system's memory manager to swap out pages containing parts of the toolchain which must then be swapped back in when necessary, thus causing large delays in execution.

10.2.4 DOM Differences

As discussed in previous chapters, the default implementations of the input data tree and the generated result document are replaced with custom versions in the modified toolchain. These new implementations are required to provide the extra functionality required of them, but this has implications on their performance relative to the default implementations. Clearly, they do not function in the same way and so it is not surprising that they may offer different performance during execution. The custom implementations of the input data document and the generated result document are integral to the modified toolchain and so any increases or decreases in processing cost are a reflection of the modified toolchain as a whole. It should also be noted that in the comparison tests using an unmodified toolchain, the input document will be represented using the object-bsaed implementation offered by Saxon due to its similarity in implementation to the custom implementation used by the modified processor.

10.3 Testing Framework Setup

10.3.1 Performance Measuring Code

To measure the performance of the different testing setups, extra code must be added to make the necessary measurements as the tests are run. We need to ensure that the same timing mechanisms are used, and that they are also used in a consistent way, so we introduce the `PerformanceMonitor` class that can be used from within both of the test setups. Calls to the various methods supported by the `PerformanceMonitor` are made from within the processing methods of the different setups to record the time take to perform various tasks. The information gathered can then be output at the end of each test run and subsequently used to analyse the relative processing costs.

The `PerformanceMonitor` class allows for the recording of different tasks during a test, as well as supporting multiple iterations of the same task to overcome any anomalous results as previously described. Therefore, the series of steps, and associated method calls, needed to perform the measurement of a given task is as follows:

1. Inform the PerformanceMonitor that a new task is being undertaken via a call to startTask(String name) where name is a string identifying the task being performed.
2. For each iteration of the task, the startIteration() method should be called. This records the absolute system time within the PerformanceMonitor object.
3. The endIteration() method should be called after the code to be measured has executed. This compares the absolute system time with the value previously stored and records the difference.

A code sample showing this process is given in figure 10.1.

```
public void someMethod() {
    /*truncated*/

    //get a reference to the PerformanceMonitor object
    PerformanceMonitor pm = PerformanceMonitor.getInstance();


    //indicate that we are starting a new task
    pm.startTask("Descriptive task name");

    //do the task many times to overcome anomalous results
    for (int i = 0; i < 500; i++) {
        pm.startIteration();

        //code we want to measure goes here

        pm.endIteration();
    }

    /*truncated*/
}
```

Figure 10.1 — Example code showing calls to the PerformanceMonitor object

Once a task has been completed, the PerformanceMonitor object can be queried regarding the results of the iterations of that task. It is these results that we present later in this chapter. A complete code listing of the `PerformanceMonitor` class can be found in Appendix B.

Measuring Memory Usage

In addition to measurements of the processing cost, we also consider the memory overheads introduced by the techniques described. The measurement of memory usage is more complex than that of processing time and, therefore, we turn to specialised tools to obtain representative results. The tool that would initially appear to be best suited is a code profiler such as the VisualVM profiler [65] that comes bundled with the latest version of the Java Development Kit (JDK). Like many other code profiling tools, VisualVM provides overall JVM memory usage statistics as well as individual class size tracking. However, both of these solutions suffer from problems that make this approach unsatisfactory. Firstly, monitoring the overall usage of the JVM makes the results subject to the same potential issues that were previously discussed in relation to measuring execution time. In the case of execution time measurements, these issues might only have minor effects, but they can be the source of *major* discrepancies with respect to memory usage statistics. Perhaps the most troublesome of these is the garbage-collector used by the JVM to reclaim unreferenced objects. As discussed, the JVM is under no obligation to respect any requests to run the garbge collector, and so any memory usage statistics may include objects and values that are no longer in use. Furthermore, the numerous optimizations performed by the JVM (in particular those that employ caching mechanisms), can increase the measured memory load even further.

The second type of functionality offered by VisualVM appears to be more promising. It provides live statistics relating to the number, and size, of each object instantiation. However, the figures presented only show the memory size of the individual objects and does not include the size of other objects that are referenced by them. Therefore, the size of our `ProcessorState` objects might appear very small because the value reported does not include the size of any referenced objects.

Fortunately, a tool exists that extends the type of functionality offered by VisualVM so that the entire dependency graph of an object is measured. ClassMexer [66] uses the same underlying functionality offered by the JVM that VisualVM uses, but it traverses all referenced objects and records the complete size of an object. The measurement is performed by simply passing a reference to the object to be measured to a static method provided by ClassMexer. It then traverses the object graph (while maintaining a list of previously measured objects so as not to include them twice if they are referenced by more than one object) and returns the overall size. Given the accuracy of this approach, the memory usage measurements made in the tests presented later, are performed using this tool.

10.3.2 Control Setup

The purpose of the tests and analysis presented here is to ascertain the performance characteristics of the modified toolchain. To understand how it performs in certain circumstances, it will be subject to a series of tests, but without a baseline against which to judge the outcome, the results are meaningless. Therefore, the tests will be run not only on the modified toolchain, but also on the orignal 'vanilla' version of the processor so that a direct comparison can be made. It should be noted that the only modifications made to the source code of the processor used in the control experiments are the addition of the various method calls to the `PerformanceMonitor` class that are necessary for the measurements to be made, as well as simple looping constructs, where necessary, to ensure that the code in question is executed a number of times.

Since the measurement method calls will only be placed around the particular pieces of code that are of interest with regard to the tests, the cost of other tasks, such as loading and parsing the input files, that are not of interest, will not affect the results produced. Therefore, the tests run on the 'vanilla' setup will be executed from the command line in the following way:

```
java -Xint -jar vsaxon.jar -xsl:trans.xsl -s:data.xml -o:out.svg
```

where the 'vanilla' version of the processor is compiled to a JAR file called `vsaxon.jar` and the various input and output files names/paths are specfied in the default way as expected by Saxon. In addition, the `-Xint` parameter is passed to the JVM to ensure that the test is run in a fully interpreted way as described earlier in the chapter. The results of the test will be output by the `PerformanceMonitor` class when the processing is complete and the processor quits.

10.3.3 Experimental Setup

The experimental toolchain on which the tests are also performed is the demonstration editor introduced in the previous chapter. Again, the necessary calls to the `PerformanceMonitor` class, as well as looping constructs to ensure repeated execution, are added around the pieces of code that are to be measured. The editor is also compiled to a JAR file (`editor.jar`) and is executed from the command line in a similar manner to the control setup. The command used is given below:

```
java -Xint -jar editor.jar
```

The `-Xint` JVM flag is once again set so as to ensure that both test setups are interpreted by the JVM as opposed to being JIT compiled. It should also be clear that the various input and output file names/paths are not provided because this is done from within the editor during its execution. However, since the `PerformanceMonitor` class remains unchanged from the version used in the control setup, the measurement results will still be output to the command line.

10.4 Results

The following sections detail individual test cases used to explore certain performance characteristics of the modified processor and associated data structures. In most cases, this involves executing an example document and performing some action, with the test being repeated on both the modified and unmodified toolchains for comparison purposes. Clearly, the values obtained as a result of completing these tests are dependent on the particular hardware used, but we are less interested in their absolute values but more in their *relative* performance. In addition, example documents of differing size and complexity will be used, where appropriate, to illustrate the performance in a particular test case across a range of simulated documents. In many cases, this range simply contains documents with a varying number of components on a single page, thus requiring varying amounts of re-processing when reacting to an edit to the document.

10.4.1 State Storage Overheads

Given that the modifications to the processor require its execution state to be gathered and stored, as described in chapter 6, this additional work inevitably leads to an *increase* in the cost of processing the XSLT stylesheet, particularly in the first instance. These 'overheads' are introduced in all processing situtations (since the state must always be stored to support partial re-evaluation), and therefore counteract the possible savings made available by targetted partial re-evaluation. The purpose of this test is, therefore, to ascertain the extra processing costs attributable to the process of storing state information.

To obtain representative figures for this test case, we perform comparative tests between the unmodified version of Saxon and our modified one. The measurements taken are limited *only* to the cost of fully processing the document (including producing the result document); they do not include the cost of setting up the procesor or parsing and loading the test documents. In the case of the modified processor, a complete re-evaluation of the document is performed by using the partially re-evaluating mechanism, by requesting that the subtree to be re-evaluated starts at the root node of the document (and hence includes the whole document).

To avoid the problems relating to classloading delays etc. that were discussed earlier in this chapter, each (re-)processing operation will be executed 500 times within each of the processors.

Test Documents

Clearly, the construction of the document has a large bearing on the overhead costs associated with storing state information. A stylesheet that utilizes large numbers of variables and parameters will introduce higher overheads than documents that do not use any. Therefore, example documents that exhibit a wide range of such state-affecting constructs are presented. In addition, each of these examples will be produced with varying numbers of document components in order to obtain data over a range of document sizes. As an example, the document shown in figure 9.9 is relatively simple since it only contains a few components, however the barcodes shown are computationally expensive to generate and so a document containing many such components would require significant processing time. An increase in document complexity is therefore easily simulated by adding more components to the document, thus increasing the overall processing requirement.

The documents used in this test are produced with the following configurations:

1. A single template containing varying numbers of `<svg:text>` elements contain the test string ‘`Hello World!`’
2. A root template with varying numbers of calls to component template definitions[1], with parameters passed as necessary

Processing Cost

The results of the first (minimal state storage) configuration for the modified and unmodified processors are shown in tables 10.1 and 10.2 respectively.

Components	Min (ms)	Max (ms)	Mean (ms)
10	0.558	40.888	0.707
20	1.009	24.808	1.232
40	1.911	14.446	2.238
70	3.257	16.999	3.644
100	4.601	16.392	4.780

Table 10.1 — Processing costs using unmodified processor

Components	Min (ms)	Max (ms)	Mean (ms)
10	0.560	9.067	0.594
20	1.012	26.906	1.131
40	1.915	35.622	2.071
70	3.271	91.680	3.690
100	4.623	73.190	4.946

Table 10.2 — Processing costs (complete re-evaluation) of modified processor

[1] The template definition is of the sort described in chapter 9 and, specifically, is the Text template detailed in Appendix B with the string ‘`Hello World!`’ passed as a parameter

The minimum processing times for each document in the data sets are plotted on the graph shown in figure 10.2 for comparison purposes.

