

Documents as Functions

John William Lumley, MA CEng FIEE

Thesis submitted to the University of Nottingham

for the degree of Doctor of Philosophy

May 2012

ABSTRACT

Treating variable data documents as functions over their data bindings opens opportunities for building more powerful, robust and flexible document architectures to meet the needs arising from the confluence of developments in document engineering, digital printing technologies and marketing analysis.

This thesis describes a combination of several XML-based technologies both to represent and to process variable documents and their data, leading to extensible, high-quality and ‘higher-order’ document generation solutions. The architecture (DDF) uses XML uniformly throughout the documents and their processing tools with interspersing of different semantic spaces being achieved through namespacing.

An XML-based functional programming language (XSLT) is used to describe all intra-document variability and for implementing most of the tools. Document layout intent is declared within a document as a hierarchical set of combinators attached to a tree-based graphical presentation. Evaluation of a document bound to an instance of data involves using a compiler to create an executable from the document, running this with the data instance as argument to create a new document with layout intent described, followed by resolution of that layout by an extensible layout processor.

The use of these technologies, with design paradigms and coding protocols, makes it possible to construct documents that not only have high flexibility and quality, but also perform in higher-order ways. A document can be partially bound to data and evaluated, modifying its presentation and still remaining variably responsive to future data. Layout intent can be re-satisfied as presentation trees are modified by programmatic sections embedded within them. The key enablers are described and illustrated through example.

Relevant Papers

As principal or sole author:

- Lumley, J., Gimson, R. and Rees, O. *A Framework for Structure, Layout & Function in Documents*. ACM DocEng, 2005.[62]
- Lumley, J., Gimson, R. and Rees, O. *Extensible Layout in Functional Documents*. SPIE-IS&T Electronic Imaging, 2006. [65]
- Lumley, J., Gimson, R. and Rees, O. *Resolving Layout Interdependency with Presentational Variables*. ACM DocEng, 2006. [66]
- Lumley, J., Gimson, R. and Rees, O. *Endless Documents: A Publication as a Continual Function*. ACM DocEng, 2007. [64]
- Lumley, J., Gimson, R. and Rees, O. *Configurable Editing of XML-based Variable-Data Documents*. ACM DocEng, 2008. [63]
- Lumley, J. *Automated Extensible XML Tree Diagrams*. ACM DocEng, 2009. [60]
- Lumley, J. *Pre-evaluation of Invariant Layout in Functional Variable-Data Documents*. ACM DocEng, 2010.[61]

As secondary author:

- Macdonald, A., Brailsford, D. and Lumley, J. *Evaluating Invariances in Document Layout Functions*. ACM DocEng, 2006. [70]
- Macdonald, A., Brailsford, D., Bagley, S. and Lumley, J. *Speculative document evaluation*. ACM DocEng, 2007. [68]
- Di Iorio, A., Furini, L., Vitali, F., Lumley, J. and Wiley, A. *Higher Level Layout through Topological Abstraction*. ACM DocEng, 2008. [15]

for

colleagues

with thanks

Judy

with love

Mum & Dad

at long last!

ACKNOWLEDGEMENTS

The author gratefully acknowledges the permission and support of the Hewlett-Packard Company for use of the technology and intellectual property developed as DDF, in HPLabs Bristol over the period 2004-8 and specifically the granting to me of permission to publish this academic thesis, which uses DDF to explore completely new document composition possibilities.

I must thank my colleagues and friends at HPLabs with whom work has been a joy over recent years, especially Roger Gimson and Owen Rees, where between us we tamed the beast we originally named DDF.

Management support from Tony Wiley (and before that Henry Sang) to let me ‘follow my nose’, has been exemplary. Many other colleagues deserve mention in their contribution to various features or posing interesting problems. In alphabetical order they include Alfie Abdul-Rahman, Helen Balinsky, Hui Chao, Xiaofan Lin, Greg Nelson and Mark Restall. Contractors made valuable additions to some of the earlier implementations of the support tools: Philip Fennell, Ian Hoyle, Peter West, Peter Woods and the group at HP Brazil.

Academic collaborators in Nottingham and Bologna used the (partly documented) experimental technology with some publishing success – at Bologna Luca Furini and Fabio Vitali deserve thanks and I must particularly acknowledge the contribution of Angelo di Iorio, who stretched DDF much much further than I anticipated. At Nottingham my PhD students, Alex Macdonald and James Ollis, have used its ideas and challenges to doctoral-level success. Steve Bagley, now my supervisor, showed how it could integrate with his approach to document components.

I am grateful to Nathan Hurst (then of Monash University) for access to his extensive bibliography on document layout.

I am indebted to my supervisors, Dr. Steven Bagley and Prof. David Brailsford for all their support and guidance during the ‘thesis years’ and Prof. Graham Hutton for advice on some aspects of functional programming.

And my partner Judy will finally get her reward for all her patience and long-suffering.

CONTENTS

Part A – Introduction, Context and Prior Art	17
Chapter 1: Introduction	18
1.1 Motivation	19
1.2 Variable documents and document engineering	21
1.3 The context of DDF	24
Tools	26
1.4 Contributions	27
1.5 Thesis outline	29
1.6 Conventions within this thesis	30
1.7 Provenance	32
Chapter 2: Prerequisites	33
2.1 XML	33
2.2 XSLT 2.0	35
2.3 Scalable Vector Graphics (SVG)	40
2.4 XPath	42
2.5 Miscellaneous	46
Chapter 3: Prior Art	47
3.1 Document processors, editors and processes	48
3.2 Separating data and presentation; logical structure	49
DocBook	50
XHTML	51
3.3 Document layout	52
Layout models and solution methods	53
Representations	55

3.4 Variable documents	58
3.5 XSL, XSL-FO and XSLT	59
3.6 Variable document editing	61
3.7 Other functional approaches	63
3.8 Partial evaluation and constant folding	65
<hr/>	
Part B – Prior Art in DDF	68
Chapter 4: Document Description Framework	69
4.1 The ‘life’ of a DDF document	71
4.2 The basic structure of a DDF document	72
4.3 Authoring and editing from the document's range	74
Chapter 5: Functional Implementation	76
5.1 Evaluating the (XSLT) functionality	77
The DDF document ‘compiler’	78
Compiler design	80
External references	83
5.2 Document workflow	84
Chapter 6: Layout in DDF	87
6.1 Extensible layout	88
6.2 Text layout	96
6.3 Advanced layouts	97
6.4 Non-local effects	101
Acyclic dependency and single-assignment presentational variables	103
Cyclic dependencies	107
Post-presentational global effects	109
6.5 Pagination	110
6.6 Conclusion	112
Chapter 7: Example Document – a Travel Brochure	113
7.1 Input data	114
7.2 Processing the input data	116
7.3 General layout model	117
Document background and common sections	118
Construction of pages	119

Providing graphics for a saleItem	123
7.4 Brochure conclusion	125
<hr/>	
Part C – Documents as Functions	126
Chapter 8: Documents as Functions	127
8.1 Definitions	128
Higher-order documents	129
Approximate tree isomorphism	129
Good XML citizen	130
8.2 Variable-data functional semantics	130
8.3 Documents as passive arguments	133
Chapter 9: Variable Layout as a Higher-Order Function	136
9.1 Layout and approximate tree-isomorphism	137
9.2 Layout with embedded function	138
9.3 Attributive layout and embedded program	139
9.4 One-to-many mappings	144
9.5 Foreign namespaces within layout	146
9.6 Modification of the SVG tree	150
9.7 Retained XSLT	153
9.8 Hybrid XSLT/meta-layout action	155
9.9 Conclusion	158
Chapter 10: Partial Evaluation and Constant Folding	160
10.1 Partial data binding	160
10.2 Constant folding of invariant layout	167
Processing attribute sets	171
Presentational variables	172
Results	172
Chapter 11: Active Documents as Variable Data	175
11.1 Simple combinators	176
11.2 Higher-order syntax for DDF	180
11.3 Resource name conflicts	182
11.4 Compound documents	184
Compound inclusion – document ‘imposition’	186

11.5 Conclusion	189
Chapter 12: Example Document – a Medical Record	190
12.1 Data and the document life	191
12.2 Implementation	195
12.3 Conclusion	205
Chapter 13: Discussion and Conclusion	206
13.1 General discussion	208
Robustness and resilience	208
Efficiency	210
13.2 Key findings	210
Universal XML	212
Interspersed namespaces	213
XSLT as the programming model	213
Document compilers	213
13.3 Redesign	214
SVG as presentation and layout	214
Hold data and structure within presentation	215
Use higher-order XSLT and additional compilation	215
Simplify external resource tracking	216
13.4 Other further work	217
User feedback	217
Document type	218
Editing documents and incremental change	218
13.5 Lessons	220
XML and XSLT	220
Correct choices	223
Difficulty, error and disappointment	224
13.6 Conclusion	227
<hr/>	
Appendices	228
Appendix A: Detailed views of main examples	229
Appendix B: Advanced Pagination	234
Appendix C: Compressed display of XML, XSLT and DDF	238

Appendix D: Employment of namespaces	244
Appendix E: Construction of the Thesis	246
Code base statistics	247
<hr/>	
References	249
Papers, Books and Journals	249
Software	256
Standards	256

LIST OF FIGURES

1. A direct-marketing offer	22
2. A general advertising flyer	23
3. Workflow management interface	25
4. Visual document editing	26
5. Main DDF toolchain	27
6. Full and compressed XSLT	31
7. An example XML tree	35
8. An example XSLT program, reversing SVG drawing order	38
9. SVG graphics, before and after reversal by the program of Figure 8	39
10. An example of SVG graphics	41
11. XPath requests from the root of an XML	44
12. XPath requests from context nodes within Figure 11	45
13. XSLT/XPath patterns matching in Figure 11	45
14. Typical workflows for a DDF document	72
15. Simplified XSLT evaluation of DDF spaces	77
16. Typical internal workflow for a DDF document	78
17. Compile & Run	79
18. The DDF documents for the brochure – main, resort & pages	79
19. The XSLT output of the compiler operating on Figure 18	79
20. The result of executing Figure 19 on a data instance	80
21. Compiler design	81
22. Simple document workflow graph	85
23. Complex document workflow graphs	86
24. Presentation declaration for a 4-page brochure	92

25. Final brochure after layout-resolution of Figure 24	92
26. Simple compound flow layout – intentional declaration	92
27. Simple compound flow layout – resulting graphics	93
28. Simple ‘square’ layout & source declaration	94
29. Modifying text texture to enhance contrast against background	98
30. Drop capitals and wrap around images	99
31. ‘Only child’ Layout	100
32. A tree-breaking example: tracking pieces within a flow	102
33. The layout requirements of Figure 32	102
34. Using a presentational variable to recover information from a layout	103
35. Identifying named pieces	107
36. Simple pagination	111
37. Example instance of travel brochure	114
38. A second instance of the travel brochure	114
39. An example customer record	115
40. Example company, resort and map details	115
41. Pre-processing maps	116
42. Canonical form for brochure data	117
43. Brochure page background	119
44. First page	120
45. Main brochure pages for salesItems	120
46. Diagram page	121
47. Customer response page	123
48. Presentations for a saleItem	124
49. The brochure reused in a different field	125
50. Approximate tree isomorphisms	129
51. Good XML citizen transformations	130
52. Named field mapping	132
53. Model projected on data or document	132
54. Simple auto-documentation	135
55. Simple rotational layouts & source declaration	139
56. Element & attribute defined flow layout	140
57. Simple attributive flow layout – resulting graphics	140

58. Attribute defined flow layout	141
59. Simple attributive flow layout – constant result	141
60. Simple attributive flow layout – data-modified graphics	142
61. Layout with two sections of variability	142
62. Two different data bindings for Figure 61 & resulting graphics	143
63. XSLT & layout evaluations for binding test1 followed by test2	143
64. One-to-many pagination layout	144
65. One-to-many layout resatisfaction code	145
66. One-to-many pagination with continuous alteration.	146
67. Three successive bindings to the paginations of Figure 66	146
68. Foreign element aware constructs	148
69. Resatisfying pagination with background templates	149
70. Layout function to support tree replacement	151
71. Original document	152
72. First stage binding and layout	152
73. Replacement of text at second binding and layout	153
74. Four successive bindings of a document	156
75. Programmatic hiding and revelation of graphical content	157
76. Four successive bindings with a data-variable graphic component	158
77. Two-stage binding of a variable brochure	162
78. Retention declarations for main page generators	163
79. Retained variability after company binding	164
80. Simple schemas for brochure data	165
81. Automatically generated retention declarations	167
82. Presentation for the brochure background and second page	169
83. Interspersed layout and program	170
84. Document template with invariant sections identified	173
85. Main document template after invariant processing	173
86. Three separate variable documents	177
87. DDF ‘higher-order’ information flow	178
88. DDF combinator document	178
89. Document evaluations	179
90. A combined document	179

91. Simplified DDF combinator & bound result	181
92. Information flows within ‘higher-order’	182
93. Combination with (styling) attribute sets	182
94. Style conflict in document combination	183
95. Directed resource renaming	183
96. Combined document with style renaming	184
97. Compound application	184
98. Compound application with duplication removed	186
99. Imposition code	187
100. Imposition for saddle-stitch binding with fixed-page argument documents	187
101. An 8-up imposition with variable-page documents	188
102. Sample patient details and medical data	191
103. Medical record after binding patient details	192
104. Pages of the medical record after binding two days’ data	193
105. Record pages after further binding	194
106. Record pages after final binding	195
107. Record with contained data & implementation markers	197
108. Conditionally revealed warnings	198
109. Conditionally altered graphics over three stages of binding	199
110. Embedded before and after evaluated variables	200
111. Constructing the accounts charges page	202
112. Graph generator at three stages during progressive binding	203
113. Continuation points within the document	204
114. Skiing brochure – pages 1 and 3	230
115. Skiing brochure – pages 5 and 8	231
116. Medical record – pages 1 and 2	232
117. Medical record – pages 6 and 10	233
118. A variable width component	235
119. Conditional page templates	237
120. Full and compressed XSLT	239
121. Full DDF	240
122. Compressed display of Figure 121	241
123. Horizontal tree display of Figure 121	241

124. Vertical tree display of Figure 121	242
125. Highly-compressed tree display of Figure 121	242
126. Substitutions for condensed XML	243

PART A
INTRODUCTION, CONTEXT
AND PRIOR ART

Chapter 1

Introduction

This chapter introduces the background and structure of the thesis, which describes and develops a research framework for variable documents known as *DDF*. The business context for automatically generated documents is surveyed briefly, a synopsis of the operational (software package) context for DDF is presented and the major contributions of the research are listed. The three-part structure of the thesis, its conventions and provenance are discussed.

In the past 20 years, *variable data documents*, documents that are constructed automatically with customisation or variation in their content and presentation, have become an important part of communication from businesses to their customers. The technologies for definition, generation and delivery of these documents have been influenced by many factors, not least of course the Web, but also by developments in printing.

This thesis uses ideas from *functional programming* as a part of such document technology. It is based on a research framework, the *Document Description Framework* (DDF), developed by the author and two principal colleagues at Hewlett-Packard Laboratories, Bristol between 2004 and 2008, which emphasised viewing a document as a ‘function’. DDF was designed to explore flexible and easily extensible variable documents, based on a highly functional model, exclusively using XML technologies. It was used for research by that team, in collaborations with academic partners and by my two PhD students at the University of Nottingham.

Details of the architecture are described, along with the reasoning behind it, possibilities that can be exploited from treating ‘document as function’ and conclusions about the overall research. Work on DDF has already been published in a series of papers[60, 61, 62, 63, 64, 65, 66] but this adds to the breadth and depth of those, as well as showing further possibilities.

This is a thesis in *document engineering*, not functional programming – emphasis is on building documents that behave in a more function-like manner, rather than developing an architecture for documents based entirely on functional languages.

The title "Documents as Functions" was chosen deliberately to emphasise the nature of documents that vary in response to some data input (as opposed to editing input). As will be shown, this ‘functional’ nature can be exploited in several, often surprising, ways beyond a simple evaluation.

1.1 Motivation

The work described in this thesis is fundamental research in document representation, exploiting the *tree* as a critical structure in document engineering: as the representation of a digital document itself, its internal logical structures, its presentational definitions and resolved visual form and programmatic definition of any variation of the document as a function of data binding. The aim of the research was to explore and demonstrate the boundaries of the possible rather than devising ‘user friendly’ interfaces that obscure the fundamental issues.

The original motivation arose from developments in digital printing technologies that made high-quality ‘every-page-different’ document printing possible at high production rates and affordable costs, the background to which is discussed in the next section. The question was how such documents should be described to maximise their utility, given that individual instances had to be generated completely automatically, but robustly. The main goal was *extensibility*, both in the types of documents (reports, brochures, photobooks...) that could be described and how ‘variable’ they could be. Equally important, as the data volumes at bitmap level are immense, are methods to reduce runtime computational loads, either by pre-evaluation or exploiting result reuse.

Any study of digital documents will soon show that the *tree* is the main organisational structure – either in terms of presentation (pages, groups, flows) or in logical relationships (group-

ings, sequences). It provides a natural representation for locality (the sub-tree), sequence (sibling order), ‘ownership’ (parent-child) and similarity (tree isomorphism). Components and sections are arranged together in ‘scopes’ which most naturally fall into tree structures. These might be visual scopes, such as within a page, where for example quality requires all the body text to have the same line spacing, or in grouping scopes, such as a set of brochure items all containing the same logical components (picture, description, price) displayed in the same relative way, but otherwise being presented in a manner such that they are clearly distinct¹. We might therefore expect that many of the techniques developed in computer science for representing and using trees in areas such as compiling, program transformation and language design will be pertinent to document engineering.

Digital documents using a variety of markup ‘tags’ have existed for 40 years or more – these tags can control aspects of final printed appearance directly (font size, position...) or describe some more abstract structure in the document (citation...) or sometimes both. More well-developed tools (*t_roff*, *L_aT_eX*) began to treat their tags as a loose grammar which can be used to impose some degree of tree structure, but neither developed the idea very far. In the absence of clearly separated sections (which trees usually provide) then small partial changes within a document require re-evaluation of the *complete* document from source code through rendering to determine the effect on the final visual form. It is either impossible, or impossibly expensive, to determine whether a given portion of the document can be re-processed on its own safely, i.e. without side-effecting somewhere else.

When the document is *programmatically variable*, as expected for variable data documents, this becomes even more problematic. However, there are three approaches that may hold promise: i) using a programming language that is written in a tree-based syntax and manipulates data (including other programs) in the same syntax, ii) using programming languages that are free from side-effects and with near-functional semantics and iii) representing all the document's definition (program, structure, presentation..) as a strict tree in a common basic syntax.

McCarthy's work on *LISP*[73] showed that it was possible to cover the first two points – a programming language *and* its data could be represented as trees, sometimes deep, sometimes flat. *LISP*'s removal of the destructive assignment operation, to be replaced by deeper bindings of variables within tree-nested scoping opened up the possibility of inferring and prov-

¹This is critical – such visual representations in a brochure can constitute a contract and merging two items together might have unhelpful consequences.

ing that parts of a LISP program would behave in certain ways, have certain effects and retain certain properties, leading to ‘safe’ program transformation.

Developments in Web technology have satisfied the third point, defining an agreed standard meta-syntax for trees: *XML*. Its model is sufficient to describe very extensive propertied trees which can contain interspersed sections defined in different namespaces implying differing semantic treatments. Serialisation makes XML almost accessible to be read and understood by a human. Many XML-based technologies have been defined and some have been implemented extensively. Of most interest here are *SVG* as a geometric layout of drawing primitives and effects and *XSLT* as a tree-transforming near-functional programming language.

By describing documents *completely* in XML and defining their variability as buried XSLT sub-trees, it is possible to isolate, reason about and modify the behaviour of sections of the document in the resulting output, free from side-effects, owing to the ‘binding as single-assignment’ model of XSLT result re-use. Coupling this with a declarative combinatoric model of presentation layout makes it possible i) to approach reuse and optimisation in a principled manner and ii) to generate robust ‘higher order’ variable documents: documents that adapt and modify in response to external data in predictable ways. Both of these are important issues for document engineering.

1.2 Variable documents and document engineering

This thesis is about *variable documents*, but to understand what that means and why they are important, we need to discuss the business context of *document engineering* that developed in the past 30 years, as well as some of the hardware technologies that have acted as enablers.

Document engineering is a term that has been used over the past 15 years, with a variety of meanings, though all are centred around solution of problems (both business and personal) by construction, transmission, management and consumption of documents. Glushko & McGrath, in their book[27], focus on the solution of business process problems through the management of documents mainly as objects. This thesis discusses the inner details of the documents themselves and only tangentially discusses systems of documents.

We can discern two broad classes of variable document, based on their utility:

- Documents that *must* be variable by their nature, such as bank statements, passports and wills.
- Documents that may become more valuable by being variable, such as through *personalisation*, with examples in direct marketing, and up-to-the-minute, such as on-demand newspapers.

Here are a couple of document examples that are, or could be, variable. The first is a direct marketing offer, personalised to the recipient. This document has been generated in two passes – the first on a conventional lithographic offset press in full colour and the second by either an ink-jet or a laser printer pass, which adds the (mostly variable) text. Those variable text areas that can be substituted in an ‘open-ended’ manner and those that *must* include some of the ‘fixed’ content in the substitution are highlighted – specifically the phrase ‘are detailed inside’ in Figure 1 needs to be added to the city data before the composite text is written².

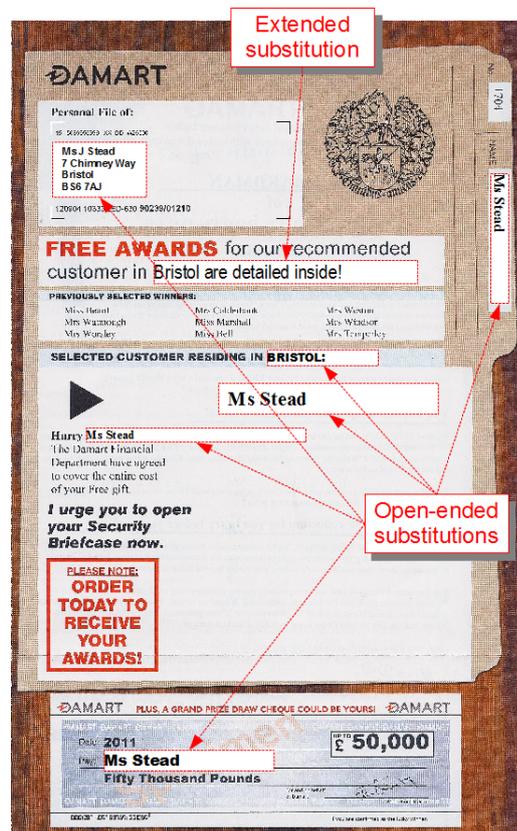


Figure 1. A direct-marketing offer

Figure 2 is a supermarket flyer presenting a series of special offers, including one that is exceptionally seasonal. Whilst this document wasn't variable, and was probably designed ‘by hand’, the type of challenge in document engineering is to build these automatically from a series

²The red boxes are merely to show extent and are not part of the final document

of design paradigms, which are incorporated in the document itself. For example if the number of ‘Gifts’ was 2 or 4, what should happen? Are the correct validities shown for any offer items³ and can they be generated automatically from the data?



Figure 2. A general advertising flyer

Software technologies for construction of such documents are described in more detail in Chapter 3, but in the late 1990s and early 21st century, developments in *digital presses*, by Xerox, Indigo (acquired by HP) and others, enabled the production of offset-quality publications that could be *different on every page* at production costs and rates that were tolerable for high-value marketing material, especially in short runs. Basically the technologies employ laser rendering at resolutions up to 1200dpi, liquid toners and extensive colour gamuts and impression sizes up to A3, at rates approaching 4 pages/sec. Increasing use of extensive customer records and data mining built higher-value marketing propositions, so that personalised communications between business and consumer became more worthwhile, thus increasing demand for variable-data printing.

³The author is aware of a mail-order house where a graphics designer added the ‘Intel Centrino’ (wireless) icon to some laptops on a two page spread, ‘for graphical consistency’ – the company had an expensive operation retrofitting WiFi functionality to many of the machines sold. The image *constituted part of the contract*.



Given these technologies and very flexible ‘data sources’ for material for publication, a consequential business problem is how to design and construct the material to feed these presses, and how to do that effectively, economically and with little or no sacrifice of publication quality and professionalism – the intention after all is to convey a message from business to consumer as clearly and effectively as possible.

The dichotomy is that the data and printing technologies provide the means for generating high print-quality, carefully crafted communications, but the documents cannot be designed on an individual craftsman basis – they must be generated *completely* automatically from data instances. And the problem is how are such documents defined in ways that i) are robust and predictable, as mistakes can be costly both in production and downstream consequences, ii) have a professional look-and-feel to them, iii) are flexible enough to adapt to the limits of the variability in the targetted data, iv) can be generated at rates at least as fast as the presses themselves⁴ and v) are capable of being defined, authored and edited by skilled designers (albeit through visual tools), rather than document engineers.

Similar ideas appear for personalised editions of newspapers[54], where the delivery medium may be web pages or ‘print-yourself’ PDF-based publications. In this case the problem becomes one of selecting and fitting larger-scale stories, features and advertising together to give a publication that is both accurate, effective (for advertisers at least!), pleasing to the eye and reflecting the ‘house-style’ of the publication. Many of the approaches being taken are hybrid, in that a carefully crafted set of templates are designed, with automation choosing which ones are appropriate for the data sets being projected.

1.3 The context of DDF

As mentioned earlier, DDF was designed as a research tool intended to lead to innovative architectures for commercial production of variable documents. The audience for such a system would range from those in document engineering research, through engineers developing document-based solutions to eventually document designers, authors, editors and creators.

The DDF framework comprised a small number of document-processing tools, outlined in the

⁴As will be discussed later, at the highest rates this needs close attention to maximising the use of invariances in the documents, and even techniques such as speculative (eager) evaluation, rather than a (lazy), ‘process everything only when the data is completely bound’ approach.

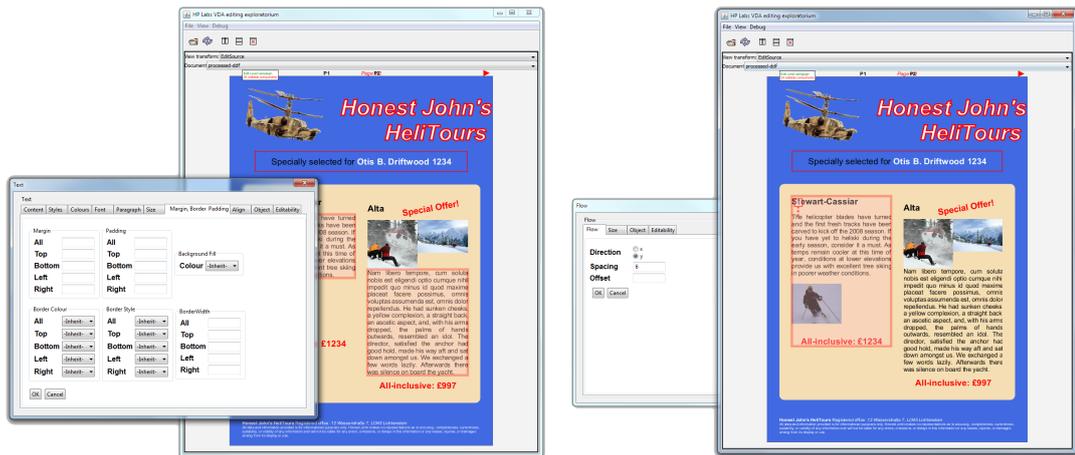


Figure 4. Visual document editing

This tool, which was configurable, was also capable of supporting users with different levels of ‘power’ – higher-level users could adjust what types of editing a lower-level user was permitted on a particular variable document⁶.

The DDF package was intended for research by HP and was used by the team in HPLabs Bristol and two academic collaborators. Apart for some user trials of the editing system, the only developers of documents were engineers familiar with XSLT as a programming language. The general ideas have all been published in the papers cited earlier. Some of the techniques (compiling, context referencing...) are covered by patents, both in application and granted, and the software developed in HP was not, at any time, made freely available, nor is it expected to be⁷.

Tools

Processing a DDF document involves a small number of tools, whose relationships are shown in Figure 5. The three horizontal tools (compiler/evaluator, layout processor and observer) are the main processing chain for binding a document to data and determining the final graphical result for display or printing. Each of these three can be extended in functionality by adding extra XSLT code to support new features or formats.

The workflow processor can schedule the application of this pipeline to documents and data bindings, driven by a declarative definition of the required workflow and data instances. The

⁵A web-based client-server version was also developed for trials. Brief details of this approach are covered in sections 4.3 and 13.5.

⁶See [63] for details.

⁷The author's use of the software for the purposes of research leading to a PhD thesis was permitted by HP under a personal licence.

DDF editor uses the workflow and a definition of ‘editability’ to arrange that final graphical views of a document can be used to edit parts of an original variable document. Most of the emphasis in this thesis is on the compile/evaluate and layout processing actions.

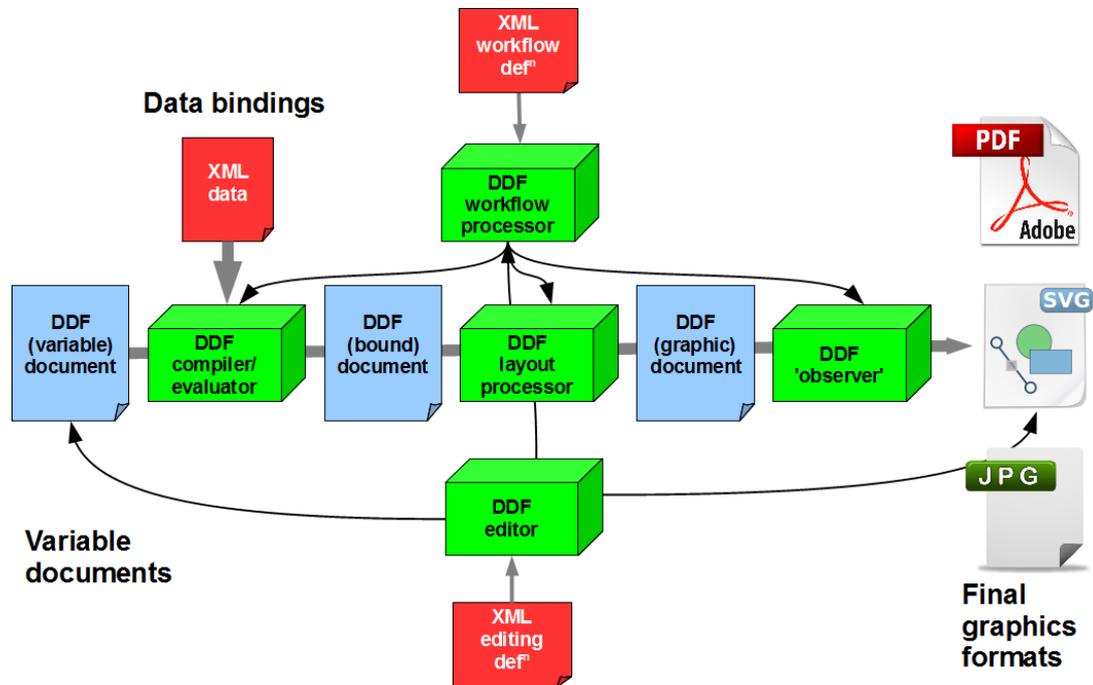


Figure 5. Main DDF toolchain

1.4 Contributions

This thesis, and the work described within, makes the following broad contributions to (or insights in) the field of document engineering:

- An XML-based variable document framework (DDF) that supports research on extensible variable document functionality, both in terms of layout capability and models, and in types of variability that can be defined. It emphasises *declaration* in its document representation, whilst not restricting the complexity of document response that can be defined. This has been demonstrated to work on large-scale documents (this thesis), on variable documents with continual activity, and as a vehicle for document engineering research by the Universities of Nottingham and Bologna, with publication and doctoral success.

- Designing variable documents primarily as functions that construct presentations, rather than as documents with variability added, provides a basis for highly flexible and smoothly extensible architectures. Such architectures are not confined to one particular type of document (e.g. a report) but can support many types from combination and reuse of document-borne function.
- Using XML as the exclusive representation of document, data, presentation and program is exceptionally valuable. Its tree representation provides suitable locality and grouping, appropriate for structure and presentation; structures in separate namespaces can be interspersed effectively, enabling description of hybrid properties in documents.
- A key to combining different XML models within single defining trees is that processing tools should behave as *good XML citizens*, following a general protocol of transferring unknown information in their source trees to equivalent positions in their results. This provides the basis for many critical features, such as hierarchical combination of heterogeneous layouts, higher-order functionality and GUI-based editing of variable documents from presentational instances.
- The XML-based functional programming language XSLT/XPath is highly appropriate both for describing intra-document variability and also as the principal vehicle for implementing processing machinery. Document variability can be placed where it should have its effect – in the correct place in the document; all structures, including the program, are accessed consistently through XPath searches; compilers can create document generators by manipulating source documents, adding necessary contextual features or modifying program to support extended (e.g. higher-order or code-propagation) semantics.
- Defining layout as a tree-aligned set of combinator functions, together with a canonical XML-based representation of presentation results (SVG), enables a very extensive suite of types of layout to be defined, implemented and extended monotonically. Arranging that these combinators are held as attribute declarations on corresponding hierarchical tree nodes provides a basis for re-satisfiable (idempotent) layout. Resolution of layout by a recursive set of pattern-matching agents supports smooth extension of the layout repertoire, as well as supporting internal re-use of partial results and meta-layout operations.

- Variable documents with higher-order semantics can be constructed through a combination of hybrid action between embedded program with compiler-supported code-propagation properties and meta-layout functions. Variable documents can be designed which continue to generate and modify presentation through an indefinitely extended sequence of data bindings.

1.5 Thesis outline

The thesis has three broad sections, followed by appendices and references:

Part A starts by describing the context of variable data documents – where they occur, their different types and scales and some of the applications that can be expected. Pre-requisite technologies are then discussed. Prior art in this area has much overlap with ‘normal’ document production, especially in layout. Both the general area of document definition, layout and production and specific approaches to variability are surveyed.

Part B introduces the design philosophy and goals for DDF, leading to a description of the essence of DDF, how it is used and its anticipated limitations. Subsequent chapters cover the structure and semantics of DDF as a ‘function’ responding to variability in data, implementation of document evaluation and the extensible model for layout. A method of editing such documents is discussed briefly. A detailed example document is then presented.

Part C examines extending the functional nature of the documents. That DDF documents could be extended as *functions* became apparent early on when I realised that the result of a ‘variable binding’ in the framework need not be a fully-grounded document – it could still be a DDF document capable of further variable response. Examples include generation of more specific templates from generic forms (e.g. a specific set of company templates built from a commercial-service generic offering) or documents that continue to respond variably to multiple stages of data binding.

These ideas are discussed in some depth, illustrating indefinitely bound documents with a medical record example. Detailed consideration of the *functional* behaviour and properties of DDF is made, both in terms of variability and its model of layout, leading to discussion of *partial evaluation*, *constant folding* and possibilities of constructing *higher-order* documents.

This final part then draws conclusions and outlines possible redesign of the architecture and

related areas of further work, such as types for variable documents and a document differential calculus.

Small-scale examples will be used throughout the dissertation. Two ‘large’ examples appear in chapters of their own, to illustrate some features in some depth. These are:

- **Travel brochure** (Chapter 7). A conventional ‘record-oriented’ customer publication, which involves substitution of data and potentially unbounded ‘content’ from a record into the publication.
- **Continual medical record** (Chapter 12). A more complex document that adds and alters presentation as it processes input data by stages. It illustrates higher-order use of functional properties.

1.6 Conventions within this thesis

The thesis was itself prepared in DDF and many of the examples are buried within its body, and evaluated as the thesis is generated⁸.

Fonts and styles are used to differentiate different meanings within inline text:

- **code** refers to some piece of defined XML program or data structure. Where needed a namespace prefix may be added, e.g. **fo:block**.
- Angle brackets `<>` around XML element tags within text are normally omitted unless content is required.
- Attributes are often identified with the XPath shortcut `@` e.g. `@fill="red"`.
- Product and software names, e.g. *DialogueLive*, *XPath*, are italicized at their first appearance. For the most common (e.g. XSLT) subsequent references are in normal font.

Key findings and lessons deriving from points in a discussion will be represented in the text by this construct.

In the PDF version of the thesis all references, cross-references and external hyper-links (URLs) are presented as hyper-links, including tables of contents and figures. Bibliographic references

⁸See Appendix E for details.

also present a ‘hover’ of the authors and title. References are sorted by classes (papers, standards etc.) and alphabetically by authors.

There will be many examples of XML-based data structures and XSLT program fragments. Some will be displayed in serialised forms, some with sections elided and some as graphical trees, especially when discussing layout descriptions. Usually sections in different namespaces are presented in different colours – the colour mapping is consistent throughout the thesis.

Displaying large XML structures (including XSLT program) in fully-serialised form takes valuable thesis real-estate and can be difficult to read, leading to a sense of “I can’t see the code for the angle-brackets”. Where there is no ambiguity, I have used a compressed readable representation, full details of which are given in Appendix C; Figure 6 has samples of full XSLT and the textually-shortened version.

<pre> <xsl:template match="A" mode="m"> <xsl:variable name="var" select="1234"/> <xsl:variable name="parts"> <xsl:apply-templates select="C D" mode="#" #current"/> </xsl:variable> <xsl:for-each select="E"> <xsl:choose> <xsl:when test="count(*) gt 23"> <fo:block > <xsl:value-of select=".,@repeat"/> </fo:block> </xsl:when> <xsl:otherwise> <f/> </xsl:otherwise> </xsl:choose> </xsl:for-each> </xsl:template> </pre>	<pre> match:A mode="m" var=1234 parts= => C D mode="#"current" ∀E : choose when:count(*) gt 23 block val(.,@repeat) otherwise f </pre>
<p>■ xsl: ■ fo:</p>	

Figure 6. Full and compressed XSLT

1.7 Provenance

With a thesis that covers prior art of the author predating the study period for the PhD it is necessary to be clear about how the work is partitioned, as well as the author's role with respect to other research colleagues' contribution.

The text of the thesis itself is entirely my own work and entirely new, save for reuse of a few illustrations from earlier professional papers, and the part on invariant layout pre-evaluation (section 10.2) which is a modest reworking of a DocEng 2010 paper, the work for which (and its publication) was within the study period.

The basic technologies of DDF, described in Part B, date from 2005-8. The detailed brochure example therein is a more extensive reworking during the study period of one used in a previous paper.

The entirety of Part C is novel, especially the critical Chapter 9 which describes the key findings of the PhD in terms of a model of interspersed variability and layout declaration within grounded graphical constructs, all within an entirely XML context. The continuous document example (Chapter 12) is a 'proper' reworking of an example proof of concept that was presented at an earlier conference.

The research proposing and exploring DDF at HPLabs Bristol, before this thesis, was carried out by me and my two principal collaborators – Roger Gimson and Owen Rees. We collectively designed the main architectures and implemented appropriate tools, as well as being joint patentees. I was totally responsible for ideas involved with the DDF compiler, the entirety of the approach and implementation for document layout as well as the key methods for document editing. I was the principal author of *all* the externally published papers on DDF.

Chapter 2

Prerequisites

Four existing software standards play a crucial role in this research: their details are discussed in this chapter in enough depth that their use within the rest of the thesis can be understood. *XML* acts as the underlying representation for almost all data and programs. *Extensible Stylesheet Language Transformations* (XSLT) is the principal programming language used both within documents and the implementation of processing tools. *Scalable Vector Graphics* (SVG) describes the grounded graphical constructs of documents. Both these are XML-based representations and can be interwoven. Finally *XPath* is a language for describing searches around an XML tree, used extensively by XSLT.

2.1 XML

Extensible Markup Language (XML) was originally designed as a meta-syntax for Web-based markup languages, but has since grown to be used for many hundreds of different languages and thousands of data representations. An XML document is a *tree*, with nodes and subtrees, and may be held within memory data structures, or within specialist databases optimised for certain types of query or in a serialised character form for storage on disc or transmission over networks. Common parsing and serialisation engines and access mechanisms such as *SAX* or *DOM* can be used between transmitted and stored XML and in-memory data structures.

Nodes can be of several types, data being held in **element**, **attribute** and **text** nodes (the first two have names, which may be namespace-qualified – referred to as a *QName*), with ancillary nodes being **comment** and **processing-instruction**. The tree structure is defined by the **element** node which can contain **element** and **text** nodes as a strictly ordered sequence of children¹. **text** nodes are Unicode strings which may contain character entities (not least to encode characters, such as ' < ', that are reserved within the XML serialisation). **attribute** nodes have a name and a string-based value and are attached as unordered sets to **element** nodes, where the names must be unique within the set attached to the element².

Using namespaces is intended to permit the mixing of XML representations for different purposes without interference, provided the namespaces themselves do not clash. For example an XSLT program (in the namespace *http://www.w3.org/1999/XSL/Transform*) can contain sections of tree with other syntaxes, such as SVG (namespace *http://www.w3.org/2000/svg*) and XSL-FO (which uses *http://www.w3.org/1999/XSL/Format*). All three of these could freely use elements with the same local name³ but which are clearly differentiated in their separate namespaces. XML makes no restrictions on the string used for the namespace, provided it is properly encoded – there is an optional convention that it might link to a further source of information about the markup space – hence these W3C⁴ examples are HTTP links to web pages, but this has no bearing at all on any XML processing.

Serialisation of the XML tree is a key feature: **element** nodes are represented with angle-bracketed opening and closing tags (**<E>...</E>**) with children in between, properly nested. **attribute** nodes are held in (order insensitive) sequences within the owning **element** head tag (**<E a="A value" b="B value">**). **text** nodes are represented as simple text, with possible character entities (**this text ends with a closing angle bracket >**). Treatment of whitespace can be defined – for most purposes extended whitespace is not considered significant and can be trimmed.

Namespaces are defined with prefix declarations within **element** head tags (**<E xmlns:a="A-namespace">**): the prefix then has scope in that element or its subtrees (**<a:EinA>**) unless

¹**comment** and **processing-instruction** nodes can be in this sequence but are peripheral to the 'data' and are irrelevant to this thesis.

²Certain attribute names are reserved such as anything starting with **xml**; **id** is by convention reserved to use for unique identifiers within a tree.

³E.g. **svg:g**, **fo:g** – actually these three don't have any common element names.

⁴World Wide Web Consortium (www.w3.org)

superseded within the scope of a subtree. A default namespace for untagged elements can be declared with `xmlns="default-namespace"` and has scope within the subtree of the declaration, again unless superseded in some descendant. Finally **element** nodes with no children (elements or text) can collapse their opening and closing tags (`<E a="A"/>` can be used as an abbreviation for `<E a="A"></E>`). Figure 7 shows a simple XML tree and its serial form.

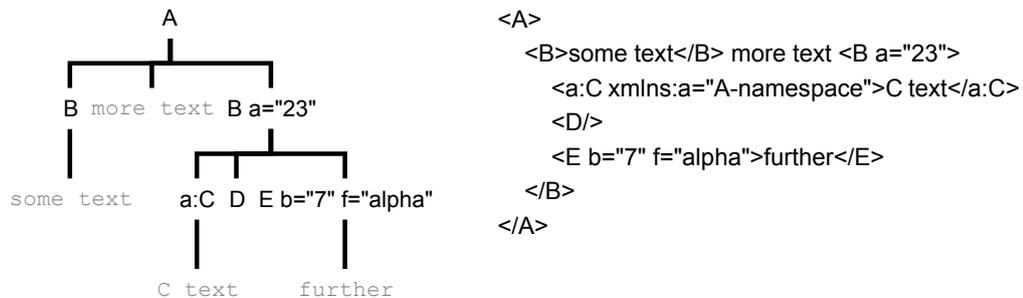


Figure 7. An example XML tree

In essence that is all XML is – a means to represent a tree of named elements, properties (attributes) and text. All the rest of the syntax and semantics is up to the language that exploits it. Languages that use tree structures extensively have a head-start – trees fall out naturally. Others that require structures such as graphs (e.g. *RDF*) will need to represent edges explicitly and use attributes or element text values to identify join points.

2.2 XSLT 2.0

The technology developed in this thesis involves very significant use of *XSLT2.0*[128], a language for transforming XML documents into other XML documents. Discussion of the development of XSLT is given in Chapter 3. This section gives a brief overview of the XSLT processing model as it stood for the 2.0 standard of 2006-8 used throughout this work – for further details the reader is referred to Kay[45].

An XSLT program generally converts one well-formed XML file into another⁵. To do this the language's model is a restricted form of functional program⁶ which operates in both 'pull' and 'push' (pattern-matching) manners and is described as a series of 'templates' that match specific types of XML component, as well as named and parametrised functions. The matches are described as XPath [123] patterns (see section 2.4 for some examples).

⁵Ingestion and emission of plain text is supported but is not the design-focus of the language.

All templates and functions are ‘top-level’, i.e. they cannot contain inner bindings of templates or functions. Templates can operate in partitioned spaces called *modes*, which can include the default mode (**#default** or ‘blank’), a set of named modes (using QNames) or in all modes (**#all**) – the latter is *very* rarely used. Invocations of templates can be in the default mode, in a named mode (**mode="html2svg"**) or continue in the current mode (**mode="#current"**). Well-designed libraries use private namespaced modes for their internal templates.

There is a well-defined system of resolution between multiple matching templates when presented with a specific component in an XML input tree. A ‘winning’ template then has complete control to generate any result it wishes, usually in the form of a tree or components of a tree. These results are yielded to the ‘calling template’, i.e. the template which requested processing of some nodes. Nodes are extracted from the input XML tree through the use of XPath, which in XPath 2.0 can also generate scalar items such as numbers, strings and dates as well as extracted nodes, and handle flat sequences of such as ‘composite values’.

Usually templates operate in a depth-first manner across the input XML tree, building up an output result tree progressively, sometimes by shallow or deep copying of subtrees, sometimes building up new composite subtrees, with or without additional processing. However they have complete freedom to wander around the tree from the ‘context point’ (the node being processed).

Instructions that transfer information from the input tree to result have seven forms: **copy** which is shallow and just makes a copy of the input context node in the result, **copy-of** which does a deep copy of the entire input trees from each of the nodes selected by the instruction (using an XPath expression), **sequence** which returns a *sequence* of nodes and **value-of** which produces a scalar (string) serialisation of the text of the selected nodes, constants and functions thereof. **element** and **attribute** create new nodes in the result (whose name can be computed) and attribute *values* (but not names) in defined result elements can be constructed dynamically using an *attribute value template* where XPath expressions whose values are to be interpolated as strings are surrounded in braces (**height="{2 * \$height}pt"**).

As well as ‘push’ mode templates, XSLT supports two forms of on-demand ‘function’ of similar form but used in different ways – the *named template* is called from XSLT (**call-template name="name"**) with a set of optional and named parameters and inherits the same context

⁶Functions are not entities that can be passed as arguments, though extensions[105] and specific techniques[77, 106] can simulate some similar operations. XSLT2.1/3.0[129] introduces function items as passable arguments to functions as well as lambdas.

as the calling point, as well as the mode for any eventual push applications. XPath can be extended with a *user-defined XPath function*, defined similarly (**function name="ns:name"**) with a sequence of parameters, but which is called from within the evaluation of XPath expressions (**ns:name(args)**). Parameters are bound by argument position, but there is no carry-through of context (including mode) from the calling position – all information must come through the arguments. Functions with the same name but differing numbers of arguments (‘arities’) are permitted. Polymorphic functions are not supported directly, but some techniques involving functions and template can simulate some appropriate behaviours.

XSLT contains the usual control primitive operations which can (only) be embedded in the result tree: iteration (**for-each**) which repeats the tree beneath for each member of a selected sequence setting the context successively to each member, grouping (**for-each-group**), conditionality (**if, choose/when/otherwise**) and recursive application (**apply-templates**) which requests that each member of the selected sequence of nodes be processed further in a push manner. Processing order can be altered by a **sort** declaration, **analyze-string** processes text with regular expressions and critical searches can be indicated by use of a **key** heuristic.

XSLT has single-assignment variables (**variable**) whose naming scope follows the nesting of the tree and can be set to have values that are either calculated from XPath expressions on the ‘current context’ or constructed from contained result trees. Values of variables can be interpolated, tested, iterated over or processed recursively within nesting scope through inclusion in the XPath expressions of instructions. A variant (**param**) can optionally have its value set ‘from above’, permitting extra state to be transmitted ‘down’ the processing stack through parameters to template execution. (In functions this acts as a strict required parameter.) These parameters can have local definition (i.e. arguments to a template) or be defined to propagate a value further down the execution stack until possibly over-ridden (‘tunnelled variables’).

All instructions generating values (**template, function, variable** and **param**) can be typed using data-type models defined in XML Schema[124] (**as="xs:double"**) which can be checked either at runtime or the information used by the compiler for optimization or error checking. The basic types (**xs:integer, xs:anyURI, xs:dateTime** ...) are normally supported by XSLT evaluation engines – schema-aware processors can use extended, user-supplied, type definitions.

The main I/O model consists of two parts: parsing and serialising between XML trees and textual representations, and the use of input and output XML trees – for some applications the

only parsing and serialisations occur at the beginning and end of long processing pipelines, within which only in-memory XML data structures exist.

Since the language is effectively functional (there is no reassignment of values of variables and the I/O model is restricted to straightforward ingestion and writing of complete tree/files, with no re-entrancy between input and output spaces), provided any extension functions used are free of side-effects⁷, then the language can be evaluated lazily. The most complete implementation *Saxon*[105] does this extensively, as well as exploiting tail-recursive properties. Further features of the language being defined in XSLT2.1 support infinite stream processing.

So far we have described the main semantics of XSLT, but not the syntax. This is where it becomes interesting, as XSLT programs are written entirely in XML: instructions are elements in a reserved namespace (<http://www.w3.org/1999/XSL/Transform>). Parameters to those instructions are either attributes (e.g. **group-by="author"**) or specialist child elements in the namespace (e.g. **<xsl:sort>**). All other elements are taken to be part of some result tree.

Figure 8 is an example XSLT program that reverses the order of all SVG elements in a tree (in effect inverting the drawing or Z order), whilst leaving all others intact:

```
<xsl:stylesheet >
  <xsl:output method="xml" indent="yes"/>
  <xsl:template match="/">
    <xsl:apply-templates select="*" mode="reverse-svg"/>
  </xsl:template>
  <xsl:template match="*|@*|text()" mode="reverse-svg">
    <xsl:copy>
      <xsl:apply-templates select="@*|*|text()" mode="#current"/>
    </xsl:copy>
  </xsl:template>
  <xsl:template match="svg:*" mode="reverse-svg">
    <xsl:copy>
      <xsl:apply-templates select="@*,reverse(*|text())" mode="#current"/>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

Figure 8. An example XSLT program, reversing SVG drawing order

It starts with an **output** declaration that controls the serialisation on file output – the output is XML and should be indented. A **template** to match the input document ‘root’ (*/*) follows,

⁷They are also constant-valued within an execution – hence **time()** produces a constant value (defined to be some time that occurred during the execution). **random()** has similar problems to that in other functional programming languages.

which will apply template matching to the child element of the root, whatever that is⁸. These templates will be in the mode **reverse-svg** and the result of the entire program will be the tree that *those* templates produce. There is no copying of the tree by default and the input tree is only ever read, and possibly copied by parts, not altered or rewritten.

Two templates are defined in that mode – the first is a common ‘copy-all’ form, which has a **match** attribute of **@*|*|text()**, i.e. any general attribute, element or text node is copied and then any sub-nodes are processed recursively⁹. The second template (**match="svg:*"**) is the one that does the real work. By default, its match pattern has a higher priority than the ‘copy-all’, so it is used preferentially for elements in the **svg:** prefixed namespace. It copies the element and processes all attributes and then processes child elements and text nodes in reverse order, using the XPath function **reverse()** to invert the sequence. Figure 9 shows sample input and output from this program in both tree and graphical form.

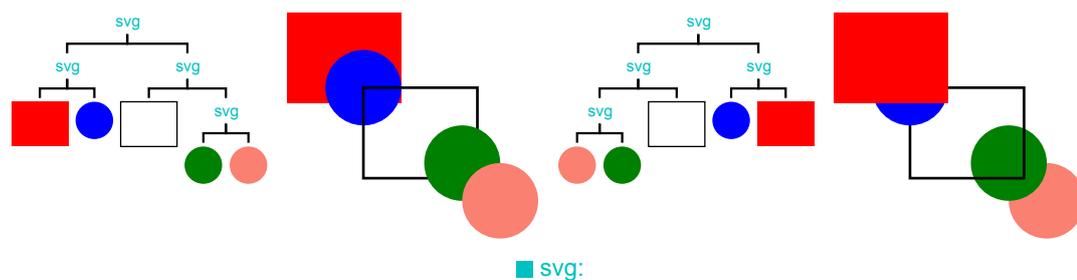


Figure 9. SVG graphics, before and after reversal by the program of Figure 8

Note that this program could operate on *any* well-formed XML structure, not just an SVG file. As it only reverses the children of SVG elements, order is preserved and *all* information is retained within elements in other spaces, so it could be used on an XSLT program that contained, or generated, embedded SVG¹⁰.

The top-level structure of an XSLT program has some restrictions in terms of permitted elements in the XSLT namespace, and has mechanisms to link to other program files, define output formats, encodings and so forth, as well as the set of templates and functions. Other elements in foreign namespaces are permitted at the top level of an XSLT (tree) program – as we shall see later this is beneficial for activities such as documentation and testing.

⁸There will be exactly one if the input is legal XML.

⁹Whilst an **attribute** or a **text()** node will have no attributes, elements or text nodes of its own, it is not an error to attempt to process them – the result of asking for them is a null sequence.

¹⁰Strictly it would need a little more: SVG children within an XSLT generator would need reversing (***[svg:*]**) and **xsl:variable** elements embedded within **svg:*** need careful attention – for implementation reasons variable definitions must precede their references.

But most importantly, XSLT programs can be manipulated just like any other XML trees – there is no problem ‘parsing’ an XSLT program in XSLT (it happens automatically on input) or generating correct syntax output (XML serialisation takes care of that). This power can be used either to analyse (such as documenting, constructing test harnesses, editing...) or to produce new XSLT programs with different properties. It is this latter capability that has been used extensively in the work described in this thesis. To handle the *quoting problem* (embedding XSLT code that is intended to be written into the output), a **namespace-alias** declaration is provided, defining a remapping of selected namespaces when a tree is being constructed, thus enabling static fragments of generated code to be embedded in the program¹¹.

2.3 Scalable Vector Graphics (SVG)

As we are eventually dealing with presentational material in graphical/pictorial form, we need a representation, preferably in a fully XML format. *Scalable Vector Graphics*[119] was chosen to represent all *grounded* layout. SVG was developed from 1999 after experience with other attempts at an XML graphics format[120]¹². It derives its geometry model from the affine user-space mappings of PostScript/PDF. Here we give a very brief overview of SVG, particularly as we have used it, and a couple of ‘proposed extensions’ that have been used, but are currently not part of the main standard.

SVG is a fully XML-based declarative means of describing the drawing of material based on vector, raster and textual graphics. The lowest level atoms are primitive drawing actions (**rect**, **line**, **path**, **ellipse** etc.), image renditions (**image**) and textual strings (**text**, **tspan**). These primitives have a comprehensive set of styling parameters (**fill**, **stroke**, **stroke-width**, **arrows**, **alignment** and so forth) and specific parameters related to positioning and sizing (**x**, **y**, **width**, **height**, **points..**) that control the rendered appearance. Three other important parameters are possible: **transform** which introduces an affine transformation of co-ordinate space between the part's co-ordinates and that of the ‘parent’, **clip-path** which defines a ‘mask’ to be applied to the rendering, and **viewBox** which provides a rectangular scaling window into the child's geometric context. The styling parameters can be attached directly to the elements or inherited from

¹¹For continual documents this ability is essential, sometimes with triple or even quadruple levels of quoting.

¹²More active compiled programmatic/graphic technologies, most noticeably Flash, have recently eclipsed its take-up. However its inclusion in HTML5, some potential halt to Flash development and the rise of the e-book where the EPUB standard supports SVG for graphics [111] holds promise of a welcome renaissance .

parents in the XML tree (or even superposed by a set of CSS styling rules).

Figure 10 shows a simple example in source and rendered form. The outer layer contains a pink rectangle and white ellipse which eclipses the background since being later in document order it is also later in drawing order. This is followed by another group containing a yellow square and two pieces of text, one of which is subjected to a scaling and rotation transform – both text pieces are clipped by the edge of their parent group.