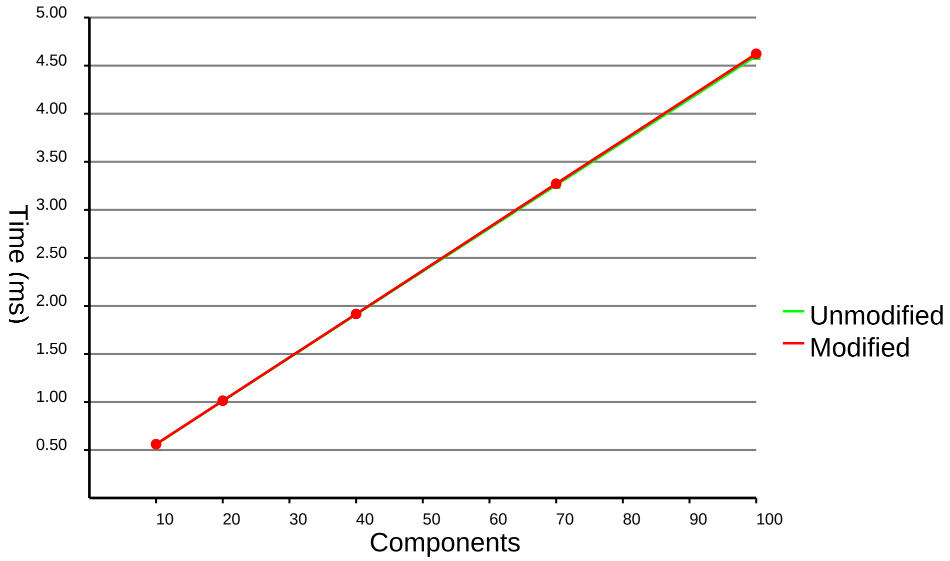


Figure 10.2 — Comparative results showing minimal state storage overheads

The graph shows that the overheads introduced on these simple documents are minor. Across the range of document sizes, the cost of storing the state of the processor is negligable. Given the simple nature of the test documents, and hence the lack of information that needs to be stored, the results are not entirely unexpected.

We now consider the overhead costs associated when processing documents that exhibit more state-affecting constructs, as discussed above. In each case, a specific number of calls are made to the named template responsible for creating the result document component in question. The results of these tests for the unmodified and modified processors are shown in tables 10.3 and 10.4 respectively.

Components	Min (ms)	Max (ms)	Mean (ms)
10	1.900	14.931	2.073
20	3.729	22.579	4.112
40	7.290	22.550	7.676
70	12.674	55.149	13.524
100	18.115	46.079	19.284

Table 10.3 — Processing costs using unmodified processor

Components	Min (ms)	Max (ms)	Mean (ms)
10	3.072	33.115	3.355
20	6.077	50.451	6.866
40	12.091	64.618	13.475
70	22.211	70.591	23.849
100	30.135	77.313	33.834

Table 10.4 — Processing costs (complete re-evaluation) of modified processor

It should be noted that a direct comparison between the absolute values for the tests presented here, and those presented earlier, which involved documents with minimal state-affecting constructs, is not possible. This is due to the fact that the stylesheets used in both cases are vastly different, and that the latter requires many more instructions to be evaluated in its execution. However, the comparison between the modified and unmodified processor in each case is the sole purpose of these tests, and when considered together, the different tests describe two extremes of the same phenomenon.

As with the previous test results, a graph showing the comparative performance of the two test configurations is given in figure 10.3 below.

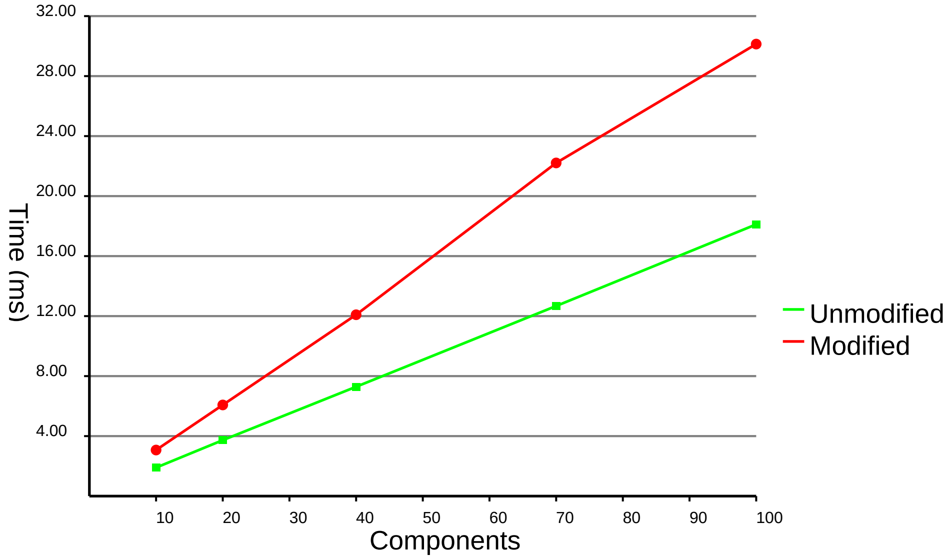


Figure 10.3 — Comparative results showing higher state storage overheads

As expected, the number of variables/parameters etc. that are utilized by the document (and therefore require storing as part of the execution state) has a large bearing of the overheads introduced by storing the processor's state and data/ stylesheet usage references. In contrast to the results for the first test, the overheads shown in figure 10.3 are much larger (around 60%-70%). Clearly, in this worst-case scenario our modified processor performs relatively badly, but when operating in partially re-evaluating mode, such a complete re-evaluation would rarely be necessary. In many cases at least some region of the document will remain invariant between one generated result instance and the next, and it is these circumstances that the following sections consider.

Memory Cost

Before moving on to test cases involving partial re-evaluation, we analyse the memory cost of storing the required state information. By using the ClassMexer tool discussed earlier, we can obtain accurate sizes for the collections of objects that we are interested in. In the case of both test systems, this includes the object hierarchies of the input document, stylesheet tree, stylesheet executable, and result document.

As with the execution time tests, both the modified and unmodified processors will execute a variety of documents that range in size (number of components) and variable/parameter usage. To avoid any confusion, the same test documents are used as in the execution timing tests. The results for the two tests are shown in tables 10.5 and 10.6, and figures 10.4 and 10.5 below.

Components	Unmodified (kB)	Modified (kB)
10	108.41	140.95
20	118.66	166.64
40	137.13	212.35
70	167.86	289.68
100	196.63	360.44

Table 10.5 — Memoy overheads when processing minimal state documents

Components	Unmodified (kB)	Modified (kB)
10	124.95	201.77
20	147.35	287.07
40	191.16	457.36
70	257.85	713.20
100	323.68	969.03

Table 10.6 — Memory overheads when processing state-intensive documents

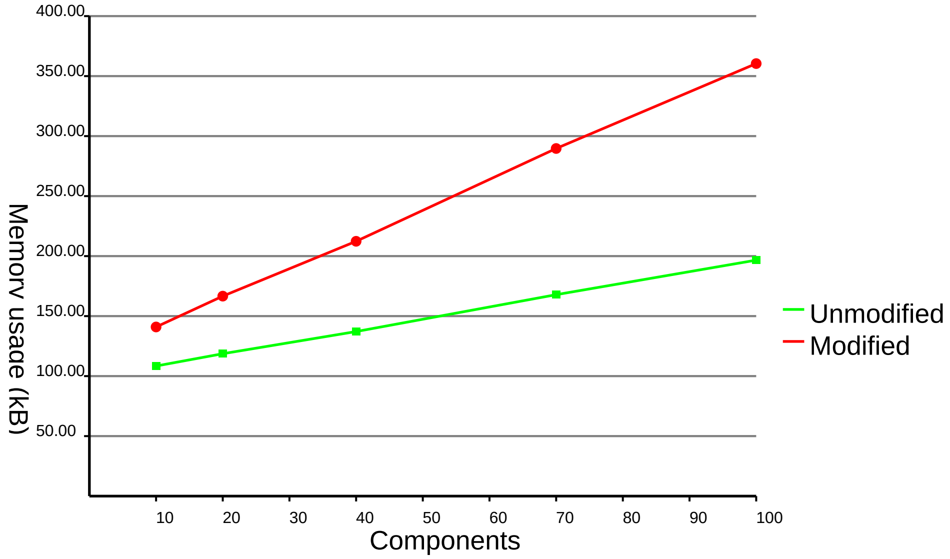


Figure 10.4 — Memoy overheads when processing minimal state documents

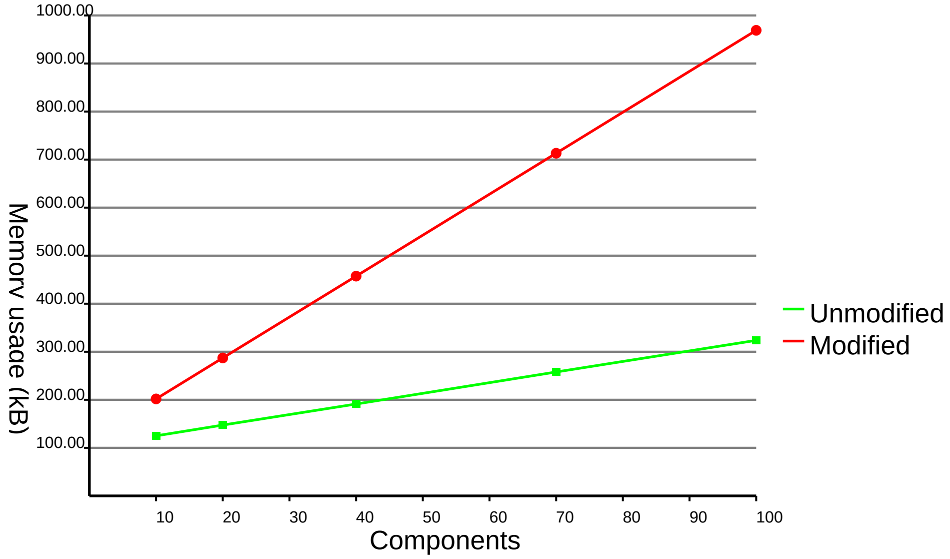


Figure 10.5 — Memoy overheads when processing state-intensive documents

As expected, the results in both test cases show a larger memory load for the modified processor due to the extra state information being stored. This is especially the case for documents that utilize a large number of state-affecting constructs. Some of this extra memory load is a consequence of the way that the internal Saxon classes are constructed. For performance reasons, we retain references to the objects already created and used by Saxon. However, because Saxon was not designed with state-storage functionality in mind, these constructs are not always optimal with respect to persistant memory storage. Furthermore, it is important that we consider the results in the context of authoring real-world documents in a WYSIWYG editing application. Firstly, the generated document must be rendered so that it can be displayed to the user. In the case of our example editing application, this rendering is handled by the Batik SVG libraries, which have memory overheads of around 4-5Mb. Clearly, when we consider documents containing high resolution images etc. these rendering overheads will necessarily increase. Given the fact that the test results show a *total* memory load (including data and stylesheet representations) of less than a megabyte, the extra cost of storing the required state information does not affect the viability of the approach.

10.4.2 Single Component Re-Evaluation

In contrast to the situation described above, the best case for our partially re-evaluating processor is when only a single component requires reprocessing. The type of edit that might lead to such circumstances can easily be imagined if we consider simply moving a component that is not related to any other. Here, the state information stored on the element to be re-processed is used to build an appropriate `XPathContext` object, and this is subsequently used to process only the relevant instruction(s).

The results shown in tables 10.7 and 10.8 correspond to the processing cost of manually selecting a single node in the result document and requesting that it be re-evaluated. The documents used are the same ones used in the previous section.

Components	Min (ms)	Max (ms)	Mean (ms)
10	0.111	18.885	0.174
20	0.111	21.802	0.184
40	0.111	20.337	0.190
70	0.111	24.758	0.199
100	0.112	25.248	0.195

Table 10.7 — Cost of reprocessing a single component (minimal state)

Components	Min (ms)	Max (ms)	Mean (ms)
10	0.286	29.142	0.384
20	0.286	37.371	0.406
40	0.287	14.309	0.338
70	0.290	20.114	0.379
100	0.290	25.307	0.384

Table 10.8 — Cost of reprocessing a single component (state-intensive)

We see that the cost of re-processing a single component is virtually constant, irrespective of the size of the document. This is not surprising since we are simply executing a fraction of the stylesheet that was fully executed in the initial processing pass. It should be noted that in the test case presented here, no modifications are made to the document between the initial processing and the subsequent partial re-evaluation. The consequence of this is that the the result document instance is identical to its original form after the re-evaluation has been performed.

This situation is sufficient to illustrate the performance of re-evaluating a single component, but the problem of reacting to, and realising, edits is the subject of later tests.

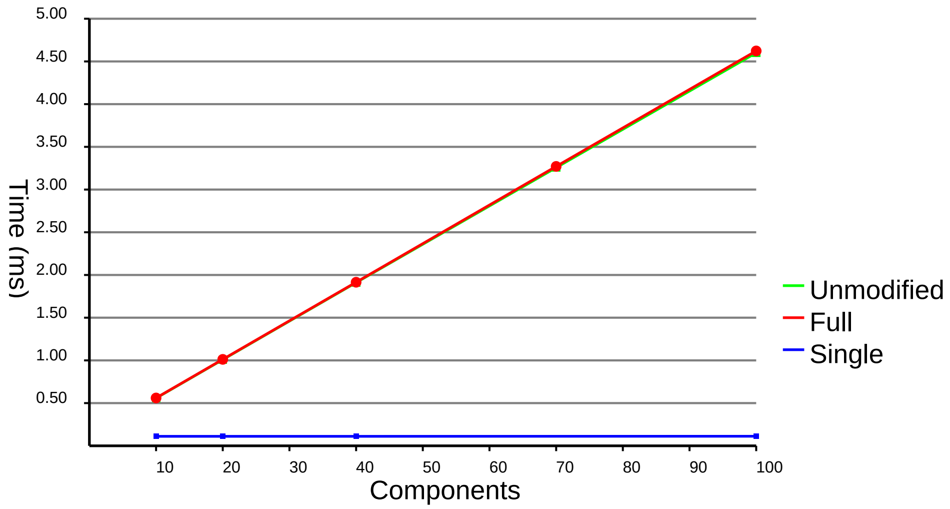


Figure 10.6 — Comparison of minimal state configurations

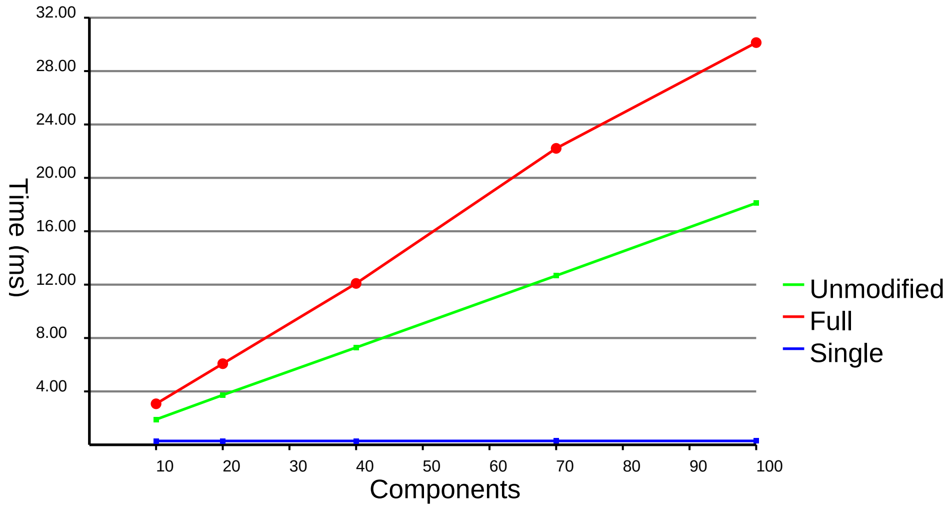


Figure 10.7 — Comparison of state-intensive configurations

Figures 10.6 and 10.7 combine the results of this test case with those presented for the complete re-evaluation tests discussed earlier. The potential gains that can be made via partial re-evaluation clearly increase as the size/complexity of the document increases (and hence the full re-processing cost also increases). The two lines representing the results from the modified processor illustrate the upper and lower bounds of the processing costs that can be expected with the given documents. Any actual edit to the document will result in a re-evaluation cost somewhere between these limits. Returning to the document depicted in figure 9.9 as an example, the cost of reprocessing one of the textual components might be relatively inexpensive (and hence close to the lower limit), wheras the reprocessing cost of one of the barcodes would be much higher due to its increased computational complexity.

10.4.3 Identification of Affected Nodes

So far, we have considered the worst case of re-evaluating the entire document and the best case of re-evaluating just a single simple component. To achieve this best case, without the user manually performing the re-evaluation, the result document nodes that are affected by an edit must be automatically re-evaluated. Chapter 8 has discussed a mechanism that can provide such automatic re-evalution, and we now investigate the processing cost of doing so.

We recall that there are four stages to initiating the re-evaluation of a result element:

1. Record the usage of the stylesheet/data during execution.
2. Use this data to link data/stylesheet nodes with the result document elements as they are produced.
3. Notify the appropriate result elements when a change to the data/stylesheet is made.

4. Re-evaluate as necessary as the result document is consumed

The cost of the first of these stages is included in the state storage costs detailed in section 10.4.1. The second stage is performed each time a result element is produced and, thus, is also included in the other test cases presented so far. Indeed, it is this cost that is a major contributor to the overheads encountered when processing the state-intesive tests, as opposed to the minimal state tests, presented in section 10.4.1. The notification of result elements to the fact that an instruction/data node has been changed is initiated from the `Instruction`'s `compile(...)` method, or the data tree modification method respectively. This operation is simply a case of setting a boolean flag on the result element indicating that it requires re-evaluation and so takes a trivial amount of time. Finally, the re-evaluation is performed by the result node itself as a result of it checking its status flag when accessed. Again, this check is simply a boolean comparison and takes negligible time before the node is re-evaluated in the same way as described in the previous section.

In summary, the bulk of the computation necessary to support automatic re-evaluation is performed during execution of the stylesheet and the subsequent production of the result document elements. These costs have already been included in the previous test cases and so the cost associated with initiating the re-evaluation is minimal.

10.4.4 Re-Evaluation of Variable Values

Section 10.4.2 looked at the process of re-evaluating a single component, but it did so in the context of simply reproducing the same result. In reality, any re-evaluation would only occur as the consequence of an edit to the document. Changes to the input data do not need any special attention because the input data nodes are directly accessed by the `Instruction` or `Expression` during execution and so any edits will be automatically effected upon re-evaluation. However, instructions that rely on variables or parameters need to be handled differently. We have seen in chapter 8 that references to variables and parameters are added to the current `ProcessorState` object to indicate that they have been used. We have also seen that the result document elements are made aware of the fact that these variables/ parameters have been changed, but because variables and parameters are calculated upon their declaration, these instructions must also be re-evaluated to produce the updated values.

In order to ensure that these values are up to date when an instruction is re-evaluated, the `StackFrame` object into which variable/parameter values are placed is responsible for their checking and potential recalculation. This approach is similar to the way in which the result document nodes themselves behave. Each time a variable or parameter's value is requested, the `StackFrame` object checks the version information of the appropriate `Instruction` and, if necessary, re-evaluates its value before returning it. Clearly, this variable/parameter re-evaluation adds to the re-evaluation cost, and this cost is proportional to the complexity of the value to be calculated. Continuing with documents used in earlier tests, the state-intensive documents used in section 10.4.1, the mean cost of recalculating the basic parameters (`x`, `y`, etc.) to a text component are in the region of 0.010ms. For simple cases such as these the cost is trivial, but it is likely to increase as the variable/ parameter complexity increases.

10.4.5 Adding New Components

The process of adding a new component to the document is detailed in chapter 9, but we consider the costs of the procedure here.