```
<svg:svg width="60" height="60" overflow="hidden">
  <svg:rect fill="pink" width="60" height="60"/>
  <svg:ellipse fill="white" stroke="red" cx="20" cy="20" rx="10" ry="5"/>
  <svg:svg x="20" y="20" width="30" height="30">
    <svg:rect fill="yellow" width="30" height="30"/>
    <svg:text font-family="Courier" font-weight="bold" font-size="6" x="5" y="18">Some text</svg:text>
    <svg:text transform="scale(1.5) rotate(20)" font-family="Courier" font-size="5" x="3" y="3">Rotated text</svg:text>
  </svg:svg>
</svg:svg>
```

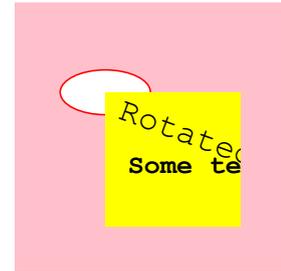


Figure 10. An example of SVG graphics

These primitives are collected together in two types of grouping: **g** which provides a means to apply common styling or transformation to a group of parts (which are obviously the children of the **g** node) and **svg**, which is a translatable group which can have an explicit rectangular extent – by default this acts as a clip-path for the contents¹³.

There are models for resources in the form of **color-space**, **gradient-fill** and **font** which can be attached to suitable parts (or groups thereof). There is also a model for reuse of sub-components (**defs**, **symbol** and **use**) which allows (unparametrised) graphics to be defined once and used in several places, subject to possible geometric transformation.

The text model used is simplistic and does not involve line-wrapping or smart-layout¹⁴ – text manipulation is limited to defining alignment and anchoring, word and character spacing and specific micro-positioning of text spans in block-absolute and predecessor-relative modes.

We chose to use the simple **svg:pageSet** and **svg:page** elements that were proposed in SVG 1.2 but never adopted, to represent pages in our paginated documents.

¹³The parts inside can extend outside this rectangle – ordinarily their display will just be clipped to this extent.

¹⁴At least in the standards of 2005-6. Later proposals are re-examining paragraph-style layouts.

It is worth re-emphasising that SVG **does not perform layout** – a flow of pieces cannot be defined in SVG such that an SVG renderer will line them up beneath each other automatically.

2.4 XPath

XPath[123] is an expressional language used to search within and derive results from an XML tree. As such it is used by XSLT as the means to select nodes within an input XML tree and define patterns against which nodes are matched.

An XPath expression used for search is evaluated at a *context node* within the tree and produces a result, assuming the expression is legal and the tree is valid. In the version used in this thesis, XPath2.0, this result is a (possibly null) *sequence* of items, which can be a variety of atomic types – nodes within the tree, numbers, strings, booleans etc. The sequence is flat, i.e. it cannot contain other sequences.

The expressions contain nine types of construct as well as three additional features supported by XPath executing within XSLT:

- Atomic constants: strings ('**Chamonix**') and numbers (-1, **2.34**).
- Paths to search through the XML tree from the context node, e.g. **resort/pic[1]/@href** which returns all the **href** attributes of the *first* pictures of each resort that's a child of the current context node or **//footnote** which returns every footnote node within the tree, even footnotes buried within footnotes¹⁵.
- Predicates over sequences, e.g. **resort/pic[@height ge 200]** which filters out pictures that don't have a height attribute or for which it is less than 200, or ***[airport][time]** choosing child elements that themselves have both **airport** and **time** children (a non-null sequence result is taken as boolean true for the predicate) or **resort[name=('Chamonix','StAnton')]** which selects either of those resorts (= is nearly 'set intersection' based on values).¹⁶
- Conjunctives on sequences: ,(comma) concatenate, | union (which preserves document order) and bracketing controlling operator precedence. Thus **(@height,150)[1]** gives a default value of 150 if a height is not defined for a piece¹⁷.

¹⁵By default // starts from the tree 'root' – **//footnote** would only find those below the current context

¹⁶The *value* of an element node is the string concatenation of its text nodes and the values of any element nodes, taken in document order and possibly cast to type if required.

- Special ‘functions’ relating to the tree – **position()** yields the position of the context piece amongst its siblings (starting at 1), **last()** gives the number of children the parent has¹⁸ and **root()** is the root of the tree. **element()**, **attribute()** and **text()** match element, attribute and text nodes of the current context. **true()** and **false()** yield constant boolean values.
- Functions on node sequences or other items, e.g. **count(../*)** returning the number of children of a node's parent or **sum(subsequence(amount,2))** which yields the sum of all the **amount** (children), cast to numbers, of the current node, except the first.
- Arithmetic and similar expressions on items that are castable as numeric, e.g. **@width * @height** which yields the area of a node from its presumed attributes, which will be cast from string to numeric.
- Conditionality, e.g. **if(count(resort) gt 1) then 'resorts' else 'resort'**.
- Iterative maps and existential expressions using local variables, e.g. **for \$i in //image return (\$i/@width * \$i/@height)** yielding a list of the areas of all images and **some \$i in //image satisfies (\$i/@width gt 200 or \$i/@height gt 200)** is true if there is a ‘large’ image.
- XPath expressions within XSLT have a small number of extra functions to attach to results from XSLT instructions, notably **current-group()** and **current-grouping-key()** accessing group partitioning and **regex-group(n)** used in regular expression analysis of strings.
- XPath expressions within XSLT can make reference to user-defined functions from the main XSLT program (**ddfl:left-of(\$piece)**) or extension functions added to the XSLT implementation engine (**pic:imageSize(@url)** where **pic:** binds to some resolvable pointer, such as a Java class **java:com.hp.hpl.ddf.picFunctions**).
- Expressions within XSLT can contain *variable interpolations* (**\$main-body**) which refer to the binding of an XSLT variable referenced by that name – scoping follows the XML tree. For example **//book[page-count=\$max-pages]** gives the books which have the largest number of pages, assuming that earlier in the XSLT tree (and in scope) appeared **<xsl:variable name="max-pages" select="max(//book/page-count)"/>**.

The most important of these constructs to understand is the path. A path is made of a sequence

¹⁷This construct appears remarkably often.

¹⁸Hence ***[last()]** being equivalent to ***[position()=last()]** yields the last child element

of steps separated by the solidus (/). Each step starts from the current context node and searches along one of thirteen possible *axes* – the main ones are **child** and **descendant** (working down the tree from the context), **parent** and **ancestor** (working up), **preceding** and **preceding-sibling** (working towards the document start) and **following** and **following-sibling** (towards the end). **attribute** selects attributes of the node and **self** steps onto the context node itself. In practice the vast majority of steps involve **child**, **descendant** or **attribute**. These can be abbreviated, e.g. **pic@height** is equivalent to **child::pic/attribute::height** and **./book** is an abbreviation of **descendant::book**.

At each step nodes are tested along the specified axis – element and attribute names¹⁹ can have namespaces, usually indicated by a prefix – all matching the name are collected (* is a wild-card). These are then subjected to any following predicate tests ([...]) which are conjunctive²⁰. All nodes that pass are then treated in sequence as the context node for the next step, if any. The final set is then returned as the result of that path *in document order*.

Figure 11 is a simple XML tree with both element and tree nodes – most of the leaf nodes have text values, shown in grey. The results of evaluating various XPath queries are also shown, where the context node for each of them is the root of the tree.

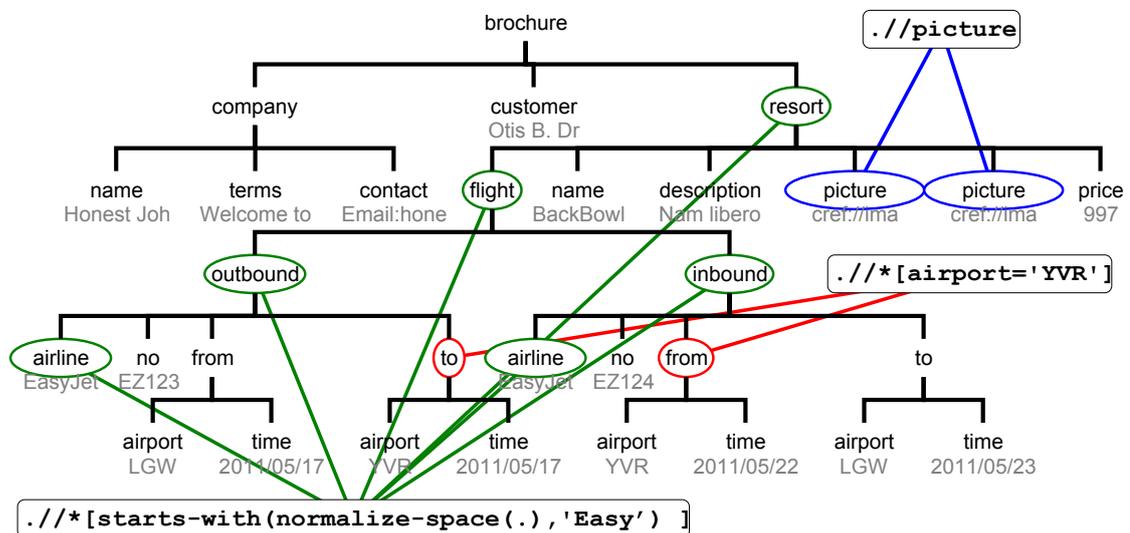


Figure 11. XPath requests from the root of an XML

But the context node can be *within* the tree and usually is in XSLT. Figure 12 shows queries evaluated from various buried context nodes – the context is outlined in grey.

¹⁹Legal names can include - ('hyphen -minus', U+002D) which is a bit disturbing at first.

²⁰In practice the predicate tests can be compiled into the axis search.

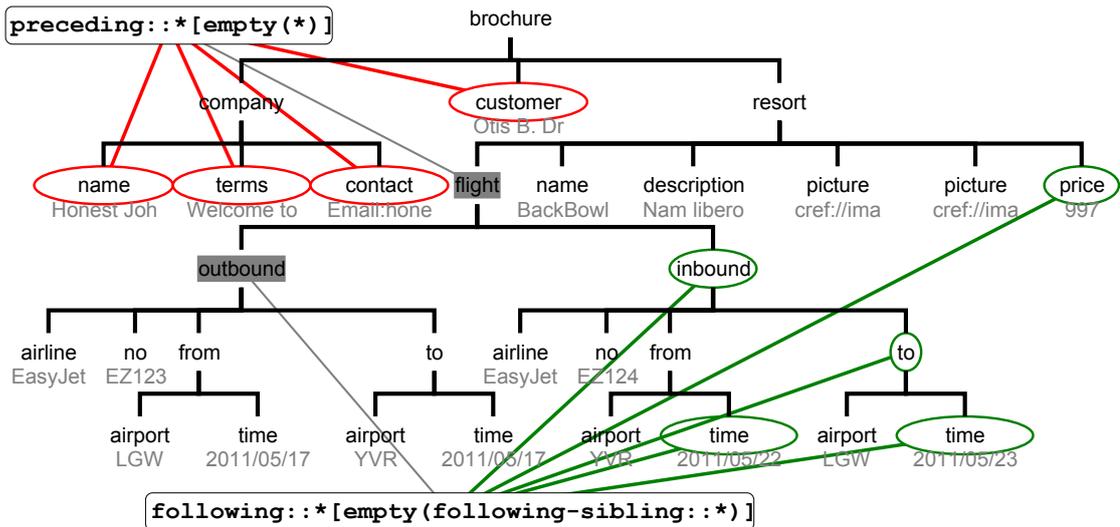


Figure 12. XPath requests from context nodes within Figure 11

When an XPath expression is used as a *pattern* within XSLT, semantics are altered slightly. The pattern can be union of patterns (`|`) and a sequence of steps match the *last* non-predicate step. Thus a pattern `image/@href | picture[@type='JPEG']/ref` matches either a `href` attribute of an `image` element or the `ref` element that is a child of a `picture` element of JPEG type. Figure 13 shows elements in the tree that would match certain patterns

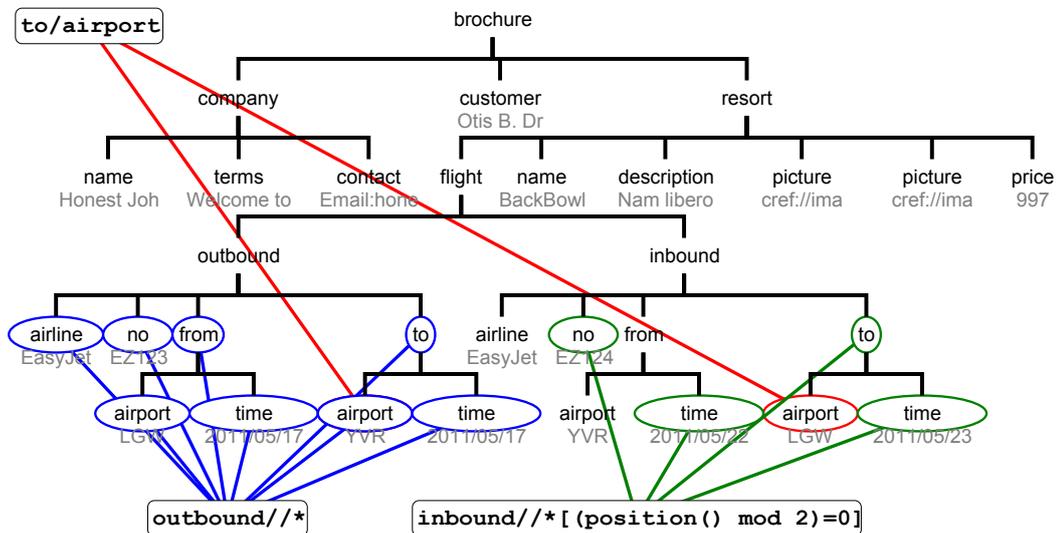


Figure 13. XSLT/XPath patterns matching in Figure 11

It should be apparent that some queries and patterns can be written in several different ways. The patterns `a/b` and `b[parent::a]` are identical, as are the queries `x/y[empty(preceding-sibling::*[following-sibling::*])]` and `x[count(*)=1]/y` (`y` is the only child of `x`). Practical XPath implementations reduce expressions to a canonical form which can be compiled into efficient code depending upon the use [19, 29].

2.5 Miscellaneous

There are a number of other terms that will be used through the thesis that warrant some simple explanation. They are:

- **URI** – Uniform Resource Identifier: a string of characters used to identify a name or resource. URI is used mostly here to define different namespaces (**xmlns:jl="johnlumley.net"** where *johnlumley.net* is the URI). A subset of URI is:
- **URL** – Uniform Resource Locator: a ‘route’ to some resource, either in some relative form (**templates/thesis.ddf**) or some absolute notation (**file://C/thesis/templates/thesis.ddf**). In the latter case the URL starts with a *scheme tag* (e.g. **http:**) which a URI resolution agent will use to decide how to decipher the rest of the ‘address’. Within DDF a specialist scheme (**cref:**) is used for tracking relative resource references through document processing.

Chapter 3

Prior Art

Prior art in document definition, layout and processing is surveyed in this chapter, with a specific bias to those automated approaches relevant to the construction of variable-data documents. Work in functional programming specifically aimed at document generation is also discussed. Effectively this is the relevant prior art that is not directly from DDF: that is covered in Part B of the thesis.

Computer-assisted generation of documents has been a significant part of ‘software engineering’, especially since the desktop PC and desktop-publishing revolutionised the print/publishing industry in the mid 1980s. Some foundations were laid in academia in the 1970s (typesetting etc.) but significant progress in the later 1980s/early 1990s centered on standardisation (Open Document Architecture[110]) and the Web. The Web actually shifted focus away from formalisation and print for some years, as many scrambled to join the ‘gold-rush’ – we still see significant remnants of this in the conflicts between HTML versions, extensions and the ‘browser wars’.

We start by discussing general document processing ideas together with editing, then examine approaches to separating presentation from content and the role of logical structure. Document formats are intricately tied up with models for their layout, which will have implications for variable documents – both semantic and syntactic approaches are reviewed. Techniques for describing variable data documents are surveyed, both in representation of variab-

ility and implications for possible processing. The XSL family is the most important of these for this thesis and its origins are described. Editing and authoring variable documents is more tricky than for static ones and several methods for this will be outlined.

Techniques of document generation using functional programming are then reviewed. As will become apparent later in the thesis, *partial evaluation* of documents is important, both for performance and to support more complex document activity – relevant prior art in this area is discussed and completes the review.

3.1 Document processors, editors and processes

Early systems for document preparation were built as standalone processors that took some source input (with modest contained ‘markup’) and generated some ‘print-ready’ output. Whilst somewhat old (1992), Furuta's review of document preparation systems[20] is still relevant to study of this prior art. The markup was often *ad hoc* and concentrated on paragraph and line boundaries, limited fonting and so forth. The output for the printer was typically a stream of content interspersed with suitable control characters to control micro-spacing. The ‘source documents’ were usually edited with a general purpose editor, i.e. treated as a uniform file of lines of characters.

These standalone processing systems developed, mainly through academic needs, in two directions: increasing functionality and extensions into specialist document domains, and attachment to more sophisticated printing facilities. The former is typified by the development of *Scribe*[87], *Troff*[48] and *TeX*[51] as *de facto* standard markups coupled with the development of extensive suites of ‘add-ons’ and ‘front-ends’ to support specialist application needs such as mathematical printing and complex tables. (*Troff* had a built-in macro system enabling such extensions, *TeX* used front-ends such as *LaTeX*[55] which exploited its macros, traps and diversions very extensively.)

At the same time computer-controlled photo-typesetters and especially the laser printer, gave a much richer output ‘palette’ and specialist formats developed, most notably *Postscript*[109] and then subsequently *PDF*[108]. Each had a rendering model for a set of pages based on affine geometries, clipping, colour spaces and gradients, primitive shapes and text placement, enabling very fine-grain image creation, down to the pixel level. Whilst *Postscript* was a reverse-Polish stack-based programming language, and thus capable of considerable self-extension, it was

replaced increasingly by *PDF*, in part for reasons of security and determinism, and later because of the increasing take-up and investment in *PDF* itself.

The advent of the personal computer and high-resolution windowed displays opened the possibility for document editors to work in WYSIWYG modes, typified by *MSWord* and *Frame-maker* so the document format became internal and proprietary to the tool, but ‘unskilled’ editing could produce semi-professional results. Higher-end versions, such as *InDesign*, concentrated on desktop-publishing, where the models for layout were more complex and much finer control was available to the author/editor. *PDF* became a *de facto* interface between the document as completed by the editing tool and external commercial printers for professional publications.

In parallel, commercial use of document generation processes developed, both in transactional situations (e.g. account summaries, bank and utility statements) and in direct marketing. The volumes involved are considerable and until the advent of the *digital press* printing solutions were typically a hybrid of offset for background and ‘constant’ material, coupled with laser or ink-jet for ‘variable’. Variable data models are usually confined to placing text (and sometimes images) into blank ‘copy-holes’.

Such large scale systems need careful attention to document breakdown and the role of the document in the business process – Glushko & McGrath[27] discuss some of the ‘infomatics’ issues involved. The rest of the thesis generally ignores the business process context and concentrate on document definition, editing and evaluation.

Technologies for the document generation tools can vary from specific application programs to generic systems using a suite of intermediate document formats and standardised processing tools. It is the latter that is my main focus for variable document generation.

3.2 Separating data and presentation; logical structure

Separating presentation and style from data and content has been a significant goal of *document engineers* from the earliest use of computers in document generation. It offers the possibility of significantly increased and robust *reuse* of document components and styles. The volume by André *et al*[2] raises many of the issues being investigated just pre-Web.

Presentational models for documents, such as *PDF*, are principally concerned with describ-

ing *how to draw* the resulting document, potentially in very great detail, as the PDF model of primitives, text, patterns, masking, transformation, transparency and colour spaces shows. It may contain other mechanisms such as grouping, forms, hyper-linking and alternative representations that together make the document richer in final interaction (for example being able to copy long meaningful sequences of text from such a document) but it doesn't inherently describe any of the *structure* of the document – something that makes reuse of parts of such documents easier¹.

Developing a model of *logical structure* for a document introduces some types of common vocabulary for ideas and concepts, such as grouping, sequence, hierarchical containment and detailing, similarity of meaning and so forth.

With a suitable model for structure it is possible to consider generation of an eventual presentation for a document as a mapping from this description into graphics, either directly, or via some propertied model (e.g. nested tables) or by generation of a presentation declaration (flows, packings, sub-assemblies etc.) that then will be interpreted to a final presentation. Changing the mappings can be a route to adaptation (for example between different media, or to differing end-user preferences); extraction of self-contained subsections of the structure can support reuse.

These models for logical structure have tended to focus on tree-based descriptions, where ‘containment’ of one part of structure completely within another is meaningful and natural (and emphasises grouping), though it can be argued that documents that quote highly from others (e.g. review material) often break this[81]. Such models, given their tree nature, are now conveniently described in XML notations, with ‘minor’ referential mechanisms to ‘link across the tree’. A few have developed significantly enough to be regarded as standards: DocBook for book-like publications and XHTML for web pages.

DocBook

DocBook[130] is an XML logical format intended for book-like material which was originally developed as a specialisation of SGML. There has been much recent work on extensive use of XSLT with DocBook[92].

¹Disciplined use of grouping constructs can make it possible to build a PDF with more readily reusable components – see Bagley[7].

DocBook works its way down from a description of a **set** of books, through **book**, **part**, **article**, **chapter**, and **section** with metadata components such as **title**, down to block-level elements such as **para**, **table** and **list** and inline elements, such as **emphasis**, **mathphrase** and **hyperlink**. Elements for linking and cross-referencing (**link**, **xref**) are defined, with a variety of models for resolution of the reference into a textual form. Bibliographic references are supported by **citation** and a family of suitable structures, though it prescribes no particular formats.

Permitted elements are designed to support suitable presentations. For example **section** elements (which are recursively nestable) can be preceded by block-level elements (**para** etc.) but not followed by such. The rationale is that whilst it is possible with general mechanisms, such as headings, to clue a move *into* a deeper level when reading in ‘document order’, is it not possible to clue a move *back out* into the parent level, unless the document uses strict indentation.

XHTML

XHTML [121] developed as an attempt at rationalisation of the HTML[118] ‘soup’ of the late 1990s, and a desire to build an XML-compliant version of the base of the Web. It has had mixed success, HTML4 being rather deeply entrenched, but for our purposes it provides a suitable set of approximate semantics to act as a logical format for report-like documents. It describes a single web page, split into a meta-data section (**head**) and content (**body**). Recent developments in e-book technologies and standards have rekindled interest in the use of XHTML: it is one of the possible internal formats in the EPUB standard[111].

Content includes primitive blocks (**p**, **li**, **img**...), inline modifications (**em**, **br**) and links (**a**). Grouping structures include **section** (which can nest and implies hierarchy), **div** and **span** which group for similar properties outside and inside primitives respectively, **table** for tabular arrangements and **ul**, **ol** and **dl** supporting different types of lists, sequences or sets.

Simple stylistic models are defined for the primitives in layout (border, padding, margins), colour and fonting (family, size, weight, text-alignment etc.) Coupled with a model of hierarchical inheritance of properties and an external styling mechanism (CSS or XSLT), it provides a default mapping into presentation, but need not necessarily be considered constrained by that.

3.3 Document layout

Generated and edited documents need layout – arranging the elements in appropriate sequences and groupings, so that both the message can be perceived clearly and styling requirements are satisfied. We can discriminate between three aspects of such layout:

- What the basic model of layout is, e.g. absolute placement, paragraph filling, columnar and pages, tables, declared constraints and so forth ?
- How the layout can be resolved – what algorithms are suitable, are they convergent, robust and stable, are dependencies cyclic, are results deterministic, are results required to be optimal or just ‘satisfactory’?
- How are layout requirements represented in the document or associated styling?

There is extensive prior art in the geometric construction of documents, particularly around text-based layout. For a comprehensive recent survey of the field, the reader is referred to the excellent review of automated document formatting by Hurst *et al*[38].

Text is usually the predominant item requiring layout, almost always in a rectangular paragraph style. This requires attention to internal detail (*micro-typography* is a good term from Hurst), preserving suitable density, appropriate line-breaking and the avoidance of indesiderata such as ‘rivers of whitespace’. Early work such as Troff[48] took a ‘first-fit’ approach, with limited two-line examination. The critical work on this is Knuth-Plass[52] which treats a paragraph as a single optimisation entity, solved with dynamic programming, using penalties for word-breaking, repeated hyphenation, inter-word spacing and repeated vertical whitespace.

Non-rectangular text containers complicate the issue a little, requiring available line width to be calculated – with mixed-size fonts the interactions become non-linear and possibly non-monotonic[40]. The character edge shapes (**A** vs. **W**) become significant with large font sizes and strongly sloping borders. In extreme cases text containers can be multi-regioned, with complex connectivity (such as embedded object images with close-around flow) – da Silva *et al* [91] show techniques within the context of XSL-FO².

²XSL-FO is described in section 3.5

Layout models and solution methods

At the level above text, simple geometric primitives and images, pieces have to be laid out, usually in pages. There are a wide variety of possible models, from simple absolute positioning to packing, hierarchical composition and flow-based arrangements. In the most general sense there is a large network of constraints: on pieces themselves (usually relationships between width and height) and between pieces, sub-assemblies and the publication context itself, e.g. page size. Such constraints can be of many types, mostly numeric relationships, but they might range in complexity from simple linear forms to non-linear and discrete. And given the possible sizes of the constraint networks, approaches to solutions are critical for any practical use.

The two main models involved are paginated flow systems, where mostly text content is flowed into a series of fixed-size containers within page templates, and hierarchically composed arrangements of components and sub-assemblies, usually considered as rectangular regions. Flow systems generally have a simple deterministic model and solution algorithms, with successive items of content being generated for the ‘next place’ in the flow and then added by parts (lines). Other minor features (footnotes, floats, marginalia and so forth) add a little to the complexity³, with some non-deterministic corner cases, such as a footnote denying space for its associated paragraph, requiring specialist treatment. Multiple intertwining flows that are often found in magazines and web pages can also be problematic – Giannetti [24] investigates approaches using the XSL-FO model with flow mappings.

Hierarchical composition is more challenging and typical of less text-centric documents, such as marketing material and brochures or even some esoteric magazines. Groups of primitive components and compound subgroups are arranged in relative dimensional positions and maintain these during translation, rotation and to varied extents, scaling.

A critical distinction is whether these arrangements are imposed by *imperative actions* at design time (align, distribute etc.) or actively maintained for a document by the environment satisfying *declarative requirements*. Most authoring systems support such imperative actions on a set of pieces within a group or even on arbitrary sets of pieces across groups, but very few maintain the relationship thereafter, which will be vital if some of the components are data-dependent in size, such as variable text or images with varied aspect ratios.

³As long as the floats appear within the text column – when the floating (picture) is able to influence the columns themselves it becomes much more complex – problems that have been investigated by Marriott, Hurst *et al* [37, 72]

It is interesting that the construction of graphical user interfaces (GUI) within programs typically has such declarative support for nested groups of widgets (text entries, check boxes, labels etc.) – as items expand and contract or change visibility due to user interaction or other change of state, the layout is continually recomputed to meet the declared constraints. Generally the definition of such constraints is either built programmatically with library support or with specialist editors that prepare such programs, though layout grammars[53] have been explored to provide a firm foundation. Similar mechanisms are crucial in computer-aided geometric design tools. In multimedia and hypertext there have been experiments on deduction of constraints between display items from rhetorical structures such as sequencing[89].

Commercial systems such as *Dialogue*[103] use a nestable set of ‘containers’ to support such composition, with text containers linkable through named flows. Others such as *InDesign*[98] and *QuarkXPress*[107] using grouping of primitives, text containers and sub-groups, with much emphasis on extensive design-time support, such as grid, alignment and distribution actions.

In print publications, page boundaries are hard outer constraints and often layout involves some ‘packing’ of sub-assemblies. In cases where there are several possible page layout topologies then a linked problem becomes the choice of which layout to use for the ‘next’ set of content. Searching for a solution for the chosen layout can involve many techniques, broadly split into continuous and discrete methods and hybrid uses, employing the topologies for boundaries.

Tables provide a significant set of challenges to layout, being a multi-connected set of individual problems, coupled with global constraints and stylistic demands. In general the problem is NP-hard, but there have been several recent examinations of heuristic optimisation techniques[41] as well as investigations on benchmark optimal solutions [22]. Many higher-level approaches attempt to reduce to a canonical tabular form and use a generic approach to solve such as in Feiner[17]. Similarly, in layout of text where paragraph width is itself a variable (e.g. a paragraph-and-image spanning a column horizontally), the non-linear (and discrete) nature of the text box dimensions requires different solution techniques[36, 57].

Other approaches have been examined based on different search techniques and optimization criteria. Grid-based layouts have benefits for text-dense documents with high rectangularity such as newspapers: Jacobs *et al*[42] is an example which chooses between a variety of gridded templates; automated construction of yellow-page directories has also stimulated much research[30] and even commercial products.

Æsthetic measures such as alignments, size similarities and symmetries have been explored by several investigators[9, 32, 84]. Non-deterministic and stochastic methods such as simulated annealing and genetic algorithms have been used, often with a guillotine-tree partitioned page structure[28], or using a physical force analogy [82], or ‘weight maps’ [58] or a probabilistic model with training sets [13].

At an even more abstract level it is possible to define a document's layout in an under-constrained manner and use a variety of search techniques. di Iorio *et al*[15] showed how ‘topological’ views of layout (similarity, order, grouping, etc.) can be described in a DDF document and evaluated by higher level searching.

Designing a layout so that it expresses the document's structure clearly has similarities with the inverse problem, i.e. document understanding from OCR results – Harrington *et al*[33] discuss the types of layout components (whitespace, group alignment etc.) which can act as effective clues to the document structure and thus form part of objective functions for searches.

Differential layout of documents is usually concerned with *adaptation* to changes in the display environment, usually in terms of page size. In modest cases (A4 vs. Letter page size) preservation of document topology can be successful by altering whitespace margins and possibly extending and repeating patterns, derived from a surface level analysis of the document[11, 12]. More radical changes will require operating from the document's logical structure, either explicitly described or inferred from context – some of the inference techniques are similar to generating a template document from an instance. Most work has typically been on ‘reduction’ (for mobile devices) but some recent work examines the problem when display space increases substantially [75].

Representations

The model for layout has to be represented explicitly or implicitly in the document or some associated ‘styling’ declaration attached. Typically this is defined by the syntax of the representation, which in the case of open XML-based formats means elements in reserved namespaces defining conditions or constraints, with properties usually attached as attributes, and objects of these conditions usually as child trees of the element. This is the approach used in *XSL-FO* and DDF.

An alternative is to split the layout between the document and some associated styling resources

such as in the use of CSS. In this case the document contains the ‘semantics’ of the layout or structure (e.g. **h** is considered to be a text heading, **section** is a collection of grouped content) and the stylesheet controls the parameters and properties for certain of those layouts.

Cascading Style Sheets (CSS)[115] was originally proposed in the mid 1990s as a means of declaratively separating style from content mainly on HTML. It is described as a series of rules which match elements in the document object model (DOM) with simple patterns and ‘classes’ – this is a much simpler match system than XPath patterns used in XSLT. The results of the matches describe stylistic properties of that element (font, colour, borders and other decoration) as well as minor decorative additions (e.g. numbering sequences, preludes and postludes) cascaded with others from earlier matches.

CSS's layout model is particularly sparse, being limited to defining sizes within flows (mostly width in absolute, relative and parent-proportional measures), some absolute positioning and defining pieces as floating, but it lacks any expressional capability – hence interdependent layouts that might be expected in highly-variable documents require external processing. However it *does* encourage separation of presentation (style) from content and can support different page layouts (e.g. sizing) by replacement of the stylesheet. Recent proposals within CSS3 [114] address control of layout with ‘template-based’ positioning (rather than absolute) with named target areas (‘slots’) aligned usually in rows and columns and *layout policies* as properties of CSS rules, which describe which template slot a given element should be placed within. As such this gives a declarative ‘content routing’ effect that in XSLT would be handled by selective expressions or as series of ‘push’ mode templates.

An alternative is to ‘overload’ an existing graphical representation with a computational network that can be triggered on external state changes (which can include the passage of time) and recomputes properties (sizes, positions, even visibility) of the representation with consequential changes to the display. SVG is a very suitable candidate for this and there have been several experiments on active graphical SVG. An integrated system of linear inequality constraints was added to an SVG framework by Badros *et al*[3]⁴. Numerical properties (attributes) of SVG primitive and group elements could be replaced by named variables and a series of declarative expression constraints (equalities and inequalities) included which related these variables and a series of implicit ones (e.g. *rectA.width*). The constraints could be prioritised (effectively into four numeric spaces for weighting the error penalty for failing a constraint

– each space acted as an ‘infinity’ for the space at next lower priority) and the solver had the benefit of always returning an answer. Whilst the solver was efficient (employing modified Simplex techniques), capable of handling large networks of constraints, and able to perform incremental re-computation after external perturbation, the constraints were strictly linear, which doesn't suit the width-height relationship for text.

A simpler *acyclic* series of constraint relationships was developed by McCormack *et al*[74], in a manner similar to that of King on SMIL (see section 3.7) – scalar attributes denoting properties, especially numeric ones, can be replaced by evaluable expressions. These relationships *can* be non-linear, but *must* be acyclic. Pre-compilers convert these expressions into a network of dependencies and thence into a JavaScript program with callbacks attached to ‘context-modifying events’ such as window resizing or mouse-dragging. This has the advantage that only the properties that must be altered as a consequence of the event are actually changed. Macdonald *et al*[69] is similar, using ‘component’ SVG with JavaScript manipulation.

Other graphical representations can be susceptible to the same approach of separating independent components. *Personalised Print Markup Language* (PPML)[113] is designed to describe re-usable ‘objects’ for printing purposes. The objects are compound constructions of graphical pieces (images, PDF pages, SVG graphics etc.) that can be subjected to affine transformations and masking. Documents and sets of documents can be described as combinations of these objects (subject only to translation – thus rendered pixel-maps can be cached and reused without loss of any print quality). Bagley and Brailsford [6] used this representation as a ‘link-editing script’ for document construction – positional results of layout are encoded in the PPML ‘top-level’, using the predefined components.

This approach has been extended to PDF documents when they are ‘componented’ – arranged in self-contained parts (termed COGs). Bagley, Ollis and Brailsford [5, 7, 8, 79] have used this to support building an effective PDF editor, which is the first stage in providing facilities for automatic layout. Pinkney *et al*[83] explore re-flowing content in this system.

⁴The constraint solver used, *Cassowary*, was the one used within DDF.

3.4 Variable documents

Most architectures for documents are designed either as a human-editable form closely linked to the editing tool (*MSWord, OpenOffice, RTF, Framemaker, InDesign, QuarkXPress ...*) or as a markup intended to be edited directly by experts or generated as output from other documenting tools (*TeX, Scribe, HTML ...*) When the requirement is to generate a large set of variable documents from some data set automatically, there are two general approaches using such architectures:

The architecture/tool may have a number of reserved forms for interpolating data into the resulting document. The most common is used for applications similar to *mail-merge*: manual editing adds interpolating fields that relate to named or positional elements from records in a dataset, each record corresponding to a separate document. These fields are represented in similar way to other ‘system’ variables, such as current date or author information. When the document is ‘printed’ from the tool, data (almost invariably text) is interpolated from the record in the dataset and then treated like any other static element.

The data model employed is usually simple – either a regular constant record (e.g. comma-separated variables), or some well-known standard query into a database, this being especially true for mail-merge where contacts are usually extracted from a mail system. There are often limited boolean guards or simple expressions that enable conditional or compound data interpolation and some support of data-conditional inclusion attached to ‘sections’ in the document.

(Cases where the data being interpolated involves *formatted content* can be much more complex to deal with and can only be successful when the content being imported is in an importable format of the document, or more usually the same format as the document itself.)

Alternatively an application is specially constructed to generate markup from data sources, the resulting documents being passed to a final processor to generate ‘readable’ publications. Such is typical of many Web-based server-side variable document systems and transactional applications (e.g. account statements.) The response to data can be very precise, but the adaptability is low – changing the document requires skilful alteration of code.

The design problem with architectures for variable documents is to choose something between these two extremes – high precision *and* good adaptability and reuse. The first can be satisfied with a combination of either a markup language or document format that has a wide

variety of features, which behave in predictable ways, along with a suitable comprehensive model for response to data variability. For example, we should be able to respond to a variable number of ‘special offers’ on a product by altering the layout dependent upon that number – this requires some coupling between the data structure and the presentational declaration.

Adaptability and reuse requires some arrangement of a *template*, or intermediate language, or document format, that deliberately emphasises a document ‘in parts’ and can capture both the declaration of layout and the response to data variability. Alterations can then be confined to operating on the template itself; reuse comes from being able to combine templates and parts thereof together in predictable ways.

Disregarding the specially-written applications, there are three main methods to achieve this: i) overload a static document tool with these interpolating features and some repetitive generator mechanism – effectively the mail-merge solution, ii) build a custom system that is designed from the start for generation from data-sets (as used by *Dialogue*), or iii) develop an architecture of models, standards and implementations from which suitable tools can be built. For this thesis the latter is the approach taken and standards of the XSL family are worthy of specific study.

3.5 XSL, XSL-FO and XSLT

After the Web had started to show some of its potential in the mid 1990s, it became apparent that HTML had insufficient flexibility to handle expected future needs in a number of areas including presentation and layout (CSS was several years in the future). A number of semi-proprietary *de facto* extensions to HTML started to appear, aligned with particular browsers and many Web designs used specific direct styling (e.g. setting fonts) to achieve required results. All this began to ‘undo’ notions of separation of (logical) content from presentation that the SGML roots of the Web tried to emphasise.

The formation of the World-Wide Web Consortium (W3C) began the process of building a series of well-defined consensus standards in the field, starting with the development of the Extensible Markup Language (XML) as the base syntax layer for a lot of other standards. On top of this one of the ‘compounds’ being developed was the Extensible Stylesheet Language XSL[117], describing detailed document formatting based on a rectangular area model and with a programmatic component for the generation of documents from potentially variable data. The

intention was that the stylesheet could be loaded into a browser, along with appropriate data, and the browser could then produce appropriate imagery, thus helping re-establish separation of data and presentation, encouraging standardisation whilst still supporting finer granularity and controllable presentation.

XSL's formatting model was based on supporting both scrollable and paginated documents defined in terms of 'page-master' objects that contained regions and columns to be filled from content streams. These content streams involve hierarchical blocks of bound textual and image content, with properties described as attributes on their XML representation. A lot of work went in to defining a model that could accommodate detailed text formatting (font positioning, text decoration, diacritical marks and other *micro-typography*) and a wide variety of languages⁵. This section was termed *Formatting Objects*.

The *variability* in a document was supported by another orthogonal near-functional programming language *Transformations* that had no side-effects, operated in both pattern-driven (push) and explicit (pull) modes, used a subsidiary language *XPath* to search within XML data, and was represented in XML in a separate namespace to that of the formatting. Since the two parts could be mixed almost arbitrarily in the XML tree, a wide variety of variable documents could be constructed – the transformation code responds to the XML data and generates formatting code that is eventually evaluated to produce final results with bound geometry, such as in PDF, Postscript or image formats.

By the time the standard reached the 'Version 1.1' stage it was clear that the two languages were sufficiently orthogonal that each could stand alone successfully. The formatting became known as *XSL-FO* and the transformation as *XSLT*. XSL-FO has developed gradually adding features such as non-rectangular text areas, but has not really been greatly successful. The format was not intended for direct document authoring, but rather as a 'standard' output form for other authoring tools which could then be used to generate sequences of final documents. It has also been used as one of the content types for PPML.

As XSLT was capable of operating on, and generating, arbitrary XML structures, not just that of formatting objects, it started to be used as a server-side XML→XML transformation tool and subsequently moved to a 2.0 version (with single-assignment variables, static and dynamic typing, user-defined functions, etc.) A final 3.0 version will make it a full functional lan-

⁵Not only L→R and R→L but vertical styles and mixed styles (e.g. L→R quoted in a R→L language)

guage in XML syntax, when functions can be treated as entities (and generated on the fly, unlike in many FP languages where they must be defined at compile-time) as well as having features to support streaming (i.e. infinite i/o) processing.

There is a complementary technology *XQuery*[126] developed alongside XSLT, principally intended for making queries over large-scale sets of XML documents or databases. It shares the use of XPath and can also generate XML output. It is functional as well but does not support template-driven processing and its syntax is *not* in XML but closer to a typical functional language (**for \$act in doc("hamlet.xml")//ACT let \$speakers := distinct-values (\$act//SPEAKER) return...**). Whilst there are many similarities and parallels with functional programming, this thesis does not employ this technology.

For several reasons⁶ the goal of XSLT sitting inside the browser wasn't achieved and both XSL-FO and XSLT are mostly executed by standalone processes. Kay has recently started a fresh attempt, utilising the increasing power of JavaScript within current browsers[46] to make browser-implemented XSLT possible, without the need for *any* disturbance of the platform.

3.6 Variable document editing

Whilst the separation of content and structure from presentation offers engineering advantages as suggested, especially in reuse, there is a major problem in how such documents are edited. The preference is to carry out such authoring and alteration in the 'presentation space' rather than in 'definitional code' – e.g. WYSIWYG rather than coding *LaTeX*. This problem has been a thorny issue for many years – Roisin & Vatton[88] examined how this might be achieved in experimental systems for 'static' documents in the early 1990s. With variable-data documents this becomes even more tricky as by their very nature the documents are *variable* – sections might vary significantly from document instance to instance and how this is described and edited in final presentation is a challenge. There could be two sorts of editing from the final visual presentation of a document instance – altering the specific *data*, such as changing the customer's name, or more usually, altering the appearance of the layout or some of the *static* content of the document. When the layout can be highly programmatic this can be akin to altering a program by operating on the *results* of the execution of that program.

⁶There was a chicken-and-egg situation requiring both XSLT content and browser functionality whilst the browser wars proceeded, though interestingly Microsoft was the one group that rapidly built and deployed XSLT1.0 functionality in Internet Explorer even before the standard was completed.

The most common user-controlled variable-data documents are probably those used for ‘mail merge’ in word processors like *MSWord*, which was discussed briefly earlier. In this case the template is created with reserved structures (often with specialist field codes) to indicate the presence of a variable interpolation point and how the value of the variable should be interpolated. Typically this is then processed using a ‘wizard’ to complete the mapping process and project the output data. These variables are usually only interpolating text, which means the styling is taken from the text in which the reference is written, and the layout (usually a text-flow) is again taken from the context. These wizards might be third-party plug-ins such as CatBase[102] for the *InDesign* class of desktop publishers.

Recent developments in variable data publishing technologies have introduced a small number of products that support a more general solution. *Pageflex*[101] provides standalone and Web-deployed tools for editing presentation material in a grounded-layout and text-flow copy-hole model, with support for editing configuration and a suite of tools. *DialogueLive*[104] supports role-based editing of document instances against a computed variable document model, which is extensible.

Quint discusses the use of structured editing in XML[86], where DTD-driven constructor programs are created on a data instance within the *Amaya* editor. He and his colleagues also approach editing documents where the stylistic and to some extent layout appearance of the document is defined in a separate CSS stylesheet[85]. In this case the CSS stylesheet is analysed to produce an inversion map, to discover *which* part styled a particular result element.

When the document processing is performed by XSLT, theoretical studies of optimized, incremental and reversible XSLT execution are relevant. Several studies [16, 47] consider optimization by static analysis to eliminate dynamic (runtime) search and generating lazy XML parsing – some of these techniques have made their way into practical compilers. Noga, Schott & Löwe [76, 90] use representations of the XML trees that are deliberately designed for lazy processing, showing how an XSLT interpreter can make effective use of it when only accessing few parts of the tree (e.g. creating a top-level outline).

Villard & Layaida [94] describe an approach to incremental processing which recomputes only those sections of output that need to be recomputed from changes to source or transformation. An execution flow tree structure could perhaps be used to trace back from the output to causal points in source document or transformational template and thus support template

editing from interaction in the result document. Ollis[78, 80] has shown how a state-tracing XSLT processor can generate significant performance optimisations for re-evaluations of parts of a computation, exploiting the functional separation of the programmatic model.

3.7 Other functional approaches

Several document generation systems have been based purely on functional programming or with a strong functional model at their heart. Impetus from ‘pretty-printers’ for code documentation has generated a number of approaches. Hughes[35] defined a relational algebra of vertical and horizontal combinators of sub-assemblies. Using this algebra, implemented in Haskell, it was possible to generate and select solutions from exhaustive sets of possible arrangements in a lazy manner, using monad representations to accommodate backtracking in search. Wadler[96] refined this approach by using a single concatenation operator between ‘sub-documents’, which is possible given the requirements for indentation as an indicator of nesting in program code, and the inherent serial nature of the code to be displayed.

Hansen[31] developed a formatting model based on pure functions defining layout, and a type for content/geometry based on boxes with a small number of properties – direction, alignment, gluing and reference points. A small initial set of primitive functions is defined (*Layout*, *Break*, *all*, *first* etc) from which useful compound functions can be built by composition, such as *paragraph*, *adj-lines* (justification) and *table*. Document descriptions are written in a domain specific language, FFL, which is evaluated through interpretation.

Kingston[50] produced *Lout* as a purpose-designed minimal expression language for document layout which is highly extensible through functional libraries. It has a relatively small number of expression operators that both combine content sub-assemblies that have a canonical rectangular aspect into compounds with a similar projection, as well as imposing styling. Aspects such as name visibility are addressed to support modularity and packaging.

The document is parsed from a single large expression into a tree which is then subjected to (multiple) lazy evaluation. A number of higher-order compound actions are supported, most notably pagination through a system of galleys into which content pieces are promoted as the layout tree is evaluated progressively – this also uses a ‘force width’ operator to impose suitable constraints from the current target galley. The implementation performs five passes across the tree, distributing and collecting information in both downward and upward directions. Cross-

references are resolved through multiple passes. Kahl[44] has taken the ideas from Lout and produced a formalisation of the galley concept using Haskell.

Scribe[21] is a report-style document generator, implemented in Scheme, which is functional, in that abstractions can be defined (in Scheme) and functional composition can be performed. But its geometry is confined to the usual text-flow model inherent in most report generators (and that of HTML) and cannot define arbitrary graphical relationships and placements. It is capable of being targeted to several output formats, including *Lout*.

Slideshow (Findler and Flatt[18]) is a functional DSL attempting to bridge the gap between LaTeX and Powerpoint in producing slide presentations that have both the ‘style’ available within GUI-based presentation tools and the extension and abstraction that programming systems provide⁷. Basic graphical components (canonically treated by their bounding rectangle) are combined through a small vocabulary of geometric relations (*superimpose*, *append* etc.) or altered through affine transforms (*scale*, *rotate* ...). Parametrised functions with single-assignment variables permit definition of more extensive abstract assemblies. A ‘search’ action is provided to enable geometric information to be acquired from subassemblies – this is used to support ‘cross-tree’ relationships, such as drawing arrows between components⁸. Further functions can generate a *sequence* of slides over a list of parameters, such that pseudo-animation is possible. *Slideshow* diagrams are defined in Scheme and evaluation generates Postscript.

Whilst it is rarely used as a direct programming language these days, Postscript can build (named) functions on the fly and provide a basis for higher level document generation libraries, though lacking a lambda construction limits higher-order capabilities⁹.

Other functional systems have been developed as attachments to an existing graphics format, most notably decorating SVG documents. King *et al*[49] approach defining continuous multimedia animation by attaching the XML dialect SMIL (which describes event causality over time) to an SVG graphical framework. Links between the two and external events (button presses, mouse drags, passage-of-time...) are supported by embedding functional expressions in attributes, both ones that are ‘known’ to one of the dialects, and specialist declaration ones.

⁷And we should perhaps, as computer scientists, preach.

⁸There are strong similarities with the presentational variable model of DDF (section 6.4) where XPath is used as the search tool.

⁹The author remembers fondly when his monthly wall calendar was a Postscript source whose first two variables, *year* and *month* were manually edited as appropriate before sending the result off to the printer.

This declarative representation can then be ‘safely’ compiled into some appropriate functional framework – several examples such as Yampa, exploit Haskell for the implementation engine.

Thompson *et al*[93] extended this idea to a more general notion of declarative extensions to XML with functional properties. The primary goal was to support reusable or event-modified behaviour that required structured XML fragments to describe and were thus not placeable directly in attribute values. They propose a variant of the static SVG **symbol**, **use** constructs (**template**, **instance**) with a parameterisation model that is purely declarative.

3.8 Partial evaluation and constant folding

Partial evaluation of some program or function involves determining the consequences on parts of their computation resulting from some knowledge of the context within which the program or function will be evaluated, and modifying it accordingly. For a function this might be a specific binding for one of its arguments; for a program it might be restrictions on its inputs. In both cases it may be possible to propagate this information through the internal semantics, to discover sub-sections whose results can be computed precisely. For example, an arithmetic expression can be replaced by its numeric result if all variable terms have been bound; conditional sections can be flattened if there is enough information to determine truth of their tests. In extreme cases an expression may be completely invariant to any influence from the input – pre-emptive evaluation and replacement in such circumstances is termed *constant folding*.

A key opportunity with documents being considered as functions is to exploit such partial evaluation either as a specific step in the document's life-cycle (e.g. observing documents when they are only partially bound to data) or to optimise performance in document generation. As one of the essential properties of functions is absence of side-effects, which extends to these document representations, this implies that under clearly defined circumstances, *parts* of the documents can be evaluated and replaced with results in a piecemeal, predictable and robust manner.

Such evaluation can take place either because the document generation context means only parts are complete enough for full evaluation (e.g. ‘continual’ or streaming documents) or during some preparative stage, such as compilation, for subsequent processing, when the optimization involves *constant folding*. As such this is *eager* as opposed to *lazy* evaluation.

There has been much work in functional and logic programming on the use of partial evaluation[43]. Strictly it implies inferring one, usually more efficient, program from another given some specialisation of its input, which often involves fixing one or more of a set of arguments to the function. Quoting Launchbury[56]: ‘at its simplest, partial evaluation may be thought of as *currying on programs*’.

This approach can also be used for some surprising applications such as generating efficient language compilers or software architectures from interpretive language definitions[71]. A simple example is a compiler partially evaluating a regular-expression interpreter for a given regular expression string in the program input, thus generating efficient specialist code to be placed in-line[1]. A related approach[10] supports partial re-evaluation by analysing internal data dependencies as a dataflow graph and propagating consequences to appropriate nodes.

For this thesis the interest lies in two specific areas of partial evaluation: i) evaluating a program when not all its data is bound – this can occur when a template is specialised to generate another template, such as binding company details to a generic ‘invoice’ template, or when data can be piecewise continuous, e.g. a medical record, and ii) evaluation of invariant sections within the document itself – this is mostly confined to sections of layout and can be merely a special case of some more generic partial evaluation.

Canonical representations of functional signatures, such as currying, where the basic function is of *one* argument and multi-argument functions are compounds, make such evaluation possible. In the case of XSLT, the language being represented in the main data type (the XML tree) makes it possible to consider surface-level evaluation techniques.

It is interesting that *XQuery3.0*[127] proposes not just first-class functions for XML search queries, but also partial function application, i.e. binding some of the arguments and returning a function of a reduced set of arguments as the result. Whether or not any partial evaluation actually takes place within that step is a decision for the implementation.

It is also worth noting overlap with techniques in high-performance compilers, where programs are modified, whilst preserving correctness, to optimise performance against some characteristics of the input data. Similarly techniques such as *partial redundancy elimination*[14], subsume both identification of common subexpressions and movement of invariant sections.

There appears to be very little exploration of layout invariance for document *construction*. Mac-

donald [67, 70] explores invariances in pure layouts for the purpose of partial binding, and exploits the details of the layout functions themselves, with associativity as a main technique, such as in flows.

Giannetti and others[26] have explored optimisations for very large document print operations based on XSL-FO. PPML's *reusable objects*[113] are used to wrap invariant sections and a parallel renderer based on FOP[100] exploits this higher-level description. It seems that documents must be designed *ab initio* for such use, or the authoring tools (such as *InDesign*) need specialist extensions.

PART B
PRIOR ART IN DDF

Chapter 4

Document Description Framework

The philosophy and general design of DDF as an extensible variable document framework is presented in this chapter, detailing choices made during the original research and outlining the major structures within a DDF document. The expected operations on documents (i.e. anticipated workflows) are discussed and approaches to authoring and editing such documents are presented briefly.

Three factors influenced the design choices for a framework for flexible variable documents. Firstly the requirements for future documents and the forms they might take were not at all clear – the roles the document could fulfill, especially when considered as a self-contained entity¹ may vary immensely. Secondly, documents would have to be capable of extreme ‘personalisation’ and, thirdly, they would need high presentational quality to ensure that their inherent communication is effective. As discussed in section 1.2, these last two requirements have generally been at odds with each other – high quality has typically involved skilled graphic designers hand-tweaking their documents – something that is not possible in high volumes².

DDF focussed primarily on ensuring that the document framework developed would be *extensible* to accommodate such uncertainties in required functionality. Some new technique would

¹The paper form we have all grown up with is ‘self-contained’ – it doesn't fail when the Internet is down.

²Eric Gill is said to have altered some of the wording of poems he was printing in order to produce a better typographical result.

not require a complete overturn of existing programs – rather, modest and unintrusive extension to the implementation machinery would be sufficient, or a document or template could be ‘programmed’ to accommodate the needs. Five main features were developed:

- The document was considered generally as an executable function over its ‘data binding’. Variation over the data was primarily declared as fragments of XSLT functional program contained within the document itself, or other documents that it would reference as dynamic templates. In this manner much of the programmatic variation can be handled *exclusively* through ‘code’ buried in the document, with no need to change the implementation framework. For example the inclusion of a *data mapper*[23] could be incorporated merely by importing a suitable library document rather than overhauling the implementation engine.
- Presentational layout for a document was described as a hierarchy of combination of sub-components, represented as a functional tree of combinators over component parts and associated control information. Programs within the document could generate declarations of layout intent (as trees), dependent upon new data binding; layout agents in the implementation interpret these declarative trees by building up resultant sub-assemblies. This is discussed more fully in Chapters 6 and 9.
- A logical document structure layer was added to increase the buffering between data and layout and enable effective use of well-developed libraries.
- A declarative system of describing sets (families) of documents and their inter-processing in an XML form. A wide variety of different document solutions could be supported with a single common implementation engine (the ‘DDF Family processor’). This could also examine inter-document dependencies and minimise rework under change conditions³.
- A methodology for authoring and editing such documents by graphical interaction with a presentation of an instance of the document – this will be outlined in section 4.3.

The first three became distinct spaces within the document – data, structure and presentation, which were conveniently represented in the XML tree. Programmatic behaviour was described by XSLT fragments embedded in each space that generates new content and declarations from information in a default source (bound variable data →data →structure →presentation).

³This will only be discussed briefly in this section and will not be examined in detail later in the thesis, save where a particular feature is required for, or is useful to, the management of document families.

A document after binding should be ‘self-contained’ – apart from obvious resources that normally live ‘outside’, such as images and fonts, most DDF documents are represented by single XML structures, i.e. stored as single files⁴. This corresponds more usefully to the human notion of a document, reduces errors in ‘cross-linking’ and also has implementational efficiencies.

4.1 The 'life' of a DDF document

DDF documents are intended to be used to make large sets of variable documents. As such there are five distinct main types of operation:

1. Binding a document to some variable data.
2. Evaluating all the ‘variability’ described in the document, which will inevitably produce new layout requirements and declarations.
3. Resolving all the layout within a document to produce a defined graphical result.
4. ‘Observing’ the document to create a visible image – usually in a well-defined visual final format such as JPEG images or a PDF file.
5. Authoring and editing the document and related companion documents.

A DDF document can also utilise resources from other DDF documents through methods of importation, inclusion or merging. This is essential if ‘library documents’ are to be used, rather than implementation frameworks being overloaded with libraries for particular types of document (e.g. documentation-generating documents). Such functionality is contained within other DDF documents, usually not destined to produce output ‘on their own’, which are then drawn on as required. The functional nature of the underlying XSLT model, especially the use of ‘push’ templates, makes this comparatively simple.

The most usual action is running the first four operations in sequence, often over a vector of data bindings – producing a vector of output final results. But the document architecture for DDF was such that a number of different types of operation could take place, including binding data in multiple stages and merging documents together at different stages of binding and evaluation. Figure 14 shows a number of possibilities, which will be explained in more detail in the following sections.

⁴Later work suggests that a heterogeneous storage system, e.g. ZIP, on the front end might be more generally useful.

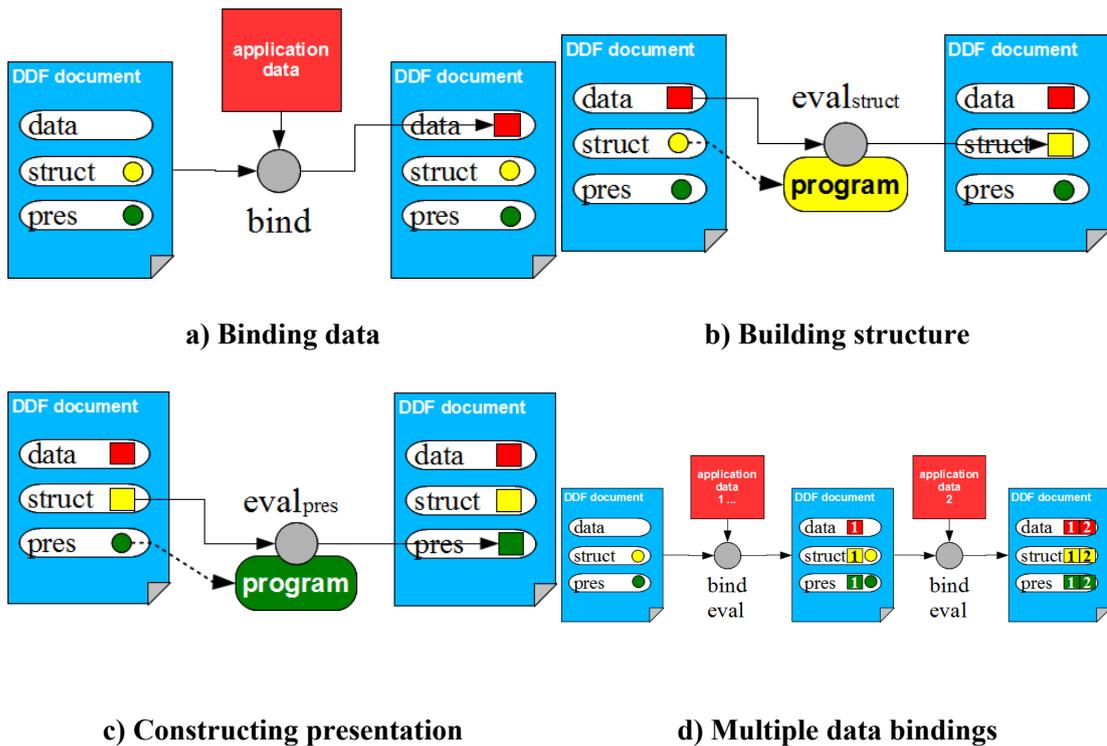


Figure 14. Typical workflows for a DDF document

4.2 The basic structure of a DDF document

With these design choices, a DDF document was represented in the following XML form:

```

<ddf:doc >
  <ddf:data> application data + programs </ddf:data>
  <ddf:struct> logical structure + programs </ddf:struct>
  <ddf:pres> presentation + programs </ddf:pres>
</ddf:doc>

```

where the top-level structures fill the following purposes:

- **ddf:data** holds pure application data. For most unbound documents this section is empty – on binding to an instance of data a copy is placed here. One reason for this is that during possible subsequent bindings and document evaluations new requirements on previously bound data may be necessary – hence the document carries around some ‘state’ information about its data bindings.

- **ddf:struct** can describe logical structures in resulting documents. This layer is primarily used to support a possible canonical buffer between data sources and presentational forms, which may involve use of a standard (such as XHTML), or a special-purpose semantics for a family of documents, all aimed at increasing document re-use.
- **ddf:pres** contains presentational instructions to create final visible forms. In an unbound document this contains sections of completed presentation and sections of XSLT that generate such instructions from the structures held within the logical structural layer. During a subsequent *layout resolution* phase these instructions are converted into a grounded presentation (SVG). This operation is covered in much more depth in Chapter 6.
- ‘**program**’ components generate new content as a result of variable bindings. These programs are described in XSLT. Each of the spaces can contain such program, though that in **ddf:data** is only used for special effects in higher-order and multiple-binding situations. Placing these programs within the space where they are going to generate results means that programs do not need separate ‘linking’ and were self-contained with the documents for which they were relevant.

There are a few other ancillary top-level structures within a DDF document such as for describing documentation or even provenance. One significant set of features support reference to external resources, most notably images held in file-systems. This consists of two possible forms: **ddf:data-external** which declares patterns in the input data that should be considered to describe references to external resources (e.g. **picture|image**) and context maps (**cmap:context-map**) declaring mappings between context tags and universal resource locations – this is described more fully in Chapter 5.

The use of XSLT as the language to describe response to variability has proven very advantageous. Firstly of course it works exceptionally well on XML-based data of many types. Secondly, being written in XML itself, it can reside comfortably within the DDF document, as well as being processed by other XSLT-based tools in a very consistent manner (including during document editing). Thirdly, it is entirely capable of describing many forms of manipulation in a flexible way.

For example consider a document that needs to partition two sets of sales items between two pages, with all the higher-value ones on the first page and the rest on the second. In the case of a fixed system⁵ some specialised extension would have to be added, or the ‘upstream’ data

generators reprogrammed to produce a **page1, page2** grouping in the data. Using XSLT it is a simple matter within the appropriate generator to place a computation of a ‘split-point’ and two different expressions:

```
<xsl:variable name="split-price" select="sum(product/@price) div count(product)"/>
<page>
  <xsl:apply-templates select="product[@price gt $split-price]"/>
</page>
<page>
  <xsl:apply-templates select="product[@price le $split-price]"/>
</page>
```

The following chapters of Part B discuss the syntax and semantics of these sections of a DDF document more fully, with many examples and explanations of some of the implementation techniques. Satisfying the goal of significant flexibility and extensibility from document-borne variation is hopefully demonstrated.

4.3 Authoring and editing from the document's range

Documents that are variable need some approach to editing them that does not require doctoral-level programming skills. A series of integrated tools should preferably present almost WYSIWYG editing, albeit with the full benefit of being able to control considerable variability in the document's layout and its response to data variation. The basic technique has been described at some length in [63] and the details will not be repeated here in this thesis.

The approach is equivalent to editing a function through ‘reverse-mappings’ from values found in the *range* of that function. In this case the document is the function and the range value is the ‘multi-dimensional’ tree result that has been generated by the document from a similarly multi-dimensional instance value in the *domain*. Obviously there are constraints on when this can be achieved sensibly and robustly, but it seems that for many ‘document-like’ functions this can be performed remarkably often: within a typical document ‘template’ most of the features are nodes within a tree-like construction. In the case of DDF, layout itself is defined by such nodes and there is one-to-one or one-to-many correspondence between the nodes in the function definition and nodes in the result.

The technique involves injecting ‘editability’ annotations as attributes on suitable (i.e. edit-

⁵Such as *Dialogue*, though that product does have data mappers that might be capable of such a trick.

able) nodes in the source document, and relying on ‘good XML citizenship’ and approximate tree isomorphism⁶ in the XML tool-chain that generates instance result documents to propagate these to positions in those (XML) results. From those results some active view can support user interface actions such as selection of elements and groups and the generation of editing dialogues. The annotations also include source ‘addresses’ so the required changes can be performed on the correct nodes in the original source document tree.

This is enabled by an extensive compiler that take descriptions of document work-flows and declarations about types of node that are editable and how. The compiler generates i) a suite of transforms that are stitched into the work-flow to add suitable annotations, ii) active views for a generic user interface which will add the desired interactive behaviour and iii) ‘effector’ transforms to alter the original sources as a result of the edit requested. Drawbacks and possible developments are discussed in section 13.4

⁶These two concepts are discussed in more depth in Chapter 9.

Chapter 5

Functional Implementation

Before final consumption by an intended recipient, a variable document has to be bound to, and evaluated over, an instance of that variable data. In this chapter we discuss how the *programmatically* variability in the document (that is the sections defined by XSLT code) is executed, by using a ‘compiling’ transformation of the document into an XSLT executable that is then run to produce a bound result document containing the appropriate presentational layout description.

Variable documents are developed specifically for binding to many instances of data and producing large sets of personalised documents on an individual basis and without human intervention. Documents must therefore be evaluated over a specific data instance, thereby determining the presentation that results. Previously two aspects of the variable document response have been shown – the data/structural (XSLT) variability (what groups of content are needed, order and choice of components, generated content) and the presentational (layout) declaration, generated from this, and whose final visual form is calculated from that declaration.

In the most extreme cases every presentational part of every instance might be totally different, but in practice there is much commonality and pre-evaluation of common subcomponents is a distinct possibility. This chapter discusses the *ab initio* evaluation of the document's functional aspect – Part C considers methods of optimisation through partial evaluation and constant folding. Resolution of layout is considered in the next chapter.

5.1 Evaluating the (XSLT) functionality

The XSLT fragments within a document are the only source of *data variability* (the only channel through which data information propagates) and can be executed in a separate phase before layout. A DDF document contains three distinct ‘spaces’ of such XSLT – **data**, **struct** and **pres**, which operate on **input**, **data** and **struct** sources respectively¹. An implementation of this phase would have to propagate the effects from the **input** into the **data** space², and thence progressively through **struct** to **pres**. Figure 15 below, using only the ‘pull’ mode, shows what is expected in production of a simple web page, where the three sections are split apart.

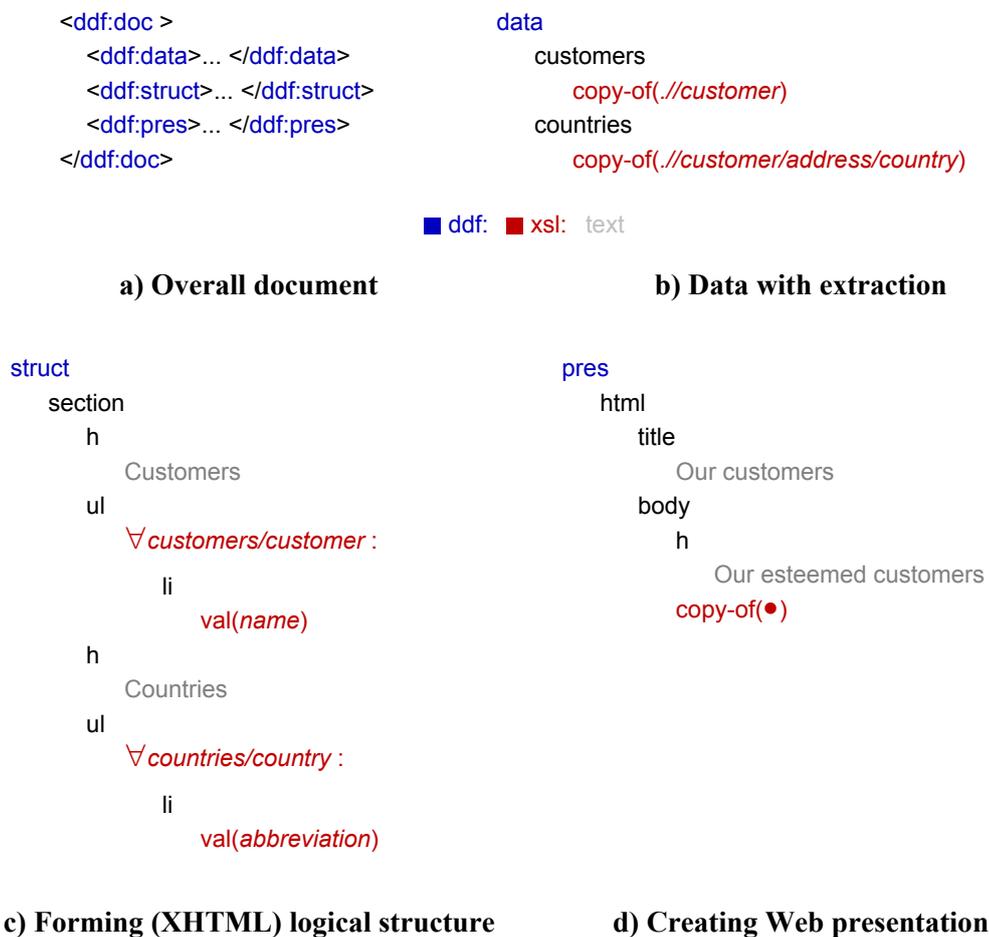


Figure 15. Simplified XSLT evaluation of DDF spaces

Diagrammatically, we are carrying out the following stages of evaluation:

¹Alternative source ‘routes’ needed for higher-order documents are described in Chapter 11.

²Programmatic operation during input is usually confined to evaluating indirections, such as inclusions, contained in the data set – this is helpful with templates that make ‘report-like’ documents, such as a PhD thesis.

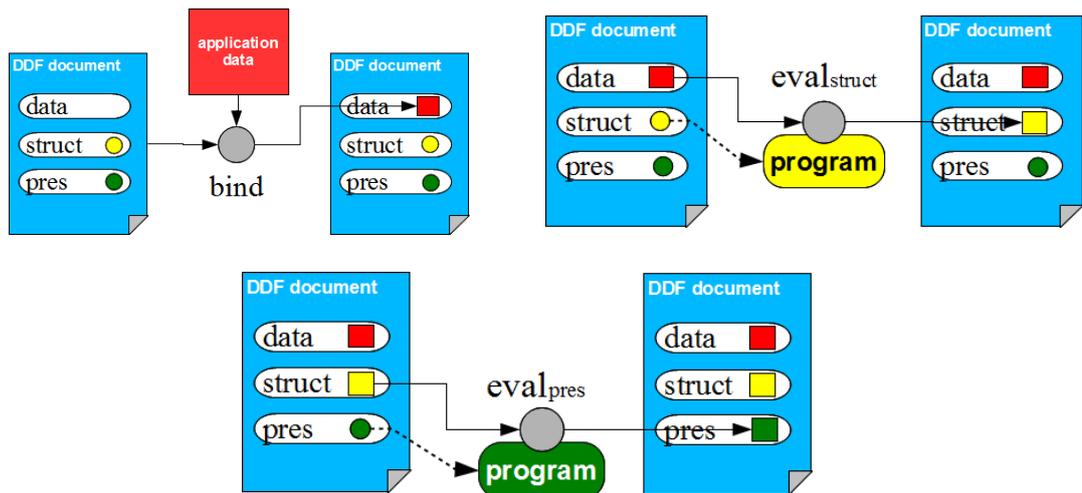


Figure 16. Typical internal workflow for a DDF document

There were a number of possible strategies for this evaluation: each space could be taken separately, the XSLT ‘extracted’ and then run as a transform on the appropriate input source to produce a modified output, this process being repeated ‘down the chain’. A processing description language such as XProc [125] could be employed with a suitable implementation engine and extractor and combinator tools to work between DDF document components.

An alternative, as used here, is to ‘compile’ the document into an equivalent single XSLT transform, which when operated over the input, carries out all the above operations in sequence, building up the final correct bound DDF document, with the data variability projected through completely and operating at the start of the document processing pipeline, as shown in Figure 5. As we shall see, this approach has a lot of attractions, especially in being able to support many of the additional features needed for more powerful document solutions.

The DDF document ‘compiler’

The compiler is naturally an XSLT program that takes a DDF ‘main’ source document as input and produces an XSLT executable program as the result. Running this executable on an input data source will produce a DDF document correctly bound as result, even though the layout resolution phase has not been executed³. What we are doing is shown in Figure 17.

³For a full production operation this might be incorporated into the compiler's output, but for research purposes keeping them separate increases flexibility. Note that the compiler itself could actually attach to the layout processor for detection and evaluation of invariant layout sections – see section 10.2.

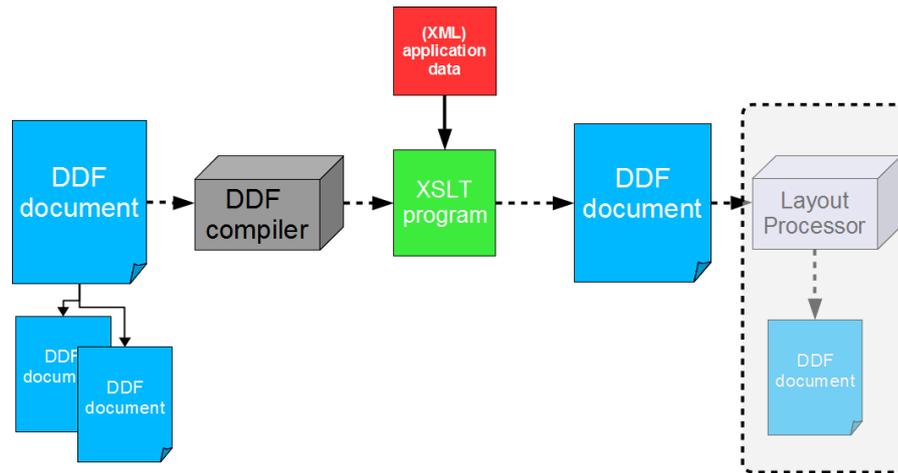


Figure 17. Compile & Run

The compiler has to extract relevant sections from the defining documents and correctly assemble them into a computational sequence that will provide the required result. This is made much easier by the use of push processing within XSLT. Figure 18 shows tree-graphics representing the three major documents making up the brochure document described in Chapter 7.

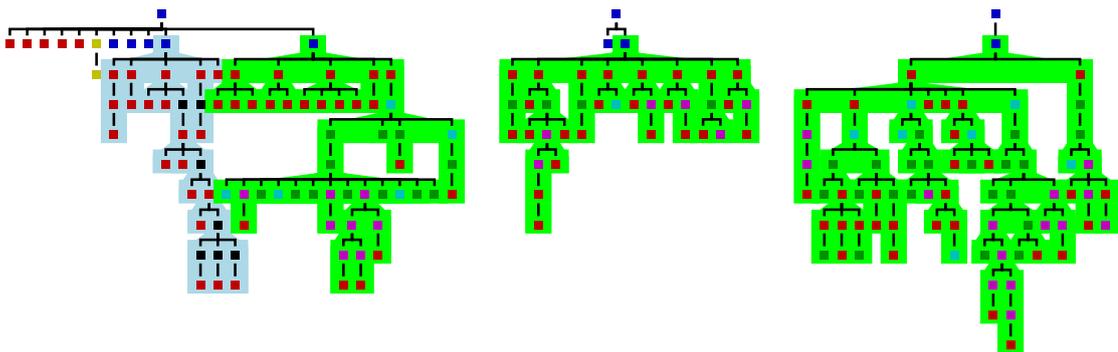


Figure 18. The DDF documents for the brochure – main, resort & pages

The sections shaded light blue are **ddf:struct** sections; those in light green are **ddf:pres.** (Only the main document generates structure, the other two are effectively reusable partial presentational documents.) The result of the compiler operating on this constructs an XSLT executable transform shown in tree form in Figure 19.

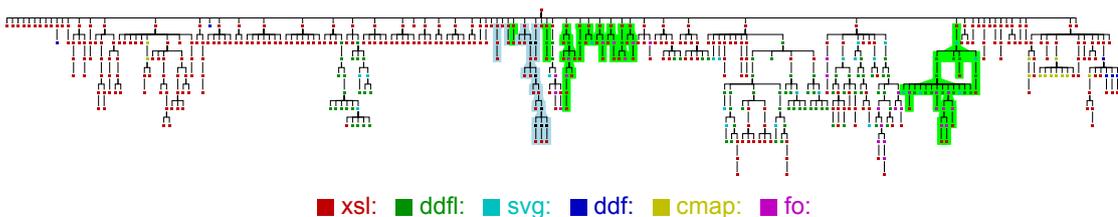


Figure 19. The XSLT output of the compiler operating on Figure 18

The shaded sections are **xsl:template** components operating in ‘default’ modes, transferred

from the original documents, with colours corresponding to those used in Figure 18. When *this* transform is executed over a data binding the result is the XML tree structure shown in Figure 20 which is a valid, but now static, DDF document. The four principal subtrees correspond respectively to the *context maps* (see next section), the *data*, *structure* and *presentation* sections of the resulting document.

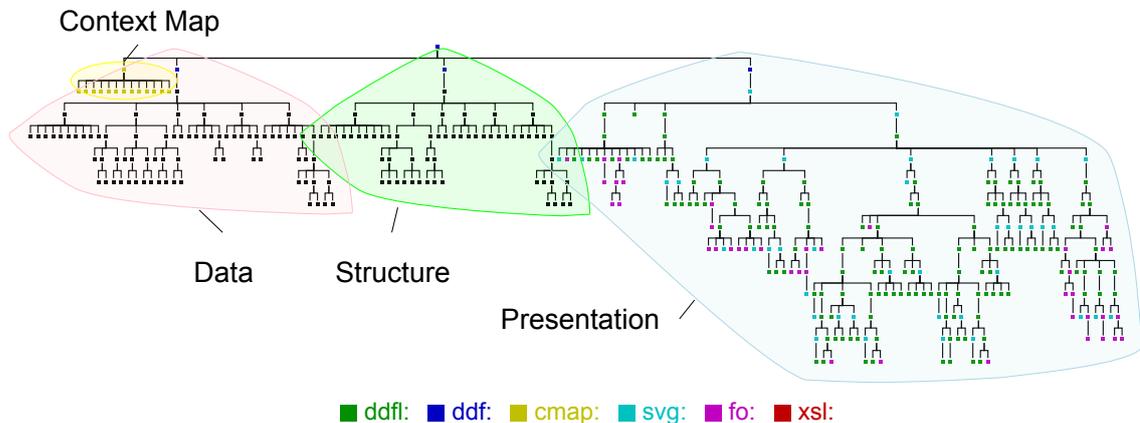


Figure 20. The result of executing Figure 19 on a data instance

The large subtree on the right (the **ddf:pres** section) holds the resulting presentational declaration with elements in three main namespaces: **ddf:** for layout directives, **fo:** for text blocks for line-wrapping and **svg:** as grounded pieces of graphic content. How this layout is resolved to final graphical form is described in the next chapter and shown in Figure 24.

Compiler design

The compiler *generates* XSLT from elements buried inside both the DDF documents and the compiler itself. Some of its actions are merely simple transformations of XLST fragments embedded in the DDF document, others generate sections of XSLT from ‘DDF’ directives and declarations therein. In part the tool is merely a program transformer, in part it compiles from elements in one language into another. In all cases it arranges that the result of execution of the generated XSLT on an XML input tree produces a legal DDF document as output. Figure 21 shows a general schematic of its major components and actions.

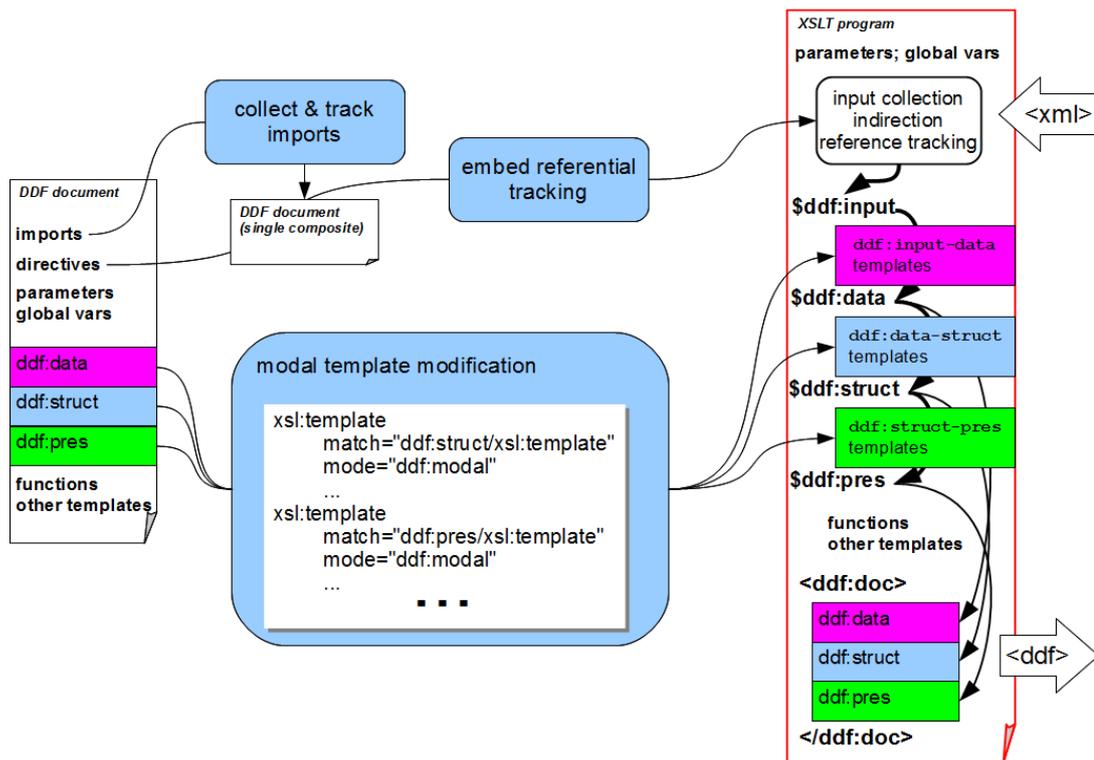


Figure 21. Compiler design

The preliminary steps performed by the compiler are:

1. Read the input (XML) DDF file, expanding recursively any importations and inclusions of other DDF files, while checking for multiple requests and only bringing in a file once.
2. Construct a transform result skeleton, first adding ‘control’ elements, e.g. **xsl:output**, that specify desired output formats, and indirect requests for any required XSLT libraries that a DDF document calls on, through **xsl:import** or **xsl:include** statements.
3. Write in all global parameters, variables and functions (appearing at the top-level in their respective documents or spaces such as **ddf:doc/xsl:param**, **ddf:pres/xsl:variable**), checking for name uniqueness. Attribute sets are processed similarly except that multiple sets can have the same name – the attributes within the superset are merged.

The next stage is key: program sections which are ‘push’ **xsl:templates** within each of the spaces are altered to operate on the correct source space. (‘Pull’ construction does not use modes and is entirely contained within a single tree.) We define a number of ‘transfer’ modes: **ddf:input-data**, **ddf:data-struct** and **ddf:struct-pres**. The templates in each of the spaces is transformed so that instructions that operate in the default (blank) mode are changed. Default templates for each of the modes (e.g. root matches encompassing static material) are also added.

This transformation is performed by ‘push-mode’ XSLT templates operating in a specific mode – **ddf:modal** and the transforming is ‘deep’, that is all code is processed by templates defined in this mode. As we shall see later, by adding to this set of code-transforming templates, the semantics of some parts of DDF can be extended smoothly. In the more complex situations described in Part C, a document might contain directives to support application to partially-bound or continual data, requiring that programmatic constructs are propagated forward into the resulting DDF documents. Adding an extra module to the compiler which contains additional templates operating in the mode **ddf:modal** means that such changes to code can be performed relatively simply and in a monotonic manner.

It is reasonable to expect that XML data sources might contain indirections to other sources, such as embedded XInclude[122] directives (`<xi:include href="URL"/>`). A document-borne library could contain code to match such situations, but having to track external references, that such remote resources might contain, requires deeper support. The compiler can generate integrated code to support such XInclude semantics in the input, which involves adding a set of additional push templates into the **ddf:input-data** suite.

Finally the main thread of the document's ‘run-time’ form is then written as a sequence of variable-generating stages that will appear in the document run-time code. When executed this will build a DDF document as the result:

```
TEMPLATES
match:/
  data=
    =>/mode="ddf:input-data"
  struct=
    => $data mode="ddf:data-struct"
  pres=
    => $struct mode="ddf:struct-pres"
doc
  OTHER-DATA
  data
    $data
  struct
    $struct
  pres
    $pres
```

External references

Early prototypes showed an issue when application data acted as relative references to external objects, most notably images. Final documents and intermediates may be in different places from sections of the original application data, thereby making such references exceptionally fragile. Immediately resolving the references to an absolute form helps only partially – a later move of a set of documents and resources would invalidate these new references.

A mechanism was needed to track such references through arbitrary XSLT program (especially when forming temporary trees where ancestry paths back to source routes are normally broken). A solution is an explicit system of context maps. *Source locations* (documents, data files) are given tags (e.g. **input.0.2** for perhaps the third file indirectly ‘imported’ by the first (main) input file⁴). Relative referential elements in that file can be altered to a URL protocol that uses that tag as a discriminant (e.g. **cref://input.0.2/photo.jpg**) and an associated *context map*, stored usually within the document, that contains the source ‘pointer’ (**<cmmap name="input.0.2">file://...</cmmap>**) from which an eventual absolute URI can be resolved as required⁵.

The implementation machinery contains a URI resolver for this **cref:** protocol and loads an appropriate set of context maps from documents. Subsequent references to such a URL will be resolved to the correct source automatically. The DDF document compiler ensures that ‘relative reference’ elements are actually tagged. But these references could occur anywhere, and in multiple forms, e.g. a customer data record may contain a **photo** element which ‘names’ a local JPEG, relative to the data record file. To track this a source-recording tag must be added. Documents that process potential referential material declares cases of such reference through a statement **ddf:data-external** containing patterns of elements or attributes that should be so treated (e.g. **svg:image/@xlink:href | customer/photo**).

The compiler converts such declarations into pre-emptive templates that will match such nodes during the initial input and if the reference is relative (i.e. not an absolute path or other URL) alters its value to be a URL using the **cref:** protocol and the context tag. At run-time the context tag parameter is set appropriately during each move into another included document to support relative references in deep inclusions.

⁴Zero based.

⁵It was expected that such maps would support switching between different forms of a resource, such as high and low resolution versions of pictures, simply by altering the mapping – little use was made of this in practice.

5.2 Document workflow

The essence of the variable document is that of a simple function, evaluated on an argument, but in practice the real value lies in repeated evaluation and other operations. This will usually be conducted through some form of workflow, most commonly with repeated application of a template document to an entire vector of data bindings producing a corresponding vector of result documents for distribution or print. This is the usual ‘publish for customers’ operation, where we can expect the data vectors to be up to $O(10^4)$ data records long or even larger. In such cases the efficiency of the workflow execution is vitally important.

Alternatively a document might be taken through a progression of partial bindings and evaluations to yield a further template. For example a very generic brochure template may be bound to a specific supplying company's information to produce a specialised brochure that can then be populated across large volumes of customer data as before.

Both of these examples need accurate description of what is required, so the implementation machinery can produce correct and efficient executable programs to generate the required documents. In the common case the following are important:

- Only the final presentations of the resulting documents are needed and these documents will *not* be processed further or bound to more data. With knowledge of this the compiler can be optimized to generate executable XSLT that drops unwanted result sections (**ddf:data**, **ddf:struct**)
- When a very large number of documents is generated it can be worthwhile pre-evaluating as much of layout as possible, either exploiting invariance or speculation. This can be achieved either by the compiler detecting such conditions or by suitable pre-processing of the main template⁶.

⁶This is discussed in Chapter 10.

- Again, with large data sets, it may be advantageous to ‘compile-in’ the layout processor to increase performance, or arrange that significant common resources and sections are passed as far as possible down the processing chain. For example a static background could be totally evaluated and passed as a common resource to the eventual PDF generator, whilst the individual document instances refer to the resource rather than containing the static background. (Overloading **svg:use** is a possibility, or employing a standard, such as PPML, for describing reuse over large document sets.)

A machine-based description of the workflow for such a case would contain either specific directives to control such actions or information (such as the size of data vectors) that would enable this to be determined. An XML representation was developed that could describe the type of relationship shown in Figure 22 where ‘Customer Data’ can be bound to an XML representation of an instance vector.

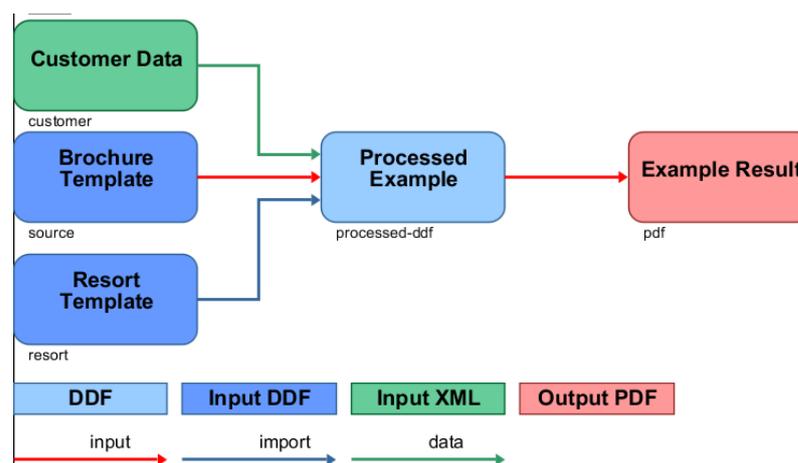
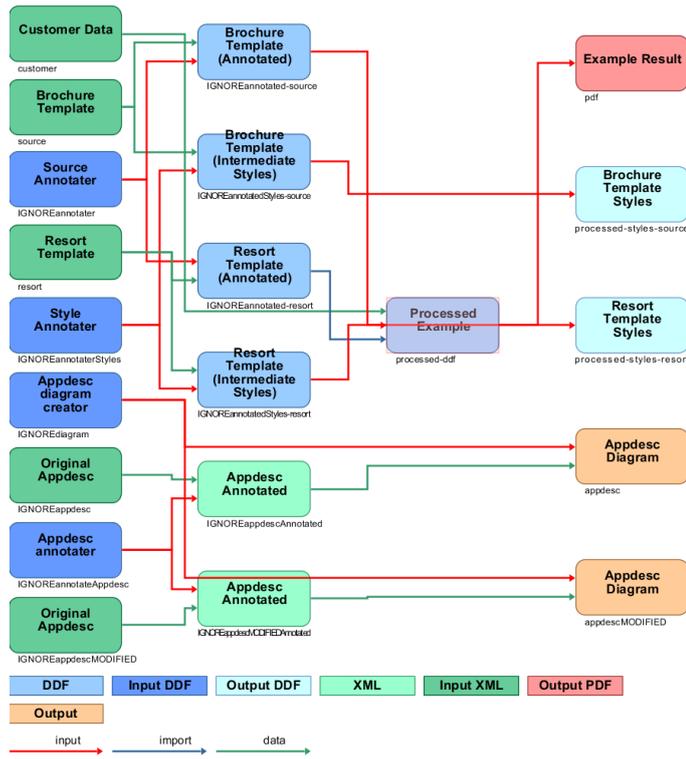


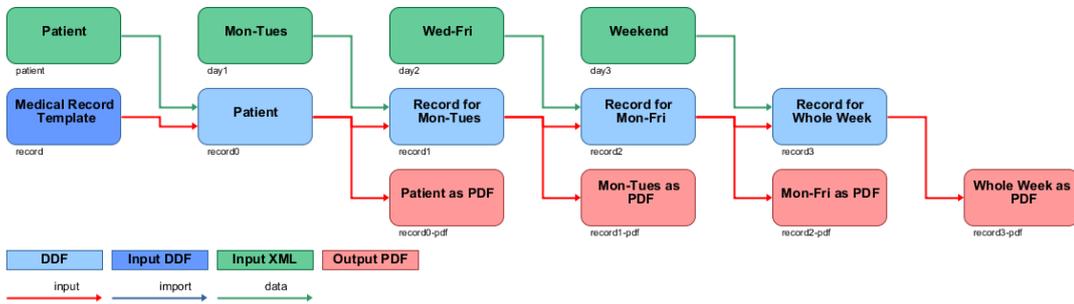
Figure 22. Simple document workflow graph

The representation can support much more complex dependencies. Figure 23a is a workflow from document editing, which itself has been generated from another workflow by an XSLT-based edit compiler⁷ to arrange that edit traces are injected correctly into source documents. Figure 23b is a multi-stage binding of data for a medical record, used in Chapter 12.

⁷This is an excellent example of the power of everything being described in XML – even the workflows descriptions can be read and modified by XSLT.



a) Editing documents



b) Multi-stage binding

Figure 23. Complex document workflow graphs

Chapter 6

Layout in DDF

The model for document layout within DDF uses a tree-based framework of ‘functional combinators’. This chapter describes this in some detail, demonstrating the use of a ‘sea of agents’ and a canonical rendered XML form (SVG) to build a system with considerable extensibility and flexibility. Additional well-founded facilities that provide XSLT-like facilities within the layout phase, such as presentational variables, can add extra flexibility within the document and are discussed..

Eventually a document has to produce a visual form, and this visual form must be effective in communicating the messages inherent in the document. Aspects of these messages are usually contained in the document's logical structure, including grouping, order, importance, commonality and so forth. The presentation must clue all these aspects through visual and geometric effects such as position, size, relative placement, fonting, colour, visual scope and decoration. The mapping of such effects from logical structure is the essential *style* of the document. When a style is also used to carry corporate overtones, this is also termed *branding*.

Some parts of style are simple properties, such as a particular font for standard body text, a colour for a background or different ‘bullets’ for nested list items. Others are *relational* – the relative placement between a ‘group’ header element and the content items of that group, and collecting footnotes at the end of a visual-scope (page) are examples. Some of these are very local in scope (a contrasting background colour for an element with variable foreground col-

our), others are global across a document (e.g. the same relative placements for members of every group corresponding to the same type of logical group), or even across a whole suite of documents.

In the most general sense these properties and relations of style are a set of constraints on and between presentational elements of the document. These constraints could be described in many different ways, from direct equational relationships (**para3_{right} < line27_{left} - 17, background14_{fill} = contrast(para3_{fill})**) to very abstract under-constraints (**group15 similar-structure group24, image24 smaller-than image52**).

In practice these mappings are described in some form of *presentational language* with consistent syntax and reasonably simple semantics, which are both relevant to the general type of document being built and computationally feasible to implement. The language can be external – source files can be written directly, such as the *Troff/Scribe/TeX* family; alternatively they can be internal to some authoring and generating tool, such as *MSWord* or *Dialogue*.

The language can vary in complexity, from the most simplistic (Euclidean and grouping constraints in *Juno*[34]; simple paragraph/list markings in *nroff*) to full-blown document control with support for ‘plug-in’ extensions, such as *InDesign*. External languages are usually written as *markup*, that is the bulk of the document is content text, with elements or commands denoting markup directives interspersed through the text, as opposed to a language where textual content is buried in ‘program statements’.

6.1 Extensible layout

One of the major design goals for DDF was that it should be easy to extend its capabilities without extensive rework of tools and implementations. Preferably such extensions should be monotonic – merely adding a new section of code via some declarative manifest should be sufficient to add new functionality, without having to rework other sections. This was particularly the case in document layout. We had no detailed knowledge of what sorts of layout would be required in future: a fixed ‘format’ such as a paginated text flow would be inadequate and only one of the types required.

There were a number of requirements:

- Many different types of layout could conceivably be needed and mixed. Some unusual examples might include placing items in colour-contrasting positions on an image background, altering text font-size to make a paragraph fit a given box or dropping optional content when given containers are full.
- The layout should be able to support functional binding within it, and the transmission of future functional binding into result layouts – this latter being necessary to support template-generating templates and continual documents.
- Text needs to be supported at ‘professional’ levels for actions such as line and paragraph wrapping, inline decoration, kerning and the like, but documents which were *not* text-centric should also be supported. The framework should not be based entirely around text.
- Grouping of components should be accomplished easily. In fact grouping should be almost built-in to the framework, if only to encourage the use of logical structure in document design.
- Functionality could also be described in documents themselves, acting as ‘macro libraries’.

However, we should also assume some ‘default’ view of the document layout. The critical choices to be made included:

- Is layout primarily *inward* or *outward* facing – do the ‘pages’ in general drive the ‘size’ of components, or *vice versa*?
- Is layout generally *local* (i.e. the effects of geometry are generally confined to neighbouring elements) or is it much more global?
- Are non-local effects *cyclic* or *acyclic* between sections of document layout?

There is also the issue of *where* in the document processing chain the evaluation of the presentation should be made. As already seen, a DDF document can contain significant sections of XSLT-based processing code, so through the use of XSLT template-bearing documents we could extend a layout repertoire in some cases *entirely within the documents themselves*. For example if every child of a flow had a defined size, positioning of those children could be accomplished easily by code within the document.

However there is a point at which the power of XSLT diminishes in solving geometric problems: the simplest of these problems to understand is that of line-wrapping. Fonts have to be identified and metrics recovered, word distances have to be computed and potential hyphen-

ation points determined, line breaks have to be decided based on measures that might cover a whole paragraph or more (Knuth-Plass) before the actual text placement can be performed. This all requires careful and detailed coding, assisted by extensive code libraries. It will probably have to be coded in something like Java – such code has to be included in a more static platform than documents themselves¹. Some of the required information (e.g. font metrics and hyphenation dictionaries) is unlikely to be contained within documents.

Considering these requirements, six main decisions were made:

1. ‘Laid-out’ presentation is described in an XML-based form. SVG is used with a canonical rectangular representation of pieces and compounds. This reflected the fact that most of the layouts we expected would primarily involve horizontal and vertical arrangements of some form and the bounding-rectangle would be a suitable starting point². SVG was developed enough to be able to represent professional level documents³, was well suited to generate printable PDF from it, and had sufficient additional features (animation, reuse) to give a little future-proofing. As we will see subsequently, having the presentational *result* described entirely in XML opens up significant possibilities in XSLT/XPath-based post-processing of presentation for many useful features.
2. Presentation is written primarily as a tree of combinator nodes that declares layout functions over a set of children. Properties for these functions are attached as attributes on the nodes or in reserved children – all other children are considered to be further presentation declarations. This matched well with supporting locality of layout action.
3. There is a standalone *layout processor* constructed as a set of templates that match specific layout combinator nodes, gather parameters, process children through recursion, form up the compound result and return it to the calling context. This is a very good match to an XSLT model. The layout processor operates in a separate phase *after* the document's variable function (XSLT) semantics have been evaluated through the compile-run action described in previous chapters. Thus layout needs very little knowledge of XSLT at all, save recognising such constructs for program-retaining documents.

¹Arbitrary side-effecting code within a document raises security issues – staying with side-effect-free XSLT within the document reduces this risk significantly.

²This does not exclude non-rectangular layout processing – implementations merely have to do a little more work.

³We used draft standards-proposed representations for page sets – these did not reach the main standard.

4. The *main* layout evaluation is outward facing, i.e. usually combining pieces that have already been evaluated and thus have a bounded size. This reflects use mainly in situations where content wasn't being packed progressively into a fixed container, but built bottom-up from components and sub-assemblies. However both directions of 'constraint propagation' are possible.
5. The layout processor is written as a series of XSLT modules with only a minimum of necessary (Java-based) extension functions. Many relatively complex operations, such as dynamic font-warping, are written as simple XSLT compound modules exploiting other modules.
6. There is a system of *single-assignment presentational variables*, akin to XSLT's variables. Document engineers can describe a large variety of *acyclic* presentational interdependencies through declaration and use of such variables and functions thereof⁴. Modest extensions in choice and iteration within this layer make it possible for many complex acyclic arrangements to be document-borne, such as pointing into trees (Figure 20) and partial magnification (Figure 35).

Choosing to represent the layout as an XML tree of combinators of primitives has many advantages. Possibly the most important is supporting *approximate isomorphism* between the declaration tree and the result tree. For example the declarative tree for one of the example brochure instances is shown symbolically in Figure 24, corresponding to the presentation tree shown in Figure 20. As will be discussed later, the shaded portion on the left describes some programmatic constructs (presentational variables) within the presentation. The large subtree on the right is a group of four page definitions – within each page is a hierarchy of combinators (**ddf!:**) over primitives nested up to ten levels deep. (The large structure within the third page of the brochure describes a labelled picture.)

When this set of layout declarations is resolved, the resulting tree (Figure 25) is completely defined in SVG which can then be visualised or converted to PDF. (The presentational variables have been interpolated into the result.) The important point is that the tree structure of the result is generally similar to that of the input. Most combinators generate a corresponding tree node in the result. By transferring additional information between these two nodes as a matter of course (such as **@name**) higher layout operations can exploit this information.

⁴Cyclic dependency means there is no simple evaluation order over 'equations' which will solve them all and requires other *simultaneous* techniques needing very detailed knowledge of the functions involved.

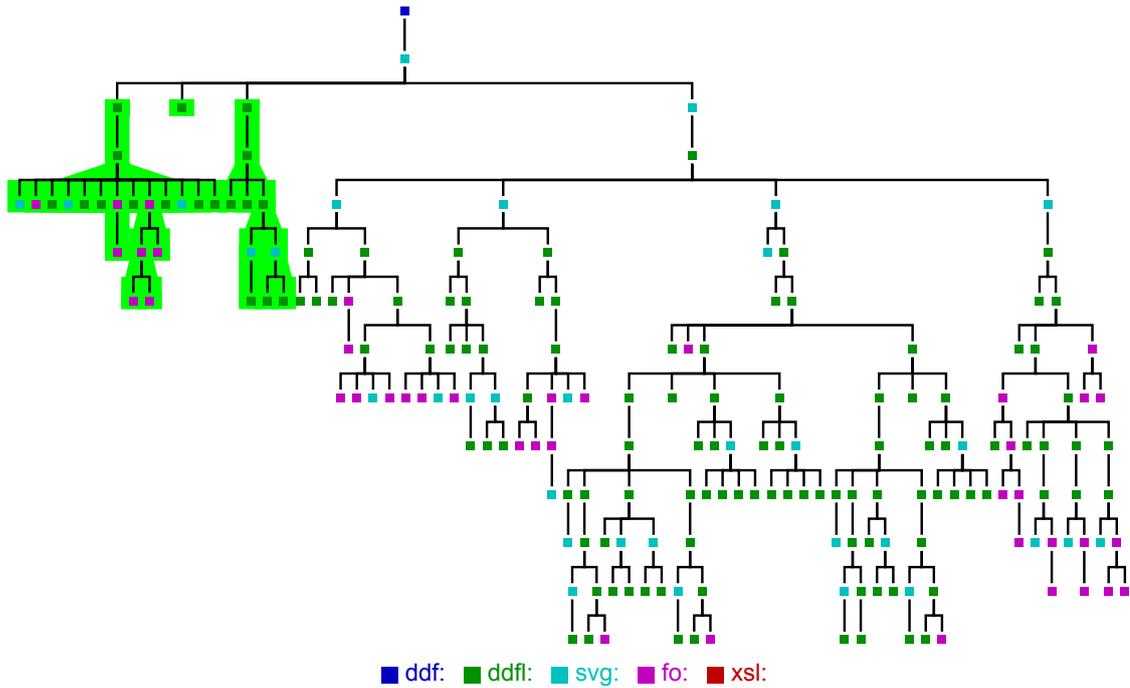


Figure 24. Presentation declaration for a 4-page brochure

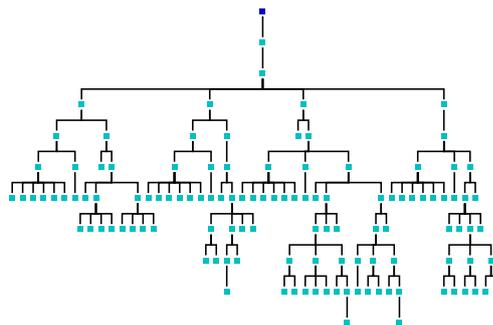


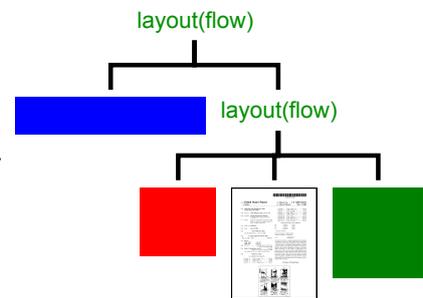
Figure 25. Final brochure after layout-resolution of Figure 24

To explain this approach in detail, here is a simple layout declaration in an XML form together with its equivalent tree (Figure 26):

```

<ddfl:layout function="flow">
  <svg:rect fill="blue" width="25" height="5"/>
  <ddfl:layout function="flow" direction="x">
    <svg:rect fill="red" width="10" height="9"/>
    <svg:image xlink:href="../../figures/patent.png" height="15"/>
    <svg:rect fill="green" width="12" height="12"/>
  </ddfl:layout>
</ddfl:layout>

```



■ ddf: ■ svg:

Figure 26. Simple compound flow layout – intentional declaration

which declares some hierarchical ‘flow’ relationships between a set of children. A suitable presentation that satisfies those declarations is shown in Figure 27 with the equivalent SVG:



```

<svg:svg height="20" width="33.32">
  <svg:rect fill="blue" height="5" width="25" x="0" y="0"/>
  <svg:svg height="15" width="33.32" x="0" y="5">
    <svg:rect fill="red" height="9" width="10" x="0" y="0"/>
    <svg:image height="15" width="11.32" x="10" xlink:href="..
/figures/patent.png" y="0"/>
    <svg:rect fill="green" height="12" width="12" x="21.32" y="
0"/>
  </svg:svg>
</svg:svg>

```

Figure 27. Simple compound flow layout – resulting graphics

A trivial XSLT template can evaluate the flow⁵:

```

match:ddf:layout[@function='flow'] mode="ddf:layout"
(element(*) children=
  => mode="#current"
  svg width="{max($children/@width)}" height="{sum($children/@height)}"
  ∇ $children :
    copy
      @*
      @y=sum(preceding-sibling::*/@height)
      *|text()

```

The essential point is that by using a *tree* both for the *definition* and the *result* forms of the presentation, we can define and construct many different types of layout without each needing much detailed knowledge of other forms. Provided all layout results produce a translatable and sized **svg:svg** component consistently, this flow ‘agent’ can process any of the children layouts. In the example the rectangles are already sized and a specialist layout agent reads the image data to find its aspect ratio, thus determining width given that its required height has been declared⁶. By using an *XML* tree there is a well-founded coherent tool (*XPath*) to examine both definitions and results within the supporting implementation code. As an example, suppose we wish to layout a series of (similarly sized) parts ‘in a square’:

⁵The implementation shown is $O(n^2)$, due to the **sum(preceding-sibling::*)** calculation – a tail-recursive recasting can make this $O(n)$. Evaluating the horizontal flow is similar.

⁶Some other SVG primitives (circle,path etc..) need pre-encapsulation to provide size and translation but this can be handled coherently by very low-level leaf-processing agents.

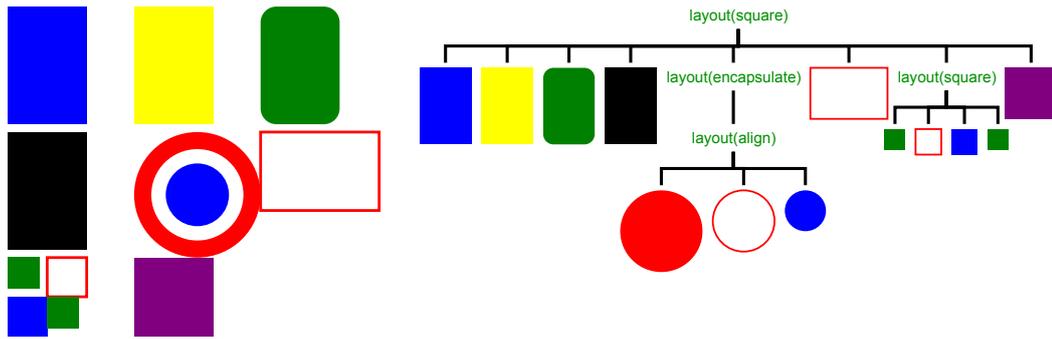


Figure 28. Simple ‘square’ layout & source declaration

Our intention can be described in a document by constructing a tree over all the children, with a parent node that i) declares ‘layout in square’ and ii) adds any other configuration parameters such as outlining, spacing etc. as attributes. The implementation is written as an XSLT template matching the type of node shown at the top of Figure 28:

```

match:ddfl:layout[@function='square'] mode="ddfl:layout"
  (element(*) children=
    => * mode="#current"

  row-length=ceiling(math:sqrt(count($children)))
  size=max(($children/@height,$children/@width))
  svg x="0" y="0" width="{ $size * $row-length}" height="{ $size * $row-length}"
  ∇ $children :
    copy
      copy-of(@*)
      @x=$size * ((position() - 1) mod $row-length)
      @y=$size * ((position() - 1) idiv $row-length)
      copy-of(*|text())

```

There are a few points to note:

- Layout command element nodes are represented as **ddfl:layout function="xxx"**, reserving the value of the **function** attribute as the discriminant – it has to be unique across all the layout agents in an implementation and definitions/descriptions of the semantics of that layout. A ‘catch-all’ agent can discover the use of ‘unknown’ or mistyped layouts.
- The first usual action is to evaluate (render) all the children – they could themselves be layouts and the recursive application of templates handles this succinctly. (In our example one of the children is itself a ‘square’ layout.) Note that the results do not have to be in one-to-one correspondence with the children – the layout for one child might be several distinct items, or for another it might bizarrely be null (see the next section for an example).

- Normally these rendered children are examined, both as a group and individually, to determine some properties of the eventual layout. In this example the number of cells on the square edge (**row-length**) is found by counting the number of elements – we could have counted the number of children pre-rendering, but this would have been incorrect if any of the children were rendered to a non-unity set. The size of the cell is calculated by finding the maximum dimension of all cell contents.
- Usually the rendered children are altered no more than by applying some simple affine transform – most usually by translation. The order of the children is usually preserved, permitting appropriate use of overlapping and obscuration for layouts that depend on such effects (e.g. centering a set of pieces on top of each other). But this is not mandatory and the processing ‘agent’ can examine and reconstruct the returned children in any way it wishes.
- Finally the results are (usually) grouped and possibly decorated (background, borders etc.) into a new rectangular group (**svg:svg**), whose size can be determined, and which can be translated easily by any code processing *its* parent layout.

This approach can be developed for a wide variety of primitive forms combined in cascaded trees. Simple examples include:

- Flows in horizontal or vertical directions with definable spacing between elements.
- Distribution of pieces in horizontal and vertical directions.
- Alignment of a set of pieces by edges or centres or even fractional dimensions.
- An encapsulator that takes a collection of pieces and combines them unmodified into a group with optional padding, margin and border properties. This then provides a single entity with appropriate rectangular bounds for a parent to further position with respect to siblings. Simple extensions can include colouring the background, adding shadows or even using shapes such as page or tab-folder icons for the background/surround ⁷.
- Positioning pieces along a described path.
- Sorting a set of children based on size, or filtering out a set of pieces outside a defined range of size.

⁷Many of the other layout combination functions use the code for this encapsulator to produce a coherent single group result to their parent.

6.2 Text layout

Almost every document requires text of some form. DDF uses part of the syntax and semantics of XSL-FO to describe text intent, particularly the **fo:block** which defines a sequence of ‘decorated’ text that is to be laid out with some provided common properties, both stylistic (font, colour, size) and associated with paragraphs (line spacing, justification, hyphenation...). The **fo:inline** element is the main means to alter text style properties within the block.

Within XSL-FO such blocks act as content stream definitions to be flowed into containers which provide the dimensional constraints. In DDF the **fo:block** can stand on its own, so a few additional attribute properties (**@width**, **@max-width**, **@single-line**) were added. The model is to line-wrap the text into a semi-infinite column of the given width returning an **svg:svg** group containing lines of **svg:text** and **svg:tspan** descendants with appropriately positioned and styled characters in SVG space. (The lines are represented as **svg:svg[@ddfl:element-type="text-line"]** structures, which allows breaking paragraphs across container and page boundaries.)

The size of the block generated is provided on the group (**@width,@height**) – if the attribute **@max-width** is specified, the width returned is the minimal bounding width, otherwise the actual width specified is returned. Group decorative properties (margin, border, padding, background) can be specified and visible elements to display that will be part of the group returned.

The implementation of this transform is carried out with a rather complex Java class that has to collect all font statistics and proceed to execute a modified Knuth-Plass algorithm to decide line break points in the presence of hyphenation and multiply-styled strings, keeping a close correspondence between the original **fo:definition** and the resulting **svg:** structure. In particular if a portion of text is surrounded by an **fo:inline** in the source, it *must* be surrounded by an equivalent **svg:tspan** (or perhaps **svg:text**) element in the result, complete with any ‘foreign’ attributes that the original element had. This is a vital part of ensuring that information can be transmitted across the generation of presentation and used for other purposes of higher-level manipulation later. A very good example of this is marking pieces of text as page-number place holders (**@ddf:page-no="."**) which can then be picked out of a page subsequently and replaced with the appropriate number.

A simple extension introduced the possibility of providing a path-defined container via the

@svg:path attribute somewhat similar to [91]. The implementation supports simple paths defining single regions without holes and which for which unique left and right boundaries exist for any vertical position (i.e. there is no horizontal re-entrancy). This is used for some more advanced text layouts described in the next section.

Finally an approximate method for ensuring a text block just fills a shape by altering the font-size (*font warping*) can be implemented in a simple two-pass activity. When an **fo:block** has defined size (**[@width and @height]** and declared with a mutable font-size (**@mutable-font="8pt-12pt"**) then such cases can be intercepted before normal processing. The font size is set to some default on a copy of the **fo:block** which is then evaluated. The result is then examined to find its area and assuming there are enough characters for the text to approximate a ‘liquid’, the font size is adjusted by $\sqrt{area/area_{test}}$. The new estimate is limited or quantified as required, and written again onto the final **fo:block** which is then evaluated and returned as the result. The entire code is some 40 lines long. Other approaches to this, with more complex shapes, are covered in [25, 40]. Little more about text needs to be discussed here; *all* the text you are reading has been generated by these means.

6.3 Advanced layouts

The previous section described the general architecture for declaring layout presentations and the implementation of their evaluation. Many layouts are simple movements of subcomponents, based on their geometry, but we can have other special-purpose forms of layout. Two relatively simple examples illustrate this and show how more advanced ones can be built.

The first example is one where some content (perhaps a title or caption) is modified to maximise contrast against a non-uniform background. In the simplest case assume that the background is a picture and text is to be written with a contrasting colour, such as Figure 29: in one case the text is treated as a block and given an ‘average’ background, in the other text colour is altered on a word-by-word basis.

```

layout(image-text-contrast) word-split="yes"
  image xlink:href="LumleyGimsonRees.JPG" width="150"
  block x="55" y="20" font-family="Helvetica" font-weight="bold" font-size="7"
    John Lumley
  block x="2" y="85" font-family="Helvetica" font-weight="bold" font-size="7"
    Roger Gimson
  block x="90" y="80" font-family="Helvetica" font-weight="bold" font-size="7"
    Owen Rees

```

■ ddfi: ■ svg: ■ unknown: ■ fo: text

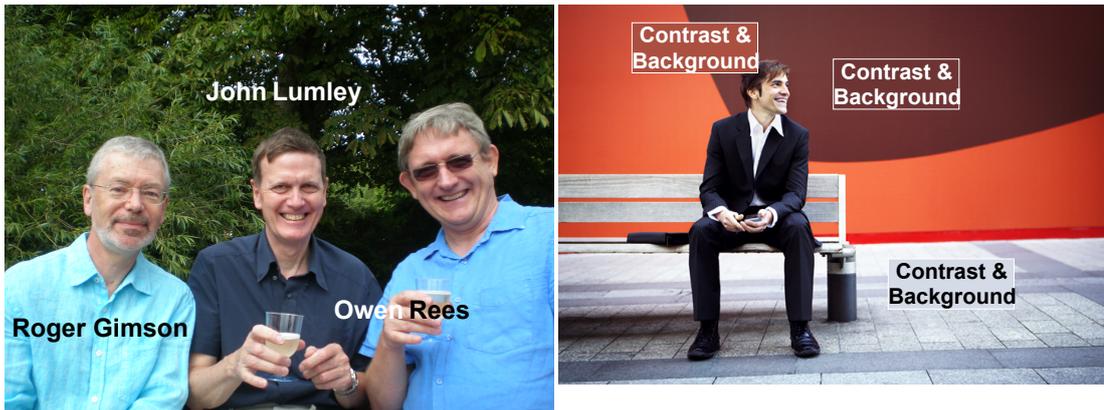


Figure 29. Modifying text texture to enhance contrast against background

The problem can be broken down into three parts: establishing what the background and foreground pieces are, determining a suitable contrasting colour and then modifying the foreground piece, followed by encapsulating (grouping) the pair to produce a single final result. Constructing the pieces is simple: the layout processor is invoked through **xsl:apply-templates** on the subtrees in the definition. Once the necessary changes have been determined then simple XSLT code can produce modified (SVG) trees for the result, by changing the **@fill** property. The new functions needed are to determine what colour the foreground piece should be. To do this we need to ‘look’ over the background image in its colour pixel space.

For pictures a single function **pic-color(URL, x_{low}, y_{low}, x_{high}, y_{high})** will suffice. This function would return the ‘average’ pixel colour (i.e. red, green and blue components) of a given rectangle within the image. Such a function can be implemented easily in Java. In the example above the function is used to measure the colour of the image behind the given text: a contrasting colour for the **@fill** property of the foreground piece can then be chosen. The same function can assist a more complex layout that positions foreground images in maximal contrast positions over a background.

The second example supports simple floats and drop capitals in text blocks (Figure 30). The paragraph primitive **fo:block** can have two extra attributes (**@text-indent** and **@text-indent-**

depth) which define an initial top-left indent. If we indicate a requirement for an Initial by the attribute **@drop-capital="n"**, then a higher-priority agent can intercept this before that for the default **fo:block**. This removes the first *n* characters and generates a separate larger text (with an optional **@drop-font-size**) for the initial.

In a written or published work, an initial is a letter at the beginning of a work, a chapter, or a paragraph that is larger than the rest of the text. The word is derived from the Latin *initialis*, which means standing at the beginning. An initial often is several lines in height and in older books or manuscripts, sometimes ornately decorated. (From Wikipedia)



ne variation on dropped capitals is rather old. In illuminated manuscripts, initials with images inside them, like those illustrated here, are known as **historiated initials**; they were an invention of the Insular art of the British Isles in the 8th century.

Initials containing, typically, plant-form spirals with small figures of animals or men that do not represent a specific scene are known as "inhabited" initials. Certain important initials, such as the B of *Beatus vir ...* at the opening of Psalm 1 at the start of a vulgate Latin psalter, could occupy a whole page of a manuscript.



Figure 30. Drop capitals and wrap around images

The initial text, being generated consistently by the layout processor, is returned as an **svg:svg** fragment and assigned to a variable *drop*. Its size can be found by the XPath expression **\$drop/(@width,@height)** and copied onto the text indentation properties for the (shorter) text block *text*, which is then evaluated in turn. Finally both *drop* and *text* are encapsulated and returned as a single **svg:svg**. Text can be flowed around images or other content on the container edges similarly, calculating a suitable **@svg:path** parameter for the **fo:block** after the wrapped entity has been evaluated⁸.

By defining and evaluating layout by parts in this manner we can build other useful effects such as skeletonisation – removing everything that is smaller than a given size or area from a completed layout, acting as a ‘level of detail’ filter.

Another useful type can be considered a *meta-layout* function. An example is a layout that *evaluates to a null result if it has siblings*, i.e. it evaluates its children if and only if it is an only child of its parent⁹:

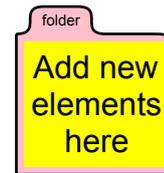
⁸The dependency is acyclic, so this could also be achieved through the use of presentational variables generated by document-borne code.

⁹*Meta-layout* functions will become important in Chapter 9

```

layout(flow) encapsulate="shape:folder;label:folder;label-font-size:2;
background-color:pink;stroke:black;padding:1"
  layout(only-child)
    block background-color="yellow" max-width="25" padding="1"
      text-align="center" font-family="Helvetica" font-size="12pt"
        Add new elements here

```

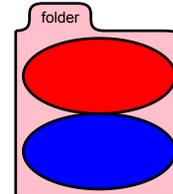


a) Empty

```

layout(flow) encapsulate="shape:folder;label:folder;label-font-size:2;
background-color:pink;stroke:black;padding:1"
  layout(only-child)
    block background-color="yellow" max-width="25" padding="1"
      text-align="center" font-family="Helvetica" font-size="12pt"
        Add new elements here
    ellipse cx="10" cy="5" rx="10" ry="5" fill="red" stroke="black"
    ellipse cx="10" cy="5" rx="10" ry="5" fill="blue" stroke="black"

```



b) With filling