`StyleElement` Construction

The first stage in adding a new component to the document is to create the necessary `StyleElement` objects to represent the `<xsl:call-template>` and `<xsl:with-param>` elements needed to reference the component template. These are then added to the `StyleElement` tree before being compiled. The cost of this procedure is clearly dependent upon the number, and definitions, of the parameters passed to the component template. However, a typical example in which four parameters are provided (`x`, `y`, `width` and `height`) with simple numerical values has an associated construction cost in the region of 0.250ms.

`StyleElement` Compilation

Once the `StyleElement` objects have been constructed, they must be compiled in order to reflect their existence in the executable form of the stylesheet. As described in chapter 5 this is simply a case of calling the `compile(...)` method on the topmost `StyleElement` object that represents the `<xsl:call-template>` instruction. In the case of the constructed `StyleNode` object described above, this compilation cost is around 0.070ms

Initial Evaluation

The final step in adding a new component to the document is to execute the required instruction(s) to produce the output elements in the result document instance. This cost is clearly dependent upon the type of component in question. A table detailing the initial processing costs of a selection of supported component types is given in figure 10.9, and a bar chart comparing the minimum values is shown in figure 10.8. The definitions of each of these component templates can be found in Appendix B.

Component	Min (ms)	Max (ms)	Mean (ms)
Text Area	0.626	12.757	0.689
Image	0.330	6.080	0.371
Rectangle	0.403	6.817	0.450
Ellipse	0.795	7.651	0.853
Star	1.581	8.685	1.672
Barcode	48.907	65.498	50.832

Table 10.9 — Initial evaluation costs for various component templates

The first observation made is the relatively large value for the 'Barcode' component when compared to the other component types. This clearly shows how the processing cost associated with non-trivial document components can be significant. Secondly, when we compare the result shown for the 'Text Area' component with the single component evaluation results (which also utilize the 'Text Area' component) shown in section 10.4 it might initially appear that the results are anomolous. It might be naïvely assumed that the cost of reprocessing a single component when it is already in the document should be the same as when it is first added. However, this is not the case — the initial processing is more computationally expensive. This because when we re-evaluate the component, the execution state is restored from the associated `ProcessorState` object, whereas during the initial processing, the `Instruction` objects that effect the `<xsl:call-template>` and `<xsl:with-param>` instructions must first be evaluated to set the required execution state. The process of state restoration is much faster than that of initially evaluating the various parameter instructions and this is reflected in the differing processing costs.

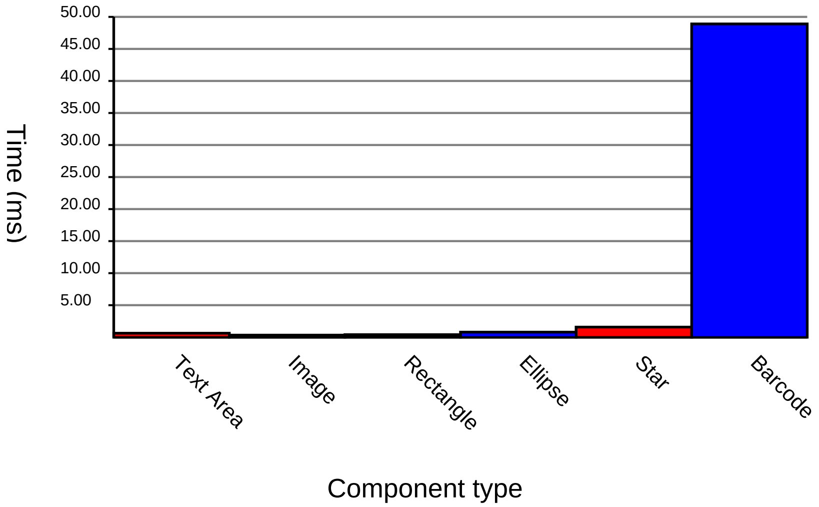


Figure 10.8 — Initial evaluation costs of various component types

10.5 Conclusion

The test cases presented in the previous sections are designed to investigate the relative perfomance of our modified processing framework against a 'vanilla' processor across a range of processing scenarios. It has been shown that the process of storing state, so that partial re-evaluation can be supported, introduces overhead costs that can, in the worst case, result in slower overall execution. However, it has also been shown that in cases where much of the document remains unchanged from one instance to the next, the cost of partially re-evaluating the necessary instructions is much lower than the cost of completely re-evaluating the document. In a production environment, a variety of edits will be performed on the document, each requiring a different proportion of the stylesheet to be re-evaluated. In situations where the document is constructed using component templates as presented in chapter 9, the effects of an edit to the document should, in the majority of cases, be relatively localised, thus avoiding the worst-case scenario described. For example, when editing the document show in figure 9.9, the cost of reprocessing only one of the textual components due to an edit would avoid the cost associated with re-processing the computationally expensive barcodes at the bottom of the page that would be otherwise unavoidable.

The memory cost of storing the necessary state information in such a processing framework is clearly higher than when not storing any state information at all, but the cost of doing so is within acceptable limits. The tradeoff of a relatively small amount of memory for a decrease in re-processing costs is one that is deemed entirely worthwhile in circumstances in which regeneration time is critical to support an interactive editing environment.

The automatic re-evaluation mechanism adds further overheads to the processing of the document, but this is an upfront cost that is only apparent when an updated part of the document is re-evaluated. In situations where a result element does *not* require re-evaluation, there are only negligible overheads introduced by this process.

In summary, our modified processing framework introduces computational overheads in worst-case scenarios, but it offers potentially large benefits in a range of other situations. The re-processing cost incurred as the result of a change to the input data or stylesheet is now dependent upon the proportion of the stylesheet that needs re-evaluating and no longer on the cost of re-processing the entire stylesheet.

Chapter 11:

Conclusions

Having presented the case for a tailored variable data document processing framework, we have discussed a series of processor modifications and methods to support such an editing scheme. Testing of the framework, and analysis of its performance in a demonstration application, has been presented in the previous chapter. We now summarise the findings of this research and consider enhancements to the work, as well as other areas of research that may prove fruitful, but were beyond the scope of this thesis.

11.1 Visually Editing Variable Data Documents

The advent of digital offset presses, and the drive towards ever more personalized content, has led to a publishing paradigm based around variable data documents. This is further reinforced by the number of variable data add-ons for existing document authoring applications, as well as a number of bespoke document preparation systems targetting variable data documents. However, these add-ons and applications typically only offer support for document variation of a limited scope, or are otherwise restricted by the inherent limitations of working alongside an authoring application that was originally designed for the creation of non-variable documents.

We therefore require a document authoring system that is designed specifically to support the full variability required of *true* variable data documents, in which the most extreme cases see one result instance differ from the next in every way possible. The underlying model for such a system comprises a variable data source and a programmtic transformation that generates the various result instances. A simple workflow based around XML, XSLT and SVG has been used throughout this thesis, but more complex solutions using similar technologies, such as the Document Description Framework (DDF) introduced in chapter 1, also exist and provide a more complete solution. The power of these workflows and frameworks come from their extensive programmatic capabilities, but this is also the source of significant usability issues. In order to produce these programmatic documents, the author must have detailed knowledge of the underlying programming language, as well as a good grounding in Computer Science concepts. Unfortunately, the creative professionals that author documents in the real world are not typically skilled in these areas and are more productive when working with a WYSIWYG editing application. Therefore, to allow fully variable documents to be easily created and widely published, there needs to be a way of authoring them through a WYSIWYG editor, as opposed to using a text editor and command-line processing tool.

The requirement for a WYSIWYG editing application is not surprising in a field that has already experienced the desktop publishing revolution of the mid-1980s. Then, the process of document creation and editing was moved from text-based languages, such as $\TeX$ and *troff*, to the graphical editors that were the forerunners to the WYSIWYG applications widely used today. Chapter 3 discussed the problems that affect such editors when we consider the properties of variable data documents and we subsequently proposed a mechanism through which the variability of a document can be exposed and manipulated in a WYSIWYG setting. The exact way in which this should be implemented with regard to interaction with the document author were not considered further since this is an area requiring extensive further research. Instead, we concentrated on the underlying processing framework and associated tools that would be required to support such an interactive editing paradigm.

11.2 Partial Re-Evaluation

Irrespective of the way in which the author interacts with the editing application, the changes made must, ultimately, be effected on the underlying data instance and/or programmatic transform. Once these changes have been made, the document is re-evaluated to generate a new result instance to be displayed to the user. The problem with this approach is that the cost of completely reprocessing the document can be prohibitively high in the context of an interactive editing environment. Therefore, methods of identifying and re-evaluating only the parts of the document affected by the edit were proposed with the aim of decoupling the cost of reprocessing from the size and complexity of the document and, instead, making it dependent on the scope and complexity of the changes made.

To achieve this goal, a mechanism for re-executing only the necessary parts of the transform has been proposed and developed. Given the XML/XSLT-based nature of the document workflows typically employed, an approach using only these technologies was proposed and evaluated, however it was deemed unwieldy and otherwise problematic. Therefore, an alternative approach [67] involving augmentation of the XSLT processor itself, as well as its supporting data structures, was chosen.

The results presented in the previous chapter show that a partial re-evaluation of the stylesheet offers potentially large efficiency gains in documents with low coupling between the component(s) being edited and the rest of the document. The overheads introduced by such a processing scheme are typically outweighed by the reduction in overall processing cost and, in the worst case, where an edit requires the stylesheet to be fully re-evaluated, the cost of storing the processor's state is acceptable.

A method of automatically identifying the parts of the stylesheet that are in need of re-evaluation was also proposed and implemented. The approach taken revolved around the fact that the result document itself had access to all of the information required to determine whether or not re-evaluation was required. Through the use of a further modified custom DOM, the result document is able to request specific re-evaluations to be performed and to replace child nodes within the document with those tree fragments produced by the re-evaluation. This approach also has further advantages with regard to the reduction of unnecessary re-processing. In situations where a change to the stylesheet or input data instance affects parts of the result document that are not of interest to the consuming application, any re-evaluation that is, in principle, required, is not performed. This avoids unnecessary re-processing where many changes are made to the same part of the stylesheet or input data instance, but the affected components are not displayed by the editing application.

11.3 Suitability of XML and XSLT

Throughout this thesis we have concentrated on variable data documents and processing systems that are based around XML technologies — in particular, XSLT and SVG. Although these technologies were primarily chosen because many existing systems are implemented using them, they also offer advantages when we consider their use in the editing context used throughout this thesis.

The first advantage is that of *locality* within documents. Given the tree structure of XML documents, the scope of properties or values defined in whichever tagset we consider, is generally restricted to the sub-trees in which they are declared. In the case of XSLT, this is advantageous in that in-scope variables and parameters can easily be located and their effect is limited to the local subtree. Furthermore, in the case of SVG, this tree structure enforces a hierarchy in which properties affect only the sub-tree of the node on which they are specified, thus providing a form of 'component encapsulation'.

Another major benefit of using XSLT as the underlying processing language is the fact that it supports only single-assignment variables[1]. Therefore, the process of identifying the parts of the stylesheet that are eligible for re-evaluation is made much simpler. Instead of having to identify all of the variables that are in scope, and then track any subsequent re-assignments, we can simply locate their initial (and only) declaration.

Although these properties may be true of other languages and technologies, their presence in the XML-based languages and formats used by many variable data document schemes is a distinct advantage when attempting to build an optimal editing framework.

[1] Parameters can be considered to be variables that are passed between functions or templates

11.4 Future Research

In addition to the research already presented, there are a number of areas that warrant further investigation.

The first topic is concerned with the usability issues involved with editing variable data documents. The work presented has concentrated on the underlying technologies to support interactive WYSIWYG editing of variable data documents, but the ways in which the variability is visually presented to the user, and how they subsequently interact with the document, have not been discussed. This is an important area of research that must be completed if real-world WYSIWYG editors for variable data documents are to become commonplace.

Another related topic stems from the work discussed in chapter 7, in which the production of the switchable input data structure was discussed. It has already been mentioned that the process of merging XML documents into a single instance is not something that was fully considered, and that there is existing literature that could offer insights into the problem. Clearly, this is something that warrants further investigation, but there is also the connected problem of generating valuable metadata during such a merging process.

The values chosen to be represented in the switchable instance, as well as the metadata stored alongside them, are essential when we consider the implications on the how the variability in the document is visually presented to the user as described above. The first task is to ensure that the chosen data represents the full range of the variable values in the data set from which the switchable instance is built. The way in which this is done will depend on the *type* of data in question, but it is also important that relevant metadata is also produced and stored alongside the included values. This metadata is essential information to the editing application when we consider the sort of interactivity and visualizations that may be employed as a result of the usability research discussed above.

In addition to utilizing this metadata to enhance the editing environment, it can also be put to good use during the final processing of the document in which the full set of result instances is produced. The method of speculative evaluation of the document as proposed by Macdonald *et al*[68] relies on such information to indentify 'fastpaths' through the stylesheet that are frequently executed when repeatedly executing the stylesheet. The metadata that we discuss here could conceivably identify the high frequency alternative values and, by extension, be used to identify the 'fastpaths' utilized in speculative evaluation.

In fact, there are a number of processing optimizations that might be considered possible as a result of the work done during the authoring of the document. Given that the editing process must indentify the parts of the document that rely on variable input data, the components produced in the results document could be tagged as either invariant or variant, with those that are invariant cached or otherwise optimized. This might be achieved through a low-level mechanism similar to that of `<REUSABLE_OBJECT>` elements defined by PPML [69], or at a higher level through component isolation schemes such as the COG model proposed by Bagley *et al* and extended to SVG documents by Macdonald *et al*[70].

This idea of discrete document components is one that was introduced in chapter 9 and this link needs to be further investigated. In addition to the 'result-level' components presented by Bagley *et al*, a comprehensive framework for 'source-level' components could be produced based on the initial ideas presented earlier in this thesis. A framework mirroring the class/object relationship in object-oriented programming could provide a way of futher increasing the locality within the stylesheet making the selection and re-evaluation of parts of the document easier and more efficient. As an extension to this idea, a tool for creating these component types from existing stylesheets would be a valuable asset, in the same way that 'result-level' COGs can be extracted from existing PDFs using the tools described in [62].

An alternative approach to generating the set of result documents could also be investigated by considering an extension of the partial re-evaluation methodologies described in this thesis. Rather than simply using the partially re-evaluating processor as purely an editing tool, and then returning to a 'standard' processor to produce the set of result documents, a scheme could be devised where the result documents are produced by the partially re-evaluating processor. Such a procedure would first require that the set of input data instances be merged into an, albeit large, switchable instance as described in chapter 7. However, rather than reducing the number of alternatives at each point in the tree, *every* alternative node/value would be included. The document stylesheet would then be processed with the switchable instance representing the first data instance from the original set. Subsequent result instances would then be produced by making the necessary edits to the switchable instance to morph it from one data instance to the next, and performing the required partial re-evaluation. Clearly, the effectiveness of this approach would be dependent on a sensible ordering of the data instances to be used, as well as a mechanism for handling the large switchable input structure that would be necessary to contain all of the data instances from a large data set. However, this approach has the potential to deliver the prospect of an XSLT-based batch-processing framework that is viable in terms of its performance.

In summary, this thesis has presented a novel solution to the problem of efficiently re-processing variable data documents within an editing environment.

Modification of an existing XSLT processor, and development of supporting document structures, has enabled re-evaluation of the document to be restricted to those parts of the underlying XSLT stylesheet that are affected by an edit. An example editing application has been presented, accompanied by a discussion of considerations relating to the document structure and interaction between the editing application and the stylesheet. An analysis of the performance of the processing tools was presented and a gratifying performance speed up was observed. Finally, further avenues of research were suggested including the possibility of extending the work to produce an efficient result document generation framework that could be of great value.

Appendix A:

Saxon Architecture Diagrams

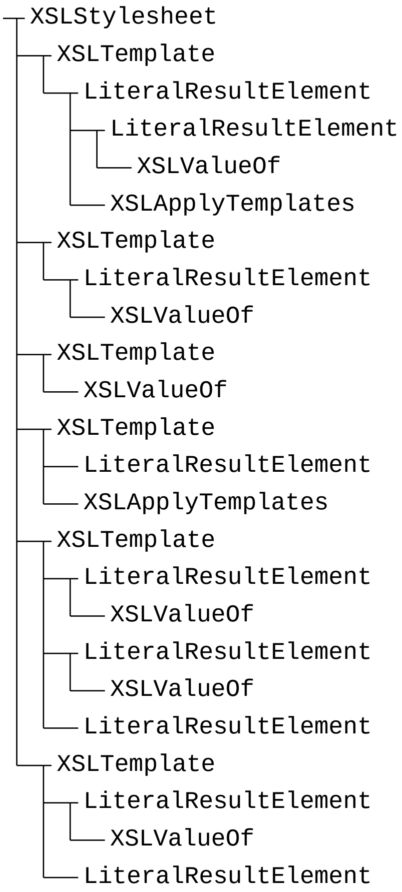


Figure A.1 — Complete XSLT stylesheet object heirarchy

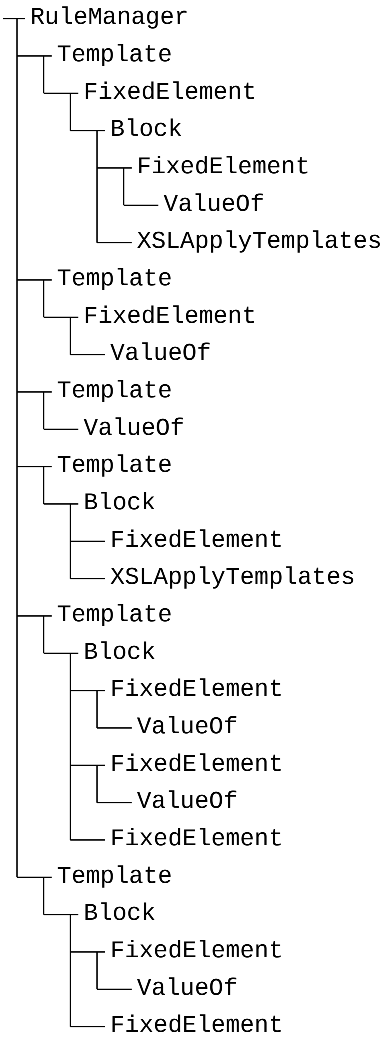


Figure A.2 — Complete compiled stylesheet object heirarchy

Appendix B:

Supporting Program Code

B.1 Example Input Data Document

The code listing given below is an example input document illustrating the structure and markup described in chapter 7.

```
<alt:instance>
   <alt:element localname="person">
      <alt:alternative_attribute localname="forename">
         <alt:value minimun="yes">Bob</alt:value>
         <alt:value frequency="0.4">James</alt:value>
         <alt:value maximum="yes">Christopher</alt:value>
      </alt:alternative_attribute>
      <alt:alternative_attribute localname="surname">
         <alt:value minimun="yes">Qi</alt:value>
         <alt:value frequency="0.3">Smith</alt:value>
         <alt:value maximum="yes">Popadopolous</alt:value>
      </alt:alternative_attribute>
      <alt:element localname="address">
         <alt:alternative_node>
            <alt:value minimum="yes">
               1 Main St.
               Nottingham
               NG1 2AB
            </alt:value>
            <alt:value frequency="0.4">
               123 Brampton Drive
               Stapleford
               Nottingham
         NG9 7YZ
      </alt:value>
      <alt:value frequency="0.25">
         54 Aldridge Close
         Toton
         Beeston
         Nottingham
         NG9 6MN
      </alt:value>
      <alt:value maximum="yes">
         Room C53
         School of Computer Science
         Jubilee Campus
         University of Nottingham
         Nottingham
         NOTTINGHAMSHIRE
         NG8 1BB
      </alt:value>
   </alt:alternative_node>
</alt:element>
<alt:element localname="store">
   <alt:alternative_attribute localname="name">
      <alt:value>Beeston</alt:value>
      <alt:value>Stapleford</alt:value>
      <alt:value>Nottingham</alt:value>
   </alt:alternative_attribute>
   <alt:alternative_attribute localname="map_url">
      <alt:value>beeston.jpg</alt:value>
      <alt:value>stapleford.jpg</alt:value>
      <alt:value>nottingham.jpg</alt:value>
   </alt:alternative_attribute>
</alt:element>
<alt:alternative_node>
   <alt:element localname="offers">
      <alt:element localname="offer">
         <alt:alternative_attribute localname="price">
            <alt:value>£9.99</alt:value>
            <alt:value>£12.99</alt:value>
            <alt:value>£129.99</alt:value>
         </alt:alternative_attribute>
         <alt:element localname="description">
            <alt:alternative_node>
               <alt:value minimum="yes">Assorted beads</alt:value>
               <alt:value>A wide selection of frozen meals.</
                  alt:value>
               <alt:value maximum="yes">A classic toy for young
                   children. A large selection of blocks in
                   varying colours complete with simple vehicles
                   and characters.</alt:value>
            </alt:alternative_node>
         </alt:element>
         <alt:element localname="title">
            <alt:alternative_node>
               <alt:value minimum="yes">Fish</alt:value>
               <alt:value>Kingsmill White Loaf</alt:value>
               <alt:value maximum="yes">Replacement Battery for 15-
                  inch MacBook Pro</alt:value>
            </alt:alternative_node>
         </alt:element>
      </alt:element>
```

```
          </alt:element>
        </alt:alternative_node>
      </alt:element>
    </alt:instance>
```

B.2 Example Component Templates

The following code listings are the contents of the component template files used to compose the documents in chapter 9.