```

match:ddfl:layout[@function='only-child'] mode="ddfl:layout"
  if:empty(..*[not(@function='only-child') ])
    => * mode="#current"

```

■ ddf: ■ fo: ■ svg: ■ xsl: ■ unknown: text

c) Processing agent code

Figure 31. ‘Only child’ Layout

This layout is useful when the presence of some form of layout needs indication, even if it is empty, such as during document editing, where a ‘new’ flow or other layout is added. Usually such layouts evaluated over a null sequence of children either produce a null result or perhaps some boundary decoration around a ‘null-point’: this may be invisible and thus cannot be selected during editing.

If the new layout is added from a template where the archetype contains a suitable description wrapped in an ‘only-child’ guard, this only-child will immediately appear in the final result and can act as a mouse-target when the initial layout is empty. As soon as another child is added, the description disappears. If, later, all other children have been deleted, the layout description/target will re-appear. Implementing this only child is trivial, guarding evaluation by `empty(..*[not(@function='only-child')])`¹⁰.

Finally, perhaps mainly of interest for documents in computer-science research, a structure of

¹⁰The code allows for multiple ‘only-child’ children – which could support some interesting effects. The simpler `empty(preceding-sibling::* | following-sibling::*)` supports just a single ‘only-child’.

combinations of subparts can often be explained in the form of a graphical tree representation – a node placed above and connected to a set of subtrees, these subtrees placed close together, preserving order and not overlapping each other. We could define a layout function **tree** that contains information about display of the node itself, either as a reserved child or attribute. Other children of the layout would then be laid out in close proximity to each other, which requires finer boundary detail than the bounding rectangle to get usefully dense results. If any of the children themselves are **tree** layout functions then they would themselves provide trees-graphical results, otherwise they are treated as any other form of layout.

In fact, all the tree examples shown within this thesis are generated in a slightly different way as the original need was for graphical display of XML trees. The **tree** layout function assumes that everything below is XML to be displayed by default in text-and-line form. However specialist nodes can alter the processing behaviour for the element node concerned: **@ddfl:evaluate** produces an atomic piece by evaluating the subtree as a complete layout; **child::ddfl:caption** or **@ddfl:caption** define the caption to use, either as a graphical piece or a textual form, **@tree:polygon** requests a shaded background for this element. Other options, such as removing nodes or eliding repetitive sequences can be supported by pre-processing. But the principle of *unknown information transmission* still applies and the resulting tree nodes *will* contain attributes from the original – this allows pointing into specific places within tree diagrams. For more details of this see [60].

6.4 Non-local effects

In the previous sections, all layout has been *local* to the specific part of the definition tree. Detailed and variable parameters for the specific definition requested can be decided during the XSLT phase of document generation, e.g. altering spacing or background because of the presence of a particular type of child. But no information from the rendered layout of *another part of the document* can be used in determining the layout. Only information from the tree below the node can be used, either from pre- or post-rendered results. As an example consider the layout effect shown in Figure 32.

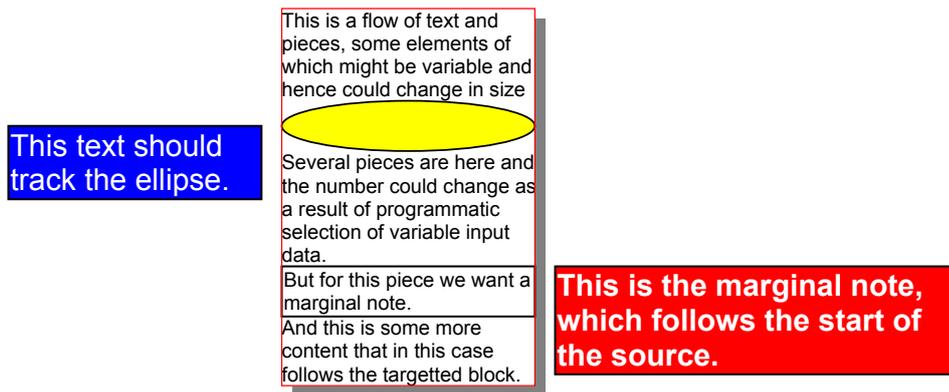


Figure 32. A tree-breaking example: tracking pieces within a flow

Here we want to position two parts (marginal notes) in positions relative to specific elements within the central flow layout. But those parts are *not* part of that flow, having no influence on the flow itself – its size or the ordering or relative positions of its constituents. In tree terms (Figure 33) the problem is to place the two higher-level blocks at vertical positions determined from where their anchor points end up after rendering the main flow:

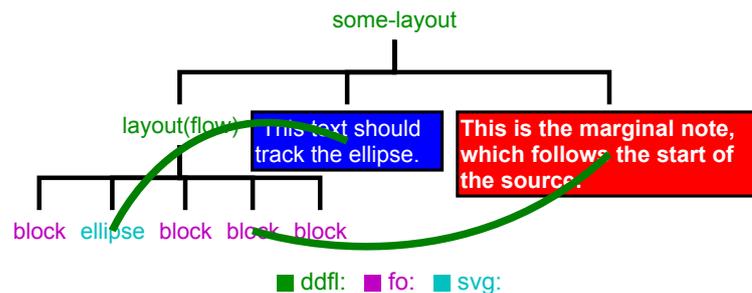


Figure 33. The layout requirements of Figure 32

Clearly there has to be some mechanism for the marginal note blocks to identify their anchors within the original definition of the flow. This problem is obviously not local to the flow itself (unless we complicated the flow with other specific ephemera such as marginal notes), but belongs to some parent. We could develop a special-purpose layout, which detected the presence of inline marginal notes, removed them, remembered the anchor elements, laid out the flow and then examined the positions of the anchors in that rendering and positioned the (rendered) marginal note blocks at the appropriate positions, finally wrapping up the whole assembly.

In some circumstances something as complex as that might have to be done, but this is a fairly simple problem which has *acyclic* dependencies between the flow layout and the layout of the notes. That is, it is possible to write a finite sequence of expression evaluations that will lead to the final solution, without having to attempt any simultaneous solutions. (The next sec-

tion will consider cyclic and near-cyclic problems.) So we could think about a system where such evaluable expressions can be written in the presentation description, with a common evaluation engine, enabling a wide variety of such problems to be tackled by the document engineer writing ‘code’ in the presentation, rather than adding further layout agents to an implementation platform.

Acyclic dependency and single-assignment presentational variables

The solution was a system of *single-assignment presentational variables*, which unsurprisingly, had a lot in common with the variable system within XSLT. Basically a variable can be bound to the *rendered* result of some layout, and then a copy of that variable, or an expression over values of several variables, can be interpolated into another layout, which is then subsequently rendered. Figure 34 shows a very simple example:

```

<ddfl:variable name="flow">
  <ddfl:layout function="flow" direction="x" encapsulate="line">
    <svg:circle cx="5" cy="5" r="5" fill="red"/>
    <svg:rect height="10" width="15" fill="green"/>
  </ddfl:layout>
</ddfl:variable>
<ddfl:layout function="flow">
  <ddfl:copy-of select="$flow"/>
  <svg:rect width="0.5" height="5">
    <ddfl:attribute name="x" select="$flow*/@width"/>
  </svg:rect>
  <fo:block font-family="Helvetica" font-size="12pt">This is <ddfl:
value-of select="$flow*/@width"/>mm wide </fo:block>
</ddfl:layout>

```

■ ddfl: ■ svg: ■ fo:

Figure 34. Using a presentational variable to recover information from a layout

The **ddfl:variable** statement binds the result of a rendering into a named variable – actually it can be a sequence of **item()** pieces, so for example we can bind to a property of the result, such as a dimension, or a complete set of nodes within some part of the presentation. Names of variables follow usual tree-scoping, so they do not have to be globally unique, and can be ‘overridden’ deeper in the tree if needed¹¹.

XML tree values of variables can be interpolated with **ddfl:copy-of**. Scalar values of variables can be added through **ddfl:value-of** which can have expressions over variables. In our example we use this to print the width of the horizontal flow by adding the string value of the width

¹¹Appendix B shows using this for indented list and tabular layout in paginated documents.

to a text block *before* it is rendered. References to variables (**\$name**) are interpolated just like in XLST, and expressions are written in XPath2.0. This allows search through the tree values of the variables for sub-trees, properties, conditions etc.

A **ddfl:attribute** command allows us to add to an element's properties *before* it is rendered, so the layout of that piece can be altered. In our example we position the narrow rectangle horizontally at the end of the flow, by setting the **x** attribute to the width of the rendered flow¹².

Implementing this variable system is not as difficult as might be imagined, though some of the coding is rather 'delicate'. There are two key points, both related to the layout processor being written mainly in XSLT.

Firstly the layout processor consists, as we have seen, of a 'sea' of agents matching layout declaration intent nodes. Some agents can be promoted to higher priority and pre-empt others for processing the same node. This means that we can pre-emptively process nodes that contain presentational variable declarations and interpolations to generate a new tree within which *all* the presentational variable content has been evaluated and interpolated. This new tree is then 'thrown' back in for processing by its usual agent, whatever that might be – this is a critical point as it means these acyclic variables can be used in *any* form of layout. Indeed almost all layout agents have *no knowledge* of the existence of such variables – in fact they never see any children of such forms.

Secondly, using a simple XPath processing extension, we can generate and process variable XPath expressions at runtime within this agent, which means we have the ability to search within variable tree-values and evaluate expressions as thoroughly as within XSLT.

This is a **key** feature – the use of indirect evaluation of XPath expressions is critically important. In XSLT3.0 this will be possible as standard; in XSLT2.0 a fully-featured extension function supports this.

The pattern ***[ddfl:variable]*[ddfl:copy-of]..** can recognise cases containing variables, both in definition and use. The XSLT template matching this case is placed at very high priority, and having gained responsibility for processing the node proceeds through a recursive process to evaluate all the children. The node is now reconstructed *without* contained variable defin-

¹²There is no equivalent of XSLT's *attribute value template* allowing values to be written directly into element attributes (e.g. **width="{3 * \$piece/@width}"** – there is no spare character sequence).

itions, but *with* the variable interpolations completed, and then submitted for reprocessing (and the children have already been fully evaluated – it is possible to mark such children so that re-evaluation is not performed.)

Since presentational variables may have sequence values of arbitrary length, they are actually stored as a stack frame of values (**ddfl:variables**) pointed to via a lookup index (**ddfl:variable-names**) which identifies position and length of the appropriate values. The last matching index for a given variable name is used: in this way local names can over-ride remote ones smoothly and variables can even be redefined in terms of their previous values. The variable storage is transmitted between layout agents as a set of ‘tunnelled’ XSLT parameters, which naturally follow appropriate processing of tree structure¹³.

Now we have explained enough to show how the marginal notes can be declared in a document:

```

layout(flow) direction="x" spacing="4" overflow="visible"
  main=
    layout(flow) encapsulate="background-color:white;stroke:red;shadow:2" overflow="visible"
      block font-family="Helvetica" font-size="4" width="50"
        This is a flow of text and pieces, some elements of which might be variable and hence
        could change in size
      ellipse name="ellipse" cx="25" ...
      block font-family="Helvetica" font-size="4" width="50"
        Several pieces are here and the number could change as a result of programmat-
        ic selection of variable input data.
      block name="marginal" font-family="Helvetica" ...
        But for this piece we want a marginal note.
      block font-family="Helvetica" font-size="4" width="50"
        And this is some more content that in this case follows the targetted block.
    block font-family="Helvetica" font-size="6" width="50" ...
      @y=$main/*[@name='ellipse']/@cy
      This text should track the ellipse.
    copy-of($main)
    block font-family="Helvetica" font-size="6" font-weight="bold" ...
      copy-of($main/*[@name='marginal']/@y)
      This is the marginal note, which follows the start of the source.

```

■ ddf: ■ fo: ■ svg: text

The top-level layout is a horizontal flow. Within this we evaluate the main text flow and assign it to the presentational variable **main**. Within the definition of the flow the two anchor points (the ellipse and a following paragraph) have been given ‘names’, in this case through an attribute **@name**. The layout agents are ‘good XML citizens’ (see section 9.5 for more details) and thus these **@name** will appear in the rendered result attached to the relevant graphic compon-

¹³See section 2.2 on page 37

ent and can be used to search for the element in that result. To track the ellipse in the flow, we add `@y` to the `fo:block` with the value from `$main/*[@name='ellipse']/@cy`. For the marginal note we copy the `@y` from the anchor paragraph.

What we have done here is to provide a *document-borne* layout computational mechanism to solve certain types of problem (i.e. acyclic geometric dependency) without resort to building a new computational layout agent. So it should be possible to extend the document vocabulary to cover more complex arrangements through suitable mappings between structural and presentation space.

As an example, assume our text flow is mapped from an XHTML sequence of paragraphs within a division: `div` → `ddfl:layout function="flow"`, `p` → `fo:block`. We want to add marginal notes (which we will assume are all to be held in the right margin) and decide to add them as `@note` attributes to `p` elements as needed. Some simple XSLT code in the structure-presentation mapping can suffice:

```

match:div[p/@note]
  flow=
    layout(flow)
    ⇒ *
  copy-of($flow)
  ∀ p[@note] :
    block
      @x=$flow/@width
      copy-of($flow/*[@name='{generate-id(.)}']/@y
      )
      val(@note)

```

```

match:p[@note]
  temp=
    copy
      @* except @note
      @name=generate-id(.)
      *|text()
    ⇒ $temp

```

■ xsl: ■ ddf: ■ fo:

Paragraphs with notes have a unique identifier added within the flow (`generate-id()`). The notes are then collected into individual text blocks and their position set to be interpolated after the flow has been laid-out: horizontally to the width of the flow (`@x=$flow/@width`) and vertically to that of the corresponding anchor (`$flow/*[@name='{generate-id(.)}']/@y`), the identifier having been placed in the XPath expression during the XSLT phase.

This concept of document-borne simple acyclic presentation computation can be extended to add several useful analogous features from XSLT semantics: conditionality (`ddfl:if`), choice (`ddfl:choose/(ddfl:when|ddfl:otherwise)`) and iteration (`ddfl:for-each`). This will be discussed more fully in Chapter 9, but a typical example using it is shown in Figure 35.

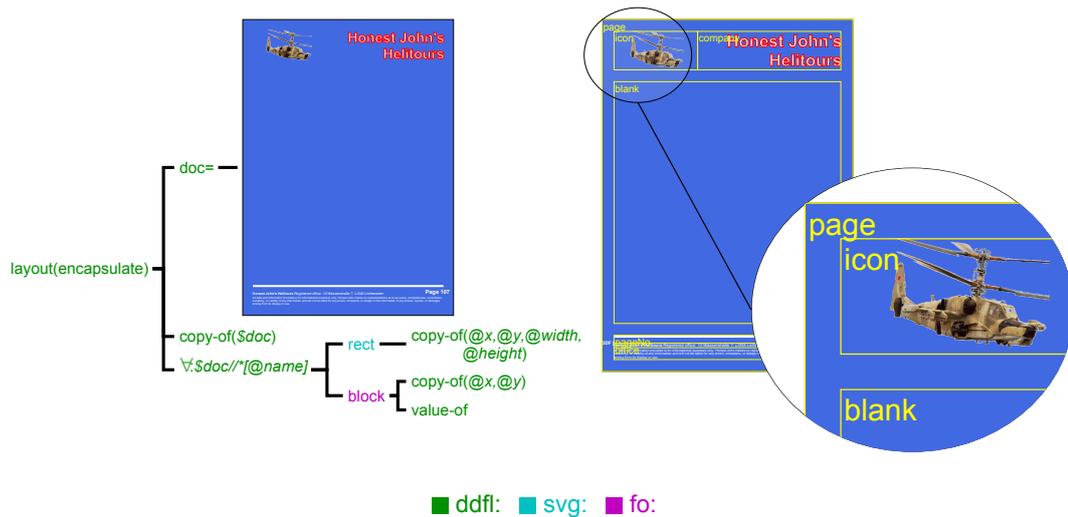


Figure 35. Identifying named pieces

In this case we have a portion of graphics with a number of pieces decorated with the attribute **@name**, probably used to denote sections that were used in construction or are available as ‘containers’ for further work. The result of evaluating the presentation is assigned to the variable *doc*, which is then interpolated without alteration. Then for each of the pieces **\$doc//*** **[@name]** a rectangle that shows the extent of the piece (**@x|@y|@width|@height**) is generated as well as a positioned label (**@x|@y, value-of(@name)**).

Cyclic dependencies

Some required layouts have dependencies or constraints between components that are mutual. A typical, though actually quite hard, example is a column containing a picture beside a (caption) paragraph, where the desire is for them to be the same height and together just span the entire column. The picture may be resized but will preserve its aspect ratio; the paragraph will roughly preserve its area but may have a variety of widths and heights. A model for ‘large’ paragraphs (lots of characters/words) yields four simultaneous equations, which have a non-deterministic analytic solution as the roots of a quadratic. Thus there are limits for real solutions in terms of a minimum column width, above which there are always two solutions.

Practical approaches to solving more general versions of this require complex approximations and searches using non-linear or quasi-linear approaches[39, 40, 57]. The integral nature of the paragraph as a set of lines introduces additional non-determinism in the solution space. In practice a series of approximations, coupled with using micro-adjustments such as line-spacing and margins can be used to generate acceptable solutions with small amounts of text.

In the most general cases we can consider all layouts to be described by a set of constraints between component parts – the nature of these constraints govern whether, and how, solutions can be found to them. Generally these require significant controlled searching algorithms with heuristics – see Hurst[36] and di Iorio[15] for examples.

An important and useful class however is when the constraints are linear and continuous over their variables – either with equalities or inequalities in the defining equations. These are important for two reasons: firstly many interesting constructs involve particular types of linear constraints between sets of pieces and secondly there are very efficient solution implementations, exploiting dynamic programming, capable of handling tens of thousands of constraints arranged effectively in sparse matrices. As an example a table can often be described as a network of constraints between the row/column boundaries and the cell contents – this can be especially useful when the cell contents have a large range of possible sizes with their own constraints, such as a picture, where the ratio of height to width is defined but size is not defined tightly. Internal cell-content constraints can be added to the general one defining the table geometry, to produce an overall set of constraints for the complete table which are then solved completely.

This class was sufficiently useful to warrant defining a generalised linear inequality constraint layout. As before the declaration is a node (**ddfl:layout function="linearConstrained"**) which contains both presentational children and the constraints between them. These constraints could be declared in several ways – by explicit reserved children (**ddfl:constraints**), on attributes (**@ddfl:constraint**) and with other properties on the node ‘call’.

We have to describe a graph of geometric relations between pieces, so each piece must be named in some manner. We use a **@name** attribute on each component which is unique within the local scope – after they have been evaluated (laid out) the resulting SVG element will retain that local identification. Then the self-constraints (width, height) can be determined by inspection and used to create suitable constraint expressions, e.g. **piece_right - piece_left = 45**.

The Cassowary[4] constraint solver was used – advantageously, it handles linear inequalities, provides a set of different strengths and is implemented in Java. The layout agent for this solver builds the ‘edges’ of the constraint graph from the declared relationships and the implicit sizes (e.g. **ajacent(left,+10)** between *bullet* and *para* becomes **bullet_right + 10 < para_left; strength:medium**, images have implicit aspect-ratios: **image1_top - image1_bottom = 0.754 * (image1_right - image1_left); strength:strong**).

Once the XLSL code has built all the constraints, they are passed to the Java API which then solves for the values. The return values (**bullet_right = 234.5**) are then decoded and appropriate properties (**@x,@y,@width,@height**) written onto the relevant component children.

By appropriate use of such constraints it is possible to describe *inward-facing* layouts, such as *packers*, where components are defined to lie within and take size resource from their enclosing parents in a recursive manner. An experimental packing layout was implemented which used a mixed strategy of computing all acyclically determined aspects and invoked constraint solution for the remainder. The most significant problem was that of text-paragraphs where the non-linear nature of the height-width relationship limited the widespread effectiveness of the approach.

Post-presentational global effects

In a practical system there are some effects that are global in scope and act after the main generation of presentation. The simplest examples revolve around page numbering and referencing, though the techniques involved are not specifically tied to the act of pagination, which is described in the next section. The basic requirements are:

- Interpolate the number of the current page into some text – usually this is in some fixed position (i.e. defined in a page template) and may be subject to various styles of numeric formatting. For example this text is sitting on page *109* or *cix* if you are Roman.
- Interpolate the (textual) number of the page on which some other part of the document lies – a *cross-reference* such as to the section on acyclic-dependency, which is on page *103*.

It is clear that in-place and backward page references could be interpolated as pagination proceeds, but forward cannot. It is also the case that on the very detailed scale the problem is indeterminate. The character length of a number varies dependent upon its order of magnitude, and real corner cases could exist where the number lies at a critically sensitive layout position, such as a line-end with page-breaking consequences. However if we ignore those rare corner cases, and the numerically-variable ‘size’ of the references is small, then we can look at the problem as a straightforward post-process of a presentation, where there is a simple canonical representation of points that are i) reference targets and ii) page-number interpolants.

Using our general approach, this can be invoked by a specialist (though simple) layout function **number-pages**, placed above the definition of the set of pages, that rebuilds the tree, look-

ing for (laid-out) text fragments marked `@ddf:page-ref`. The value of this property can determine what to do – a string pattern is assumed to be some *id* of another part of the layout – it is searched for and the number of the page it is found in is used. The default would be the number of the current page. Simple expressions (e.g. offsets) and numeric formats can be added as additional properties. The resulting number as a string then replaces the text fragment, which has been set to some suitably sized placeholder such as *XX*.

This approach leaves some additional whitespace or risks overlap, but a simple two-pass procedure could be used to avoid this, albeit at the cost of repeating the entire layout, and still with some minor risk of result indeterminacy¹⁴. A trial layout is performed, the page references of all points determined, the page reference source elements replaced with the appropriate textual representation and then the whole layout repeated, without further page number interpolation¹⁵.

This may appear to be somewhat limited, but the existence of this feature late in presentation, coupled with suitable document-borne functionality in the structural and presentational spaces means a lot of page-number related features are now fully-supportable. Tables of contents are often needed. Much of the decision on what to table in the contents is a logical structure issue (‘how important is this part of the document?’), but we still need to interpolate the page numbers. The mechanism to do this is simple – it is just a large set of cross-references at the start of the document¹⁶. A similar search-and-substitute mechanism is used in this thesis to interpolate the running footer (*Layout in DDF*).

6.5 Pagination

Most of the layouts described previously can be considered to be generally ‘linear’ functions over their children – concepts of superposition have reasonable meaning. But there can be layouts that are distinctly non-linear: the most common and important example of this is pagination. In its simplest case, pagination is an order-preserving one-dimensional bin-packing problem, which reduces to partitioning a sequence of pieces into groups each of which has the largest ‘length’ less than or equal to some target. (This does not mean pagination is just about text

¹⁴Other document systems, such as Scribe and Lout face the same problem and tackle it in a similar manner

¹⁵This might be warranted for a final pass on an important document, such as a PhD thesis.

¹⁶Minor enhancements could make this capable of handling multiple page-number sequences in a single document (i-xxiv, 1-234 etc)

– any types of content can be included.) Practical pagination is not quite as simple, adding a variety of increasingly complex features: paragraph splitting across pages, multiple columns, footnotes, floats, headers-and-footers, page-variable templates, widows-and-orphans etc, etc.

We can declare basic requirements for a pagination very simply – deciding the semantics and constructing an implementation is not quite so straightforward. There are two general types of argument: ‘containers’ to be filled, presumably in some sort of sequence, and ‘filling pieces’, whose order should generally be maintained.

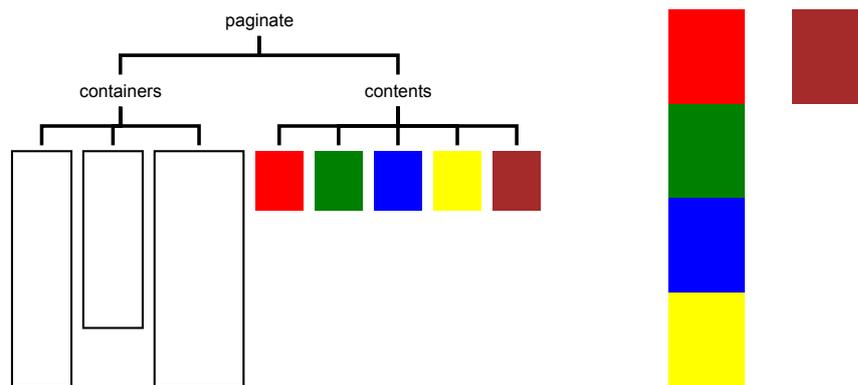


Figure 36. Simple pagination

In the simplest case (Figure 36) the containers are really one-dimensional holes (with a height) and the heights of the filling pieces are used to determine whether each piece will fit in the remainder. Remaining general we assume that the pieces could be either primitives or compounds composed of other pieces or compounds – a compound would be evaluated to a sized group (or set of groups)¹⁷ and then all processed in document order. The implementation is a tail-recursive function¹⁸ that takes as arguments:

- **filled**: containers that are filled ‘completely’
- **current**: pieces that are already ‘in’ the current container
- **containers**: sequence of containers – the head of which is being filled
- **depth**: current place to pack the current container
- **pieces**: parts awaiting placement

¹⁷A compound could even be another ‘pagination’ to give a close-packed variable data effect – implementations are fully recursive so this is distinctly possible.

¹⁸Lout[50] uses a tree-transforming implementation to do the same thing, moving evaluated content pieces from the ‘source’ subtree into container subtrees, generating new empty containers as required – the effect is the same.

Once this basic framework is working then we can start to add the ‘bells and whistles’ that are needed for a reasonable paginator. The key is in the central test of the implementation:

```
(element()) next=  
  ⇒ $pieces[1]  
choose  
  when:$depth + $next/@height lt $containers[1]/@height  
    paginate($filled, ($current,$next), $containers, $depth + $next/@height, rest($pieces)  
  otherwise  
    this-filled=encapsulate(($current-contents, $next)  
    paginate(($filled,$this-filled), (), rest($containers), 0, $pieces)
```

Basically the next piece is evaluated and then subjected to a series of tests. In the case shown either i) the piece fits, in which case it is added to the current bin and then recursion focusses on the next piece with a new depth, or ii) it does not, so the current bin is closed and the part is tried in the next container. Some end cases needed to be added (no more parts, no more containers) and infinite guards implemented (e.g. dealing with a piece larger than any remaining container). How this approach can be extended to produce a large-scale practical paginator (the sort that prepared this thesis) is described in more detail in Appendix B.

6.6 Conclusion

Lector, si exemplum requiris, circumspice.

This chapter has described the syntax, semantics and implementation of a extremely extensible layout system which can support a wide variety of layouts, as well as constructed acyclic layout forms that can be generated within variable documents themselves. Part C discusses what the layout system looks like in terms of functions and functional behaviour together with how it can be extended to support evaluation in the face of partial data binding.

Chapter 7

Example Document - a Travel Brochure

To complete the discussion on DDF as a ‘simple’ variable-data document framework, this chapter presents a larger-scale document, which uses many of the techniques discussed previously. This example is a multiple-page travel brochure generated from a personal ‘customer profile’ acting as the variable data. The design of the document, its various features and how it is processed are outlined. We also show how, by suitable choice of intermediate canonical forms, the document can be re-used for other purposes.

The brochure¹ contains a mixture of different types of data variability and layout. This document has a lot of computational power and is probably too complex to be created through a WYSIWG editing system, but will illustrate a variety of the techniques that can be employed. Most of the document ‘code’ and data structures will be shown in tree form to re-emphasise the tree-based nature of the documents involved and also show how the use of elements in multiple non-interfering namespaces can be powerful.

Figure 37 shows a single instance of the brochure for a particular customer; more detail will be shown in later sections and larger versions can be found in in Figures 114 and 115 of Appendix A. As can be seen there is a common page background, some salutation, a legal (Terms & Conditions) constituent, resort descriptions built as groups, some generic pictori-

¹A simpler version of this document was used in earlier papers [61, 63]

al pages, some maps and a form for subsequent inquiries etc. Figure 38 show another instance for a different customer and company binding, which now has 8 pages and some additional filler material.

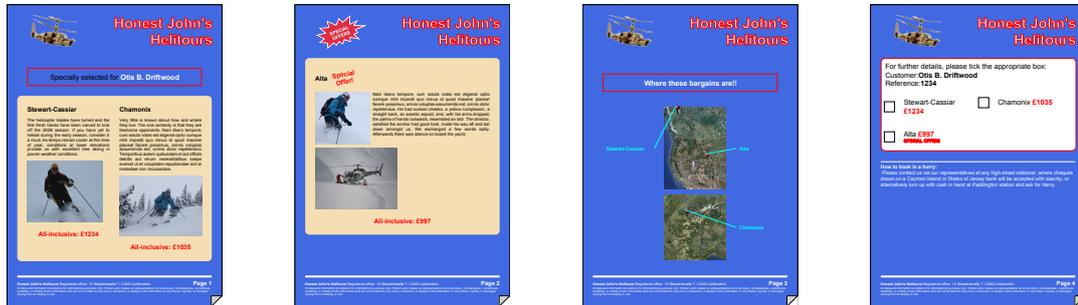


Figure 37. Example instance of travel brochure



Figure 38. A second instance of the travel brochure

7.1 Input data

The brochure is designed to produce a publication for a customer, who is represented by a data record, shown in Figure 39. Given that there is a lot of common data between different customers (e.g. resorts), much of the data is included from other sources, in this case XML files via **xi:include** directives. In this example the company details (which include some filler material), map resources and each of the selected resorts (Figures 40 and 41) are added via inclusions. Details for the resorts vary: some have special properties and most refer to image resources.