Text Area Template

```
<template name="component-c6e300ed-97b5-470a-9737-4748d1638f3d"
    displayname="Text Area">
  <param name="x" displayname="X"
      editor="testingeditor.property_editors.FloatPropertyEditor"
      default_value="0.0"/>
  <param name="y" displayname="Y"
      editor="testingeditor.property_editors.FloatPropertyEditor"
      default_value="0.0"/>
  <param name="width" displayname="Width"
      editor="testingeditor.property_editors.FloatPropertyEditor"
      default_value="0"/>
  <param name="height" displayname="Height"
      editor="testingeditor.property_editors.FloatPropertyEditor"
      default_value="0"/>
  <param name="content" displayname="Content"
      editor="testingeditor.property_editors.TextPropertyEditor"
      default_value="'Lorem Ipsum'"/>
  <param name="font-family" displayname="Font Family"
      editor="testingeditor.property_editors.FontPropertyEditor"
      default_value="'Helvetica'"/>
  <param name="font-size" displayname="Font Size"
      editor="testingeditor.property_editors.IntegerPropertyEditor"
      default_value="12"/>
  <param name="font-weight" displayname="Font Weight"
      editor="testingeditor.property_editors.FontWeightPropertyEditor"
      default_value="'normal'"/>
  <param name="font-style" displayname="Font Style"
      editor="testingeditor.property_editors.FontStylePropertyEditor"
      default_value="'normal'"/>
  <body>
    <svg:text x="{$x}" y="{$y}" font-family="{$font-family}"
        font-style="{$font-style}" font-size="{$font-size}" font-
      weight="{$font-weight}">
      <xsl:value-of select="$content"/>
    </svg:text>
  </body>
</template>
```

Image Template

```
<template name="component-e02e64c8-a203-4ee7-a7f3-ce8103f9d701"
    displayname="Image">
    <param name="x" displayname="X"
        editor="testingeditor.property_editors.FloatPropertyEditor"
        default_value="0.0"/>
    <param name="y" displayname="Y"
        editor="testingeditor.property_editors.FloatPropertyEditor"
        default_value="0.0"/>
    <param name="width" displayname="Width"
        editor="testingeditor.property_editors.FloatPropertyEditor"
        default_value="40.0"/>
    <param name="height" displayname="Height"
        editor="testingeditor.property_editors.FloatPropertyEditor"
        default_value="40.0"/>
    <param name="src" displayname="Source"
        editor="testingeditor.property_editors.FilePathPropertyEditor"
        default_value="'file://localhost/Users/james/Desktop/Uni/Research/
       PhD/ThesisProcessor/figures/placeholder.jpg'"/>
    <body>
       <svg:image x="{$x}" y="{$y}" width="{$y}" height="{$height}"
           xlink:href="{$src}"/>
    </body>
</template>
```

Rectangle Template

```
<template name="component-9b96ffa3-275a-4c30-b76b-62555a1ed5a4"
    displayname="Rectangle">
    <param name="x" displayname="X"
        editor="testingeditor.property_editors.FloatPropertyEditor"
        default_value="0.0"/>
    <param name="y" displayname="Y"
        editor="testingeditor.property_editors.FloatPropertyEditor"
        default_value="0.0"/>
    <param name="width" displayname="Width"
        editor="testingeditor.property_editors.FloatPropertyEditor"
        default_value="40.0"/>
    <param name="height" displayname="Height"
        editor="testingeditor.property_editors.FloatPropertyEditor"
        default_value="40.0"/>
    <param name="fill" displayname="Fill"
        editor="testingeditor.property_editors.ColorPropertyEditor"
        default_value="'rgb(255, 255, 255)'"/>
    <param name="stroke" displayname="Stroke"
        editor="testingeditor.property_editors.ColorPropertyEditor"
        default_value="'rgb(0, 0, 0)'"/>
    <param name="stroke-width" displayname="Line Width"
        editor="testingeditor.property_editors.FloatPropertyEditor"
        default_value="1.0"/>
    <body>
       <svg:rect x="{$x}" y="{$y}" width="{$y}" height="{$height}"
           fill="{$fill}" stroke="{$stroke}" stroke-width="{$stroke-
          width}"/>
    </body>
</template>
```

Ellipse Template

```
<template name="component-17c9a224-86a4-4fff-a155-88132c6384c7"
    displayname="Ellipse">
```

```
   <param name="x" displayname="X"
        editor="testingeditor.property_editors.FloatPropertyEditor"
        default_value="0.0"/>
   <param name="y" displayname="Y"
        editor="testingeditor.property_editors.FloatPropertyEditor"
        default_value="0.0"/>
   <param name="width" displayname="Width"
        editor="testingeditor.property_editors.FloatPropertyEditor"
        default_value="0"/>
   <param name="height" displayname="Height"
        editor="testingeditor.property_editors.FloatPropertyEditor"
        default_value="0"/>
   <param name="fill" displayname="Fill"
        editor="testingeditor.property_editors.ColorPropertyEditor"
        default_value="'rgb(255,255,255)'"/>
   <param name="stroke" displayname="Stroke"
        editor="testingeditor.property_editors.ColorPropertyEditor"
        default_value="'rgb(0,0,0)'"/>
   <param name="stroke-width" displayname="Line Width"
        editor="testingeditor.property_editors.FloatPropertyEditor"
        default_value="1.0"/>
   <body>
       <svg:ellipse cx="{$x + ($width div 2)}" cy="{$y + ($height div
           2)}" rx="{$width div 2}" ry="{$height div 2}" fill="{$fill}"
           stroke="{$stroke}" stroke-width="{$stroke-width}"/>
   </body>
</template>
```

Simple Barcode Template

```
<template name="component-7064ab3e-0bbd-49f2-a935-7f291c530cde"
    displayname="Barcode (Code 39)">
   <param name="x" displayname="X"
        editor="testingeditor.property_editors.FloatPropertyEditor"
        default_value="0.0"/>
   <param name="y" displayname="Y"
        editor="testingeditor.property_editors.FloatPropertyEditor"
        default_value="0.0"/>
   <param name="width" displayname="Width"
        editor="testingeditor.property_editors.FloatPropertyEditor"
        default_value="40.0"/>
   <param name="height" displayname="Height"
        editor="testingeditor.property_editors.FloatPropertyEditor"
        default_value="40.0"/>
   <param name="value" displayname="Value"
        editor="testingeditor.property_editors.TextPropertyEditor"
        default_value="'1234'"/>
   <body>
       <svg:g transform="{concat('translate(', $x, ', ', $y, ')')}"
           width="{$width}" height="{$height}">
          <xsl:variable name="lut">
             <entry char="0" value="bwbWBwBwb"/>
             <entry char="1" value="BwbWbwbwB"/>
             <entry char="2" value="bwBWbwbwB"/>
             <entry char="3" value="BwBWbwbwb"/>
             <entry char="4" value="bwbWBwbwB"/>
             <entry char="5" value="BwbWBwbwb"/>
             <entry char="6" value="bwBWBwbwb"/>
             <entry char="7" value="bwbWbwBwB"/>
             <entry char="8" value="BwbWbwBwb"/>
             <entry char="9" value="bwBWbwBwb"/>
             <entry char="A" value="BwbwbWbwB"/>
             <entry char="B" value="bwBwbWbwB"/>
             <entry char="C" value="BwBwbWbwb"/>
             <entry char="D" value="bwbwBWbwB"/>
             <entry char="E" value="BwbwBWbwb"/>
             <entry char="F" value="bwBwBWbwb"/>
             <entry char="G" value="bwbwbWBwB"/>
             <entry char="H" value="BwbwbWBwb"/>
             <entry char="I" value="bwBwbWBwb"/>
             <entry char="J" value="bwbwBWBwb"/>
             <entry char="K" value="BwbwbwbWB"/>
             <entry char="L" value="bwBwbwbWB"/>
             <entry char="M" value="BwBwbwbWb"/>
             <entry char="N" value="bwbwBwbWB"/>
             <entry char="O" value="BwbwBwbWb"/>
             <entry char="P" value="bwBwBwbWb"/>
             <entry char="Q" value="bwbwbwBWB"/>
             <entry char="R" value="BwbwbwBWb"/>
             <entry char="S" value="bwBwbwBWb"/>
             <entry char="T" value="bwbwBwBWb"/>
             <entry char="U" value="BWbwbwbwB"/>
             <entry char="V" value="bWBwbwbwB"/>
             <entry char="W" value="BWBwbwbwb"/>
             <entry char="X" value="bWbwBwbwB"/>
             <entry char="Y" value="BWbwBwbwb"/>
             <entry char="Z" value="bWBwBwbwb"/>
             <entry char=" " value="bWBwbwBwb"/>
             <entry char="*" value="bWbwBwBwb"/>
             <entry char="-" value="bWbwbwBwB"/>
             <entry char="$" value="bWbWbWbwb"/>
             <entry char="%" value="bwbWbWbWb"/>
             <entry char="." value="BWbwbwBwb"/>
             <entry char="/" value="bWbWbwbWb"/>
             <entry char="+" value="bWbwbWbWb"/>
          </xsl:variable>
          <xsl:variable name="patterns">
             <token value="{$lut/entry[@char = '*']/@value}"/>
             <xsl:for-each select="1 to string-length($value)">
                <xsl:variable name="char" select="upper-
                   case(substring($value, position(), 1))"/>
                <token value="{$lut/entry[@char = $char]/@value}"/>
             </xsl:for-each>
             <token value="{$lut/entry[@char = '*']/@value}"/>
          </xsl:variable>
          <xsl:variable name="output">
             <xsl:for-each select="$patterns/token">
                <xsl:variable name="token" select="@value"/>
                <xsl:variable name="token_pos" select="position()"/>
                <xsl:for-each select="1 to string-length($token)">
                   <xsl:choose>
                      <xsl:when test="substring($token, position(), 1) =
                          'B'">
                         <svg:rect height="{if (($height - 22) lt 0)
                             then 0 else ($height - 22)}" width="3"
                             stroke="black" fill="black"/>
                      </xsl:when>
                      <xsl:when test="substring($token, position(), 1) =
                          'b'">
```

```
                        <svg:rect height="{if (($height - 22) lt 0)
                            then 0 else ($height - 22)}" width="1"
                            stroke="black" fill="black"/>
                     </xsl:when>
                     <xsl:when test="substring($token, position(), 1) =
                         'W'">
                        <svg:rect height="{if (($height - 22) lt 0)
                            then 0 else ($height - 22)}" width="3"
                            stroke="white" fill="white"/>
                     </xsl:when>
                     <xsl:when test="substring($token, position(), 1) =
                         'w'">
                        <svg:rect height="{if (($height - 22) lt 0)
                            then 0 else ($height - 22)}" width="1"
                            stroke="white" fill="white"/>
                     </xsl:when>
                  </xsl:choose>
               </xsl:for-each>
               <svg:rect height="{if (($height - 22) lt 0) then 0
                   else ($height - 22)}" width="1" stroke="white"
                   fill="white"/>
            </xsl:for-each>
         </xsl:variable>
         <xsl:for-each select="$output/svg:rect">
            <xsl:copy>
               <xsl:variable name="pos" select="position()"/>
               <xsl:attribute name="x" select="sum($output/
                  svg:rect[position() lt $pos]/@width)"/>
               <xsl:for-each select="@* | text()">
                  <xsl:copy-of select="."/>
               </xsl:for-each>
            </xsl:copy>
         </xsl:for-each>
         <svg:text x="0" y="{if (($height - 10) lt 0) then 0 else ($height
             - 10)}" font-family="Courier" font-size="10">
            <xsl:value-of select="concat('*', $value, '*')"/>
         </svg:text>
      </svg:g>
   </body>
</template>
```

B.3 Performance Measuring Code

The following code listing details the contents of the `PerformanceMonitor` class used in the perfomance analysis of the tests presented in chapter 10.

`PerformanceMonitor.java`

```
package net.sf.saxon;

import java.io.PrintStream;
import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.HashMap;
```

```java
public class PerformanceMonitor {

    private Task currentTask;
    private HashMap<String, Task> tasks = new HashMap>String,
    Task<();
    private long startTime;

    private DecimalFormat df = new DecimalFormat();

    private static PerformanceMonitor instance = new
    PerformanceMonitor();

    private PerformanceMonitor() {
        df.setMaximumFractionDigits(3);
        df.setMinimumFractionDigits(3);
    }

    public static PerformanceMonitor getInstance() {
        return instance;
    }

    public void startTask(String name) {
        currentTask = tasks.get(name);
        if (currentTask == null) {
            tasks.put(name, currentTask = new Task(name));
        }
    }

    public void startIteration() {
        startTime = System.nanoTime();
    }

    public void endIteration() {
        currentTask.addIteration(System.nanoTime() -
    startTime);
    }

    public void printResults() {
        System.out.println("Performance Results:");
        System.out.println("======================");
        System.out.println();

        for (Task t : tasks.values()) {
            System.out.println("Task: " + t.name);
            System.out.println("----------------------");
            System.out.println("Iterations - " +
    t.getIterationCount());
            System.out.println("Min - " + t.getMinTime() /
    1000000.0 + "ms");
            System.out.println("Max - " + t.getMaxTime() /
    1000000.0 + "ms");
            System.out.println("Mean - " + t.getMeanTime() /
    1000000.0 + "ms");
            System.out.println();
            System.out.println();
        }
    }

    public void printRawResults() {
```

```
    System.out.println("Raw Performance Results:");
    System.out.println("========================");
    System.out.println();

    for (Task t : tasks.values()) {
        System.out.println("Task: " + t.name);
        System.out.println("----------------------");
        int count = 0;
        for (Long l : t.getRawData()) {
            System.out.println(++count + ": " + l /
1000000.0 + "ms");
        }
        System.out.println();
        System.out.println();
    }
}

private class Task {

    private String name;
    private ArrayList<Long> iterations = new
ArrayList<Long>();

    public Task(String name) {
        this.name = name;
    }

    public void addIteration(long time) {
        iterations.add(time);
    }

    public long getMinTime() {
        long min = Long.MAX_VALUE;
        for (Long t : iterations) {
            if (t < min) {
                min = t;
            }
        }
        return min;
    }

    public long getMaxTime() {
        long max = Long.MIN_VALUE;
        for (Long t : iterations) {
            if (t > max) {
                max = t;
            }
        }
        return max;
    }

    public long getMeanTime() {
        long total = 0;
        for (Long t : iterations) {
            total += t;
        }
        return total / iterations.size();
    }

    public int getIterationCount() {
            return iterations.size();
        }

        public ArrayList<Long> getRawData() {
            return iterations;
        }

    }

}
```

Glossary

Batik An open source Java library produced by the Apache Foundation for displaying and manipulating SVG.

DDF *Document Description Framework*: An XML-based document format designed by Hewlett-Packard Laboratories specifically for the purpose of supporting programmatic variable data documents.

DOM *Document Object Model*: An object-based model for representing XML documents. A model of the document is built as the document is parsed and, once complete, the DOM can be queried or otherwise accessed.

IDE *Integrated Development Environment*: A software application that provides tools and support for software development. An IDE usually comprises a source code editor, a compiler and associated build tools, and a debugger. Advanced features such as Drag-and-Drop user interface builders are also included in some products.

Mis-registration A common problem encountered when printing a document using multiple passes. Because the printing medium is printed onto several times, it is often positioned slightly differently on each pass. Therefore, each printed image from the different passes can be out of position relative to those produced by the other passes. This often results in blurred or overprinted content.

PDF *Portable Document Format*: A final-form page description langauge produced by Adobe Systems Inc. It is based upon the PostScript langauge, but removes many of the programming aspects to leave a declarative document description.

PDL *Page Description Language*: A language that describes the content and appearance of a page in more abstract terms than a final bitmap representation; common examples include PostScript and PDF.

PPML *Personalized Print Markup Language*: An XML-based printer language designed to support efficient variable data printing by allowing reusable content to be identified for caching by the RIP.

PostScript A stack-based programming language developed in the early 1980s by Adobe Systems Inc. Although Turing-complete, it is mostly used as a page description language.

RIP *Raster Image Processor*: A software component in a printing system that is responsible for generating the final bitmap (raster) from an input source, typically a page description in a supported PDL.

Relax NG A schema language for XML in which patterns are used to specify the allowed structure and content of a document. It can be written as an XML document itself, however there is an alternative (non-XML) ‘compact’ version that is also popular.

SAX *Simple API for XML*: An event-based interface for processing XML documents. Events are generated as the document is parsed and appropriate custom handlers are called in order to process the content of the document as it is encountered.

SVG *Scalable Vector Graphics*: An XML-based tag set used to describe vector graphics. Although designed primarily for use in conjunction with XHTML to provide high-quality graphics for web sites, it is also used as an output format for many document processing systems.

Saxon An open source XSLT processor written by Michael Kay. It is written in Java and makes use of JAXP (Java API for XML Processing).

VDP *Variable Data Printing*: A printing paradigm in which multiple instances of the same document yield different result documents depending upon the data provided. The variation can range from simple 'copy-hole' documents to ones in which the entire content is dependent on the variable data.

XML *eXtensible Markup Language*: A meta-syntax for defining other languages. Based on its predecessor SGML (Standard Generalized Markup Language), XML makes document well-formedness easy to check and enforce through a strict grammatical structure.

XPath An expression language used by XSLT and other XML-based technologies. Its primary purpose is that of selecting nodes from a tree structure and filtering the results according to a series of predicates priovided by the user.

XSLT *eXtensible Stylesheet Language for Transformations*: A language designed for transforming XML tree structures from one form to another. Version 2.0 overcomes some of the shortcomings of version 1.0, most notably by providing the ability to transform temporary tree fragments.

Bibliography

All URLs were correct as of 10th May, 2011.

[1] Keith Moore, "Every Page is Different: A New Document Type for Commercial Printing", in ACM *Symposium on Document Engineering*, p. 2, ACM Press, 10th October 2006

[2] "Ipex", `http://www.ipex.org`

[3] John Lumley, Roger Gimson and Owen Rhys, "A Framework for Structure, Layout & Function in Documents", in *Proceedings of the 2005 ACM symposium on Document Engineering*, pp. 32–41, November 2005

[4] "Extensible Markup Language (XML) 1.0", 10th February 1998, `http://www.w3.org/TR/1998/REC-xml-19980210`

[5] "Internationalized Resource Identifiers (IRIs)", January 2005, `http://www.ietf.org/rfc/rfc3987.txt`

[6] "Simple API for XML (SAX)", `http://www.saxproject.org/`

[7] Stephen C. Johnson, "YACC: Yet Another Compiler Compiler", AT&T Bell Laboratories, `http://dinosaur.compilertools.net/yacc/`

[8] Marcus L. Noga, Steffan Schott and Welf Löwe, "Lazy XML Processing", in *Proceedings of the 2002 ACM symposium on Document Engineering*, pp. 88–94, November 2002

[9] "Extensible Stylesheet Language Transformations (XSLT) Version 2.0", 23rd January 2007, `http://www.w3.org/TR/xslt20/`

[10] "Extensible Stylesheet Language (XSL) Version 1.1", 5th December 2006, `http://www.w3.org/TR/xsl/`

[11] Michael H. Kay, "What Kind of Language is XSLT?", IBM developerWorks, 1st February 2001, `http://www.ibm.com/developerworks/xml/library/x-xslt/`

[12] "XML Pointer Language (XPointer)", 16th August 2002, `http://www.w3.org/TR/xptr/`

[13] "XQuery 1.0: An XML Query Language", 23rd January 2007, `http://www.w3.org/TR/xquery/`

[14] Alfred V. Aho, Brian W. Kernighan and Peter J. Weinberger, "The AWK Programming Language", Addison-Wesley Longman Publishing Co., Inc. , 1987,

[15] Donald. E. Knuth, "The TeXbook", Addison-Wesley, 1984

[16] Brian W. Kernighan, "A Typesetter-independent TROFF", in *Computing Science Technical Report No. 97*, Bell Laboratories, March 1982

[17] "GNU Troff (Groff)", GNU Project, `http://www.gnu.org/software/groff/`

[18] Donald D. Chamberlin, "An Adaptation of Dataflow Methods for WYSIWYG Document Processing", in *Proceedings of the ACM Conference on Document Processing Systems*, pp. 101–109, ACM Press, 1988

[19] Kenneth P. Brooks, "A Two-view Document Editor with User-definable Document Structure", Department of Computer Science of Stanford University, 1988

[20] Greg Nelson, "Juno, A Constraint-based Graphics System", in *SIGGRAPH '85: Proceedings of the 12th annual conference on*

Computer graphics and interactive techniques, pp. 235–243, ACM Press, 1985

[21] Adobe Systems Inc., “FrameMaker”,

[22] Eric Laffoon and Andras Mantia et al, “Quanta+”, `http://quanta.kdewebdev.org`

[23] Iréne Vatton, Laurent Carcone and Vincent Quint, “Amaya”, `http://www.w3.org/Amaya`

[24] Richard K. Furuta, Vincent Quint and Jacques André, “Interactively Editing Structured Documents”, in *Electronic Publishing*, vol. 1, pp. 19–44

[25] Microsoft Corporation, “Microsoft Word”, `http://office.microsoft.com/en-gb/word/`

[26] “Office Open XML File Formats (Standard ECMA-376) — 2nd edition”, ECMA International, December 2008, `http://www.ecma-international.org/publications/standards/Ecma-376.htm`

[27] “CatBase”, `http://www.catbase.com/`

[28] XMPie, “uDirect”, `http://www.xmpie.com/`

[29] Hewlett Packard, “Dialogue Live”, `http://welcome.hp.com/country/us/en/prodserv/software/eda/products/dialogue-live.html`

[30] John Lumley, Roger Gimson and Owen Rhys, “Configurable Editing of XML-based Variable-Data Documents”, in *Proceedings of the 2008 ACM symposium on Document Engineering*, pp. 76–85, September 2008

[31] Stefan Wirag, “Modeling of adaptable multimedia documents”, in *Interactive Distributed Multimedia Systems and Telecommunication Services*, vol. 1309, pp. 420–429, Springer Berlin / Heidelberg, 1997

[32] Susanne Boll, Wolfgang Klas and Utz Westermann, “Multimedia Document Models: Sealed Fate or Setting Out for New Shores?”, in *Multimedia Tools and Applications*, vol. 11, pp. 267–279, Kluwer Academic Publishers, August 2000

[33] Dick C. A. Bulterman, "User-centered Abstractions for Adaptive Hypermedia Presentation", in *Proceedings of the sixth ACM international conference on Multimedia*, pp. 247–256, ACM Press, 1998, `http://homepages.cwi.nl/~dcab/PDF/mm98.pdf`

[34] Uriel Jourdan, Cécile Roisin and Laurent Tardif, "Multiviews Interfaces for Multimedia Authoring Environments", in *Proceedings of the 1998 Conference on MultiMedia Modeling*, IEEE Computer Society, 1998

[35] Susanne Boll, Wolfgang Klas and Utz Westermann, "A Comparison of Multimedia Document Models Concerning Advanced Requirements", in *Technical Report*, Universität Ulm, 1999, `http://dbis.eprints.uni-ulm.de/362/1/BKW99c.pdf`

[36] Dick C. Bulterman and Lynda Hardman, "Structured Multimedia Authoring", in *ACM Transactions on Multimedia Computing, Communications and Applications*, vol. 1, no. 1, pp. 80–109, February 2005

[37] "Flash Profressional", Adobe Systems Inc., `http://www.adobe.com/products/flash`

[38] Tien Tran-Thuong and Cecile Roisin, "Structured Media for Authoring Multimedia Documents", in *Series in Machine Perception and Artificial Intelligence*, vol. 55, pp. 293–314, World Scientific Publishing, 2003

[39] Ben Schneiderman, "Designing the User Interface: Strategies for Effective Human-Computer Interaction (3rd ed.)", Addison-Wesley Longman Publishing Co, 1997

[40] Michael Terry and Elizabeth D. Mynatt, "Recognizing creative needs in user interface design", in *Proceedings of the 4th conference on Creativity & cognition*, pp. 38–44, ACM Press, 2002

[41] Manaka Kenji and Sato Hiroyuki, "Static Optimization of XSLT Stylesheets: Template Instantiation Optimization and Lazy XML Parsing", in *Proceedings of the 2005 ACM symposium on Document engineering*, pp. 55–57, ACM Press, November 2005

[42] Ce Dong and James Bailey, “Static analysis of XSLT programs”, in *Proceedings of the 15th Australasian database conference*, vol. 27, pp. 151–160, 2004

[43] Lionel Villard and Nabil Layaïda, “An Incremental XSLT Transformation Processor for XML Document Manipulation”, in *Proceedings of the 11th international conference on World Wide Web*, pp. 474–485, ACM Press, 2002

[44] “Xalan-Java”, Apache Foundation, `http://xml.apache.org/xalan-j/`

[45] Michael H. Kay, “The Saxon XSLT Processor”, `http://www.saxonica.net`

[46] Michael H. Kay, “XSLT and XPath Optimization”, in *Proceedings of XML Europe 2004*, April 2004

[47] Doug Tidwell, “XSLT: Mastering XML Transformations”, O'Reilly, August 2011

[48] Michael H. Kay, “Saxon: Anatomy of an XSLT processor”, IBM developerWorks, April 2005, `http://www.ibm.com/developerworks/xml/library/x-xslt2/`

[49] James A. Ollis, David F. Brailsford and Steven R. Bagley, “Tracking Sub-Page Components in Document Workflows”, in *Proceedings of the 2008 ACM symposium on Document Engineering*, pp. 86–89, September 2008

[50] “XML Schema Part 2: Datatypes Second Edition”, 28th October 2004, `http://www.w3.org/TR/xmlschema-2/`

[51] “The Apache Xerces Project”, Apache Foundation, `http://xerces.apache.org/`

[52] “XML Schema Part 1: Structures Second Edition”, 28th October 2004, `http://www.w3.org/TR/xmlschema-1/`

[53] Organization for the Advancement of Structured Information Standards (OASIS), “RELAX NG Specification”, 3rd December 2001, `http://www.relaxng.org/spec-20011203.html`

[54] Boris Chidlovskii, “Schema Extraction from XML Collections”, in *Proceedings of the 2nd ACM/IEEE-CS joint conference on Digital libraries*, pp. 291–292, ACM Press, July 2002

[55] Boris Chidlovskii, “A structural adviser for the XML document authoring”, in *Proceedings of the 2003 ACM symposium on Document engineering*, pp. 203–211, ACM Press, November 2003

[56] Luuk Peters, “Change Detection in XML Trees: a Survey”, in *3rd Twente Student Conference on IT*, June 2005

[57] Amélie Marian, Serge Abiteboul, Gregory Cobena and Laurent Mignet, “Change-Centric Management of Versions in an XML Warehouse”, in *Proceedings of the 27th International Conference on Very Large Data Bases*, pp. 581–590, Morgan Kaufmann Publishers Inc.2001,

[58] Tancred Lindholm, “A three-way merge for XML documents”, in *Processedings of the 2004 symposium on Document Engineering*, pp. 1–10, ACM Press, October 2004

[59] Sebastian Rönnau, Geraint Philipp and Uwe M. Borghoff, “Efficient change control of XML documents”, in *Proceedings of the 9th ACM symposium on Document Engineering*, pp. 3–12, ACM Press, September 2009

[60] Sebastian Rönnau, Geraint Philipp and Uwe M. Borghoff, “Efficient and reliable merging of XML documents”, in *Proceedings of the 18th ACM conference on Information and knowledge management*, pp. 2105–2106, ACM Press, November 2009

[61] “Batik”, Apache Foundation, `http://xmlgraphics.apache.org/batik`

[62] Steven R. Bagley, David F. Brailsford and James A. Ollis, “Extracting Reusable Document Components for Variable Data Printing”, in *Proceedings of the 2007 ACM symposium on Document Engineering*, pp. 44–52, August 2007

[63] Oracle Corporation, “Java Runtime Environment (JRE) for Windows”, `http://www.java.com/en/`

[64] Brent Boyer, "Robust Java benchmarking, Part 1: Issues", IBM DeveloperWorks, 24th June 2008, `http://www.ibm.com/developerworks/java/library/j-benchmark1.html`

[65] Jiri Sedlacek and Tomas Hurke, "VisualVM", `https://visualvm.dev.java.net/`

[66] Javamex UK, "ClassMexer", `http://www.javamex.com/classmexer/`

[67] James A. Ollis, David F. Brailsford and Steven. R. Bagley, "Optimized reprocessing of documents using stored processor state", in *Proceedings of the 10th ACM symposium on Document engineering,* pp. 135–148, ACM Press, September 2010

[68] Alexander J. Macdonald, David F. Brailsford, Steven R. Bagley and John Lumley, "Speculative Document Evaluation", in *Proceedings of the 2007 ACM symposium on Document Engineering*, pp. 56–58, August 2007

[69] D. DeBronkart and P. Davis, "PPML (Personalized Print Markup Language): A New XML-based Industry Standard Print Language", in *XML Europe 2000*, pp. 1–14, June 2000

[70] 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,* pp. 61–63, November 2005