```

<brochure>
  <xi:include href="HonestJohn.xml"/>
  <xi:include href="resorts/maps.xml"/>
  <customer>
    <name>Otis B. Driftwood</name>
    <ref>1234</ref>
  </customer>
  <xi:include href="resorts/StewartCassiar.xml"/>
  <xi:include href="resorts/Chamonix.xml"/>
  <xi:include href="resorts/Alta.xml"/>
</brochure>

```

Figure 39. An example customer record

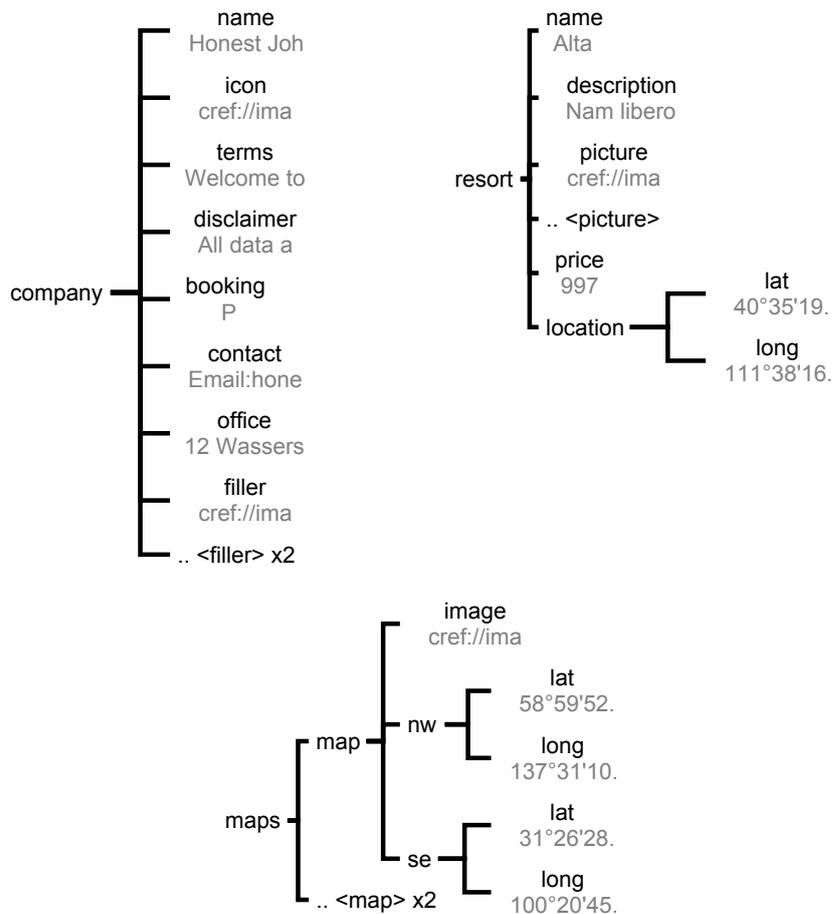


Figure 40. Example company, resort and map details

The map data could come from a database, and geographic locations for a resort might be retrieved from some service, but the essentials of the information we need are given in the two files. With all the data collected for this instance of the document (which will appear in the **data** section of the main DDF document) we can proceed to process it.

7.2 Processing the input data

While this document does not use a standardised logical structural layer, the mapping from **data** to **struct** spaces can still perform useful work, constructing a canonical representation from which the presentation layer can be built, and thus increasing potential reuse. We focus on producing **saleItem** groups, each with **title**, **description**, **picture** and **price** elements, as well as some **diagram** sections.

The inclusions (**xi:include**) are interpolated automatically by the DDF model during the input phase (they are invisible within **data**, save for an attribute recording the source location). The data is then processed by several push-mode templates – one each for **maps** and **resort** elements. The resort is converted to a **saleItem** (with **location** deleted for neatness), but we can choose only the **maps** we actually need and convert them to diagrams in the process (Figure 41). Note that it would be possible to sort sections of the input (for example according to customer preference in **price** order), but this is more a business-logic issue and belongs further upstream – in general we preserve the order between similar items from the original data.

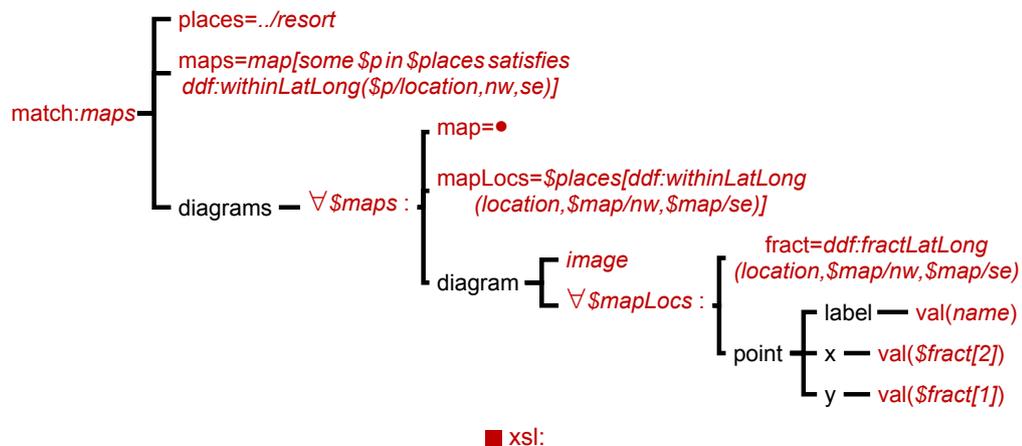


Figure 41. Pre-processing maps

We take each of the maps provided (already decorated with bounding latitudes and longitudes) and predicate them against surrounding any of the resorts. For each of those maps required we represent it as a **diagram**, to which are added each of the enclosed resorts with its fractional position within that map encoded as a **point** with **label**, **x** and **y** children. This processing is assisted by a few functions (**ddf:fractLatLong()** which converts **deg°min'sec"** to numeric degrees, **ddf:withinLatLong()** which determines if a location is within an area) which would conventionally be contained within a library for supporting geographic operations – the point

is that this can be contained within a document that is simply imported for use. By choosing this approach we decouple the presentation from *any* knowledge of geography and open the possible reuse of the presentation in other forms of brochure. Doing this on the first example produces an intermediate data structure shown in Figure 42.

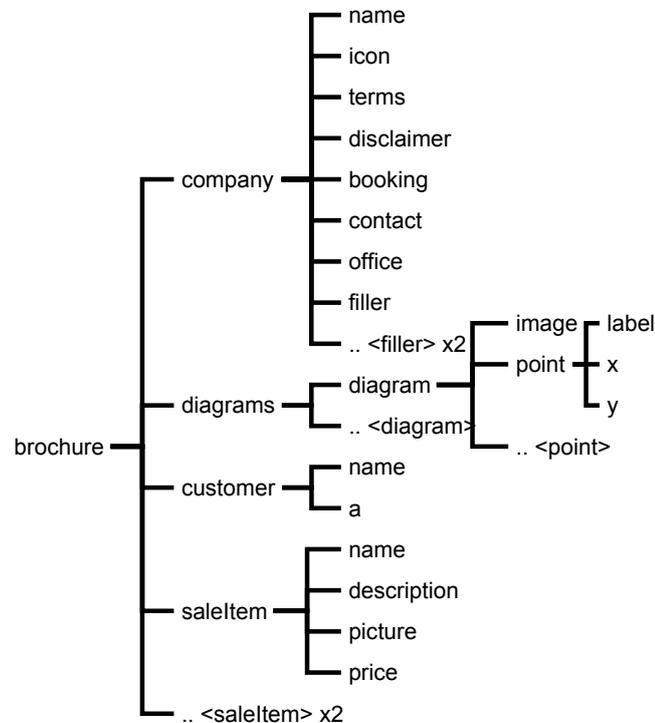


Figure 42. Canonical form for brochure data

With the data now written in terms of **customer**, **company**, **saleItem** and **diagram** items we can turn to the generation of a suitable presentation.

7.3 General layout model

Apart from the overall appearance and styling, the main design choices for the brochure presentation revolve around two aspects. Firstly, is the document going to be a fixed or variable number of pages in length and secondly, what sub-parts of the data are being displayed and how? The use of a canonical form within the document's **struct** space suggests that *all* the elements there should be presented if possible – any deletion of content on *semantic* grounds should have been carried out previously – this section decides how all the data will be presented.

As the number of **saleItems** is likely to be variable, it was decided to use a variable number of pages – from four upwards. There are two choices on how to generate the pages. The

most general and highly suitable if the content is organised as a flow, is to use a presentational paginator (section 6.5). But pagination is computationally expensive. In this case where each item may be considered to be similar in size, and the number of items is known at ‘run-time’ we can use an XSLT iteration to generate a sequence of pages, each containing layout for the appropriate content subsequence. This also makes it easier to insert filler material if we wish to ensure that the brochure has a multiple of four pages, which is greatly preferable for printing.

Document background and common sections

Good practice suggests maximising common sections, for both authoring and processing efficiency. The document background and the company icon are obvious candidates. Some (company dependent) parts of the background and the icon are data-variable which requires prior processing of layout to yield an invariant section that can be interpolated into each page. By assigning this to a *presentational* variable (Figure 43), which is interpolated into the start (i.e. the draw-order background) of each page, rather than an XSLT variable to be interpolated, we ensure that the generation is carried out only once per document instance. This also opens the possibility of positioning further items in places that depend upon the background.

The background uses a constrained layout (section 6.4) to position the necessary items – a **rect** (*page*) acts as the background and provides the page dimensions. Two text blocks (*company* and *office*) display the company name and contact details and are constrained relative to the top right (**align(top,offset=10) align(right,offset=10)**) and bottom left of *page* respectively. A **rect** (*icon*) will act as a ‘hole’ to accept icons at a later stage and is similarly constrained relative to the top left of *page*, as well as having the same height as, and abutting to the left of, *company*. A small text block (*pageNo*) aligns to the bottom right of *page*. This block contains an **fo:inline** element with a **ddf:page-ref="."** attribute – this will be carried through to the resulting layout (as an **svg:tspan**) with the attribute still attached and the layout function **number-pages** (which is an ancestor of this element in the eventual layout tree) will interpolate the final page number into such elements.

Finally a **rect** (*blank*) is constrained to lie between the top and bottom elements. This will be used eventually as a place to position further elements within the blank space in the page – this will adapt as the size of the header and footer elements alter. This technique makes it possible to generate quite complex systems for page template elements which are held in an impor-

ted 'document' rather than having to be buried in some system resource such as a paginator.



Figure 43. Brochure page background

Construction of pages

There are four types of page in the document: i) those containing descriptions of **saleItems** (Figure 44) which includes the first page, ii) a set of labelled diagrams, used here for maps, iii) some optional 'fillers' and iv) a conclusion/order form.

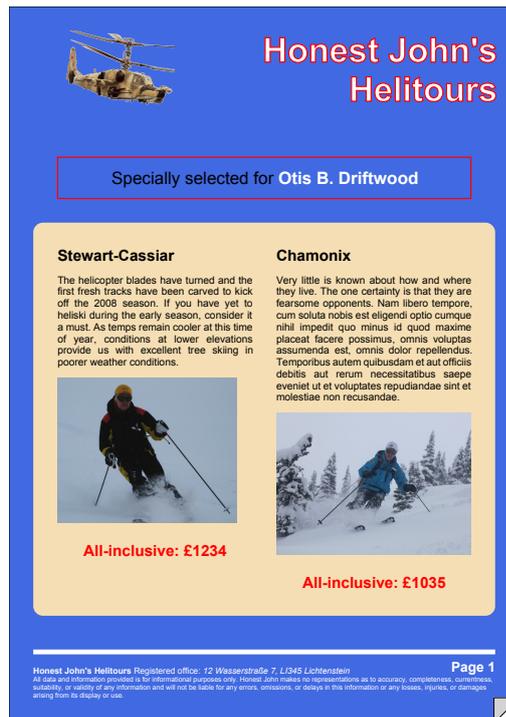


Figure 44. First page

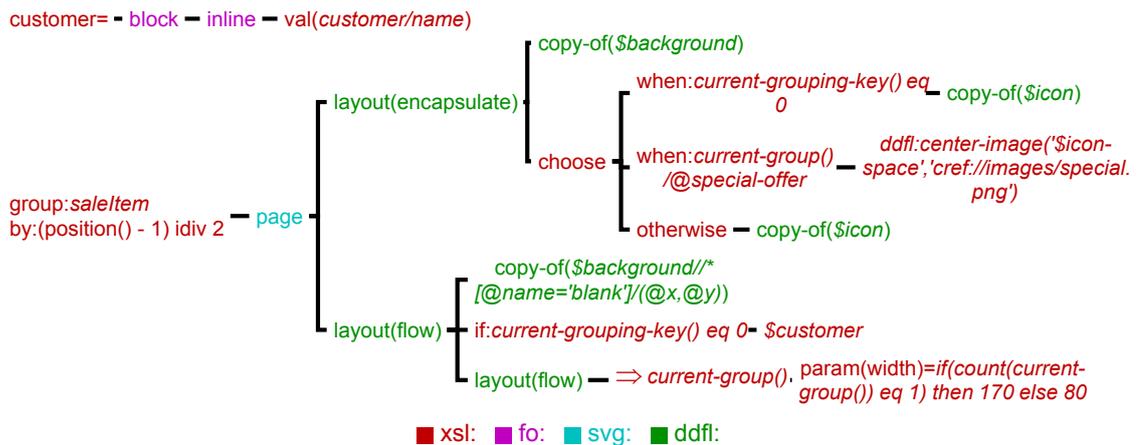
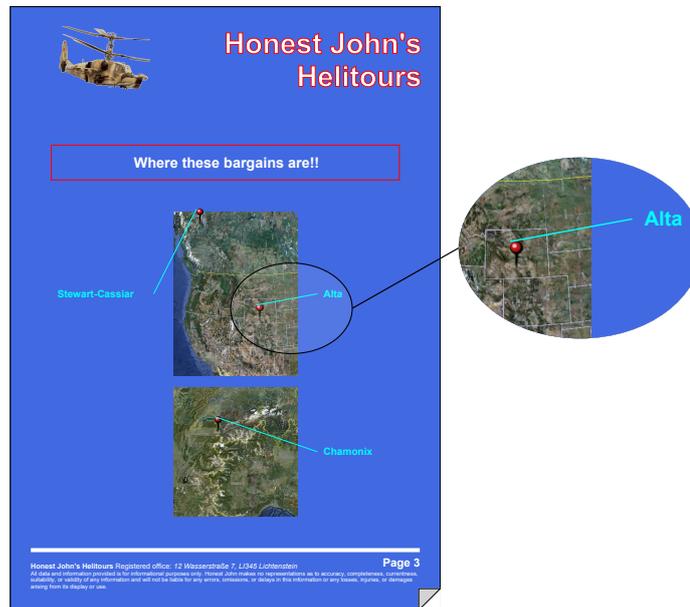


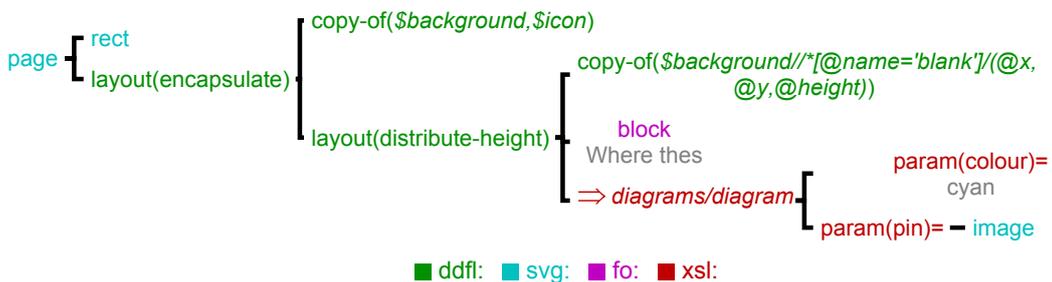
Figure 45. Main brochure pages for salesItems

The **saleItem** pages are generated by an XSLT grouping construct, grouping them in pairs by taking the position of the **saleItem** modulo 2 and generating a new page for each subset. That page encapsulates the background and some icon – either the default or where there is a special offer on an item, the **special.png** image. For the first page (when the grouping ‘key’ is 0), the default icon is always used. Then a flow is placed on top of the *blank* rectangle of the background, by interpolating the x/y co-ordinates within the resolution of the layout. In this flow is a customer greeting for the first page only (using a *pull* operation), followed by a horizontal flow filled with appropriate presentations for the **saleItem** components of this group. These are generated by *push* processing, providing a **width** parameter that varies depend-

ent upon how many items there are. (More properly in this case the width should be propagated during presentational processing – this can be done, but is more cumbersome to illustrate.)



a) Result



b) Definition

Figure 46. Diagram page

The page containing labelled diagrams couples the background and icon with a set of diagrams distributed vertically across the *blank* space using the layout function **distribute-height**. The diagrams are generated through push-processing with two arguments – the colour to use for labels and lines and an image of a pin to use as a marker, in place of the default coloured circle². What the diagrams are is determined by a fairly complex template supplied by the **library-images** document, which sizes and places the image, and for each of the **points** (which have **x** and **y** fractional positions and **label** texts) places a named marker at the calculated point over the image. The **labels** are divided into left and right groups, sorted in vertical order, formed as named text blocks and distributed across the height of the image. Finally connecting lines

are drawn on top by determining placed positions of corresponding markers and labels. This involves considerable use of presentational variables.

When documents are formed of *quanta*, e.g. fixed sized pages or as here, quadruple pages, there is always the issue of managing the shortfall or overflow between the presented material and the presentational space actually available – large areas of unintentional blank space often detract from style. Sometimes with minor shortfall or overflow adjusting whitespace or other stylistic tricks such as changing the size of fonts or images can be adequate. In more extreme cases material must be trimmed or added. For this simple brochure adding filler pages to round-up the document size to a multiple of 4 pages was the design decision. It would be possible to add this as a higher-level layout process (using some marker in an already paginated layout where filler content can be placed) but in this design all the filler pages are inserted at once and then, during the XSLT phase, the number actually required is computed from the material provided in the data, altering the **@visibility** property (**visible | hidden**) of the filler pages to reveal just enough³:

```

n-pages=((count(saleItem) - 1) idiv 2) + 3
n-fillers=if(($n-pages mod 4) eq 0) then 0 else (4 - ($n-pages
mod 4))
∇ company/filler : {
  (xs:integer*) pos=position()
  page {
    @visibility=if($pos le $n-fillers) then 'visible' else 'hidden'
    copy-of($background,$icon)
    ddfi:center-image("$background//*[ @name='blank' ]",.)
  }
}

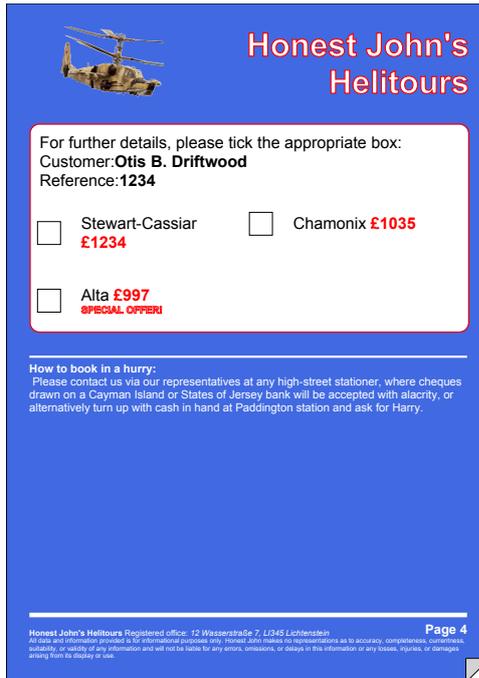
```

■ xsl: ■ svg: ■ ddfi:

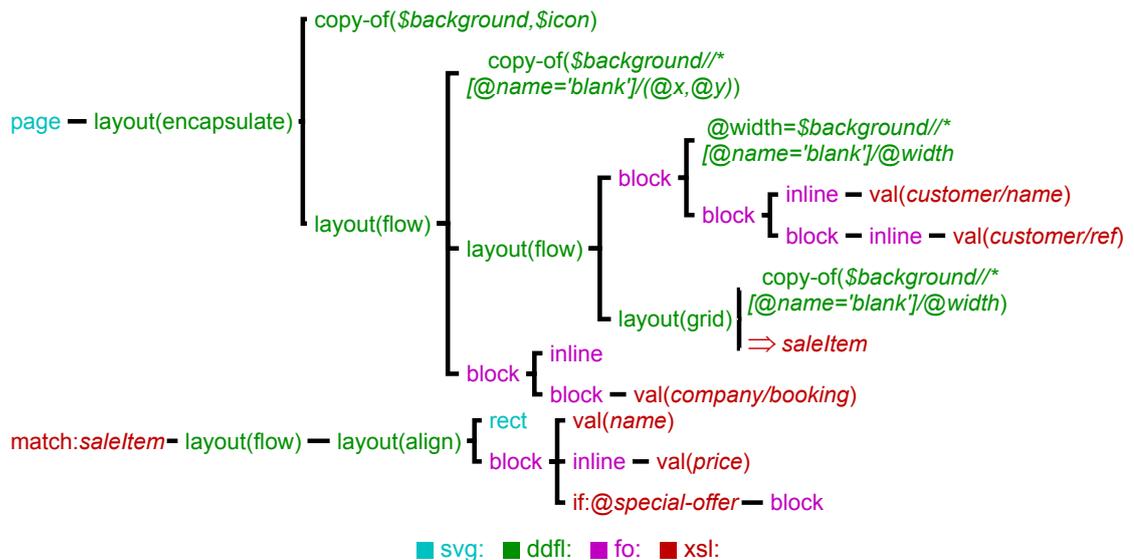
The final page is a rudimentary form for requesting further details. Construction uses some flows and a **grid** layout to arrange each of the **saleItems**, with a simple template in a separate mode providing the presentation for each.

²The choice of a pin is suitable for a map – so perhaps this should have been decided in the structural mapping?

³This approach, exploiting a property of SVG, is suitable for use in partial evaluation, described in section 10.1



a) Result



b) Definition

Figure 47. Customer response page

Providing graphics for a saleItem

The main presentations for the **saleItems** including description, pictures and price (Figure 48) are built from several templates which introduce variation for special offer and multiple-picture cases:

These templates rely on higher priorities for greater specialisation – a common technique within XSLT but perhaps one that will be difficult to employ in directly-edited documents. The templates all work *inpush* mode so, for example, special offers alter the *order* of elements in the flow (**picture** before **description**) and then rely on other templates to provide the actual details. The **name** for a special offer is processed in two XSLT stages – intercepted first by a specific template matching a special-offered parent (**name[../@special-offer]**) which forms a constrained layout around a rotated ‘special-offer’ text block and whatever the **name** would produce *without* the special offer. This is achieved using **xsl:next-match**, which invokes the next-highest priority template matching this node – here the result is interpolated directly, but it could be assigned to a variable and examined.



Figure 48. Presentations for a saleItem

Where there is more than one picture for the **saleItem**, the first picture and first description are combined by placing the picture into an **fo:float** with the text of the description. In this process we pass on a **width** parameter to the picture generator that is a fraction of the current width so the picture will then be sized appropriately. The use of tunnelled variables makes this ‘cascading and modification’ of information comparatively easy⁴.

⁴See section 2.2 on page 37

7.4 Brochure conclusion

The design of this variable document has been discussed in some considerable detail, to illustrate what sort of computational complexity is possible with a framework like DDF. The use of *push mode* processing means that specialist cases can be added cumulatively and increases the possibility of reuse⁵. Having a canonical form for the brochure in terms of **saleItems** means we can contemplate reusing the variable document in another field entirely as can be seen in Figure 49.

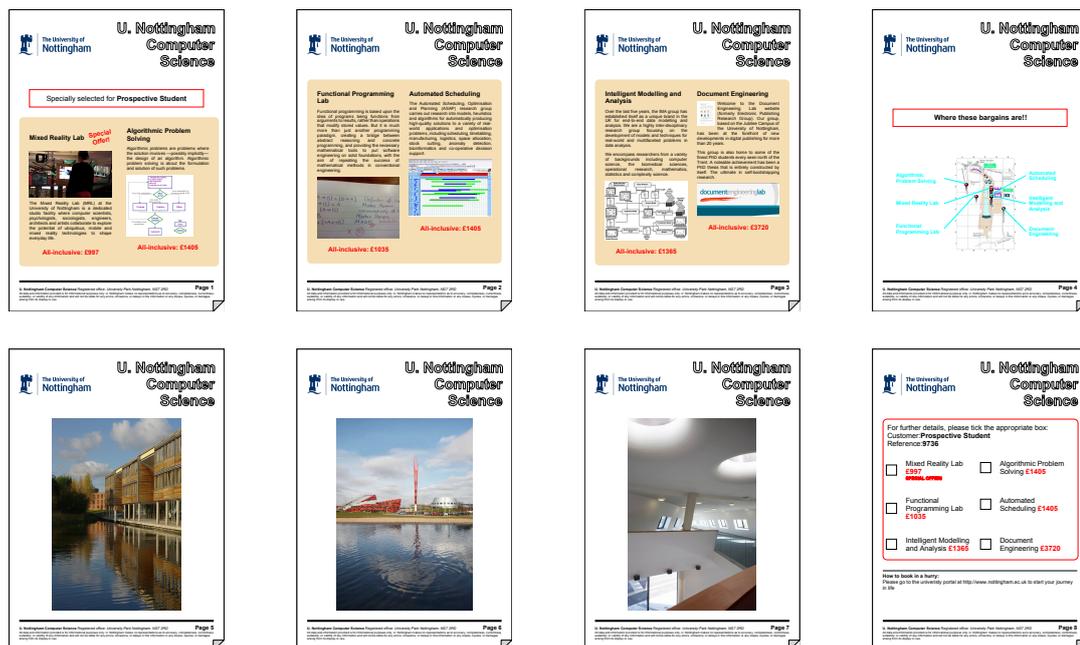


Figure 49. The brochure reused in a different field

In this case some of the styling (background colour) has been optionally defined by some data-dependent element (`((//style/color,'royalblue')[1])`) – similar light-touch styling can be added without altering the layout topology and resulting geometry.

A variable document of such complexity (and considerably more) can be developed by document engineers. But as section 4.3 showed, the extent that documents of this general form can be created and edited through using intuitive and generally familiar graphical tools is a little more restricted.

⁵Albeit at the cost of making debugging trickier

PART C

DOCUMENTS AS FUNCTIONS

Chapter 8

Documents as Functions

Previous chapters have described the prior art of DDF. Part C discusses using DDF documents more extensively as *functions*, leading to suggestions for a novel document architectural model based on findings and ideas described here. This chapter outlines the semantics of the response to data variation through mappings to structural and layout intent, as well as documents treating other documents as passive arguments. The following chapters investigate the functional nature of the extensible document layout model and its implementation (showing extensions to produce ‘higher-order’ capabilities, supporting continually variable behaviour), possibilities of optimisation, variable documents as arguments and an extended example, finishing with a general discussion and conclusion.

Through Part C there will be a developing emphasis on a small number of key findings that make complex functional behaviour a possibility for variable documents. These are:

- Programs and tools should behave as *good XML citizens*, transferring unknown information in their XML source material to approximate tree-isomorphic positions in their results¹.
- Approximate tree isomorphism should be a goal for major evaluation steps if possible – this is especially true in the resolution of layout declaration to final grounded graphical form.

¹Definitions of this term and that in the next item are presented in section 8.1

- Declarative intent for layout can be added as attributive decoration to a grounded graphical result, leading to a degree of *idempotency* in layout. Previous results can be reprocessed and produce the same output, or when partially modified, can lead to successful and correct re-layout without recourse to an independent definition.
- Interspersing of elements in differing namespaces within the overall XML tree supports active modification of a previously bound document, both when simple properties (such as sizes) are altered and when substantial topological changes (elements deleted or added) are made.
- Hybrid actions between differing semantic spaces (especially XSLT and the DDF layout model) can support significant robust and continuous modification of documents.
- The use of a compiler to generate an executable which implements the semantics of the document, enables suitable retention of programmatic intent in variable documents, through partial and multiple bindings.

These new features will require different workflows (e.g. multiple stages of binding – the workflow processor just needs to be supplied with a different configuration) and some alterations to the toolchain (Figure 5) used for the work described in Part B. But the modifications are relatively modest and additions rather than replacements:

- New layout agents added to the layout processor as reactive XSL templates (in mode **ddf:layout**) and a few supporting functions.
- An additional module in the DDF compiler supporting code propagation, which contains a dozen or so reactive templates operating in mode **ddf:modal**, and acting pre-emptively alongside the templates shown in Figure 21.

Much of this will be illustrated on a large scale with the example document of Chapter 12. In the final conclusion the implications of these findings will be discussed, illustrating a possibly cleaner and simpler document form than DDF, based mainly on SVG, XSLT and the declarative combinator layout model of DDF.

8.1 Definitions

There are a few terms that will be used frequently in Part C that warrant specific (if informal) definition, as they are crucial to the extended capabilities described herein.

Higher-order documents

Two types of higher-order document behaviour are explored, roughly corresponding to similar forms encountered in functional programming. The first, and more important for this thesis, is where the binding and evaluation of a variable document *produces another variable document as result*. This relies in part on treating *program as data* within a document and extending the semantics of the document to support such behaviour, which is of a higher order than normal. In the second a variable document takes one or more other variable documents as arguments for use within some subsequent binding².

Approximate tree isomorphism

A tool that transforms one XML to another preserves *approximate tree isomorphism* if the input and output trees can be put into approximate correspondence with each other such that:

1. If some element A on the input tree generates a result element A_o in the output, and a child element of A , B also generated an element B_o in the result, then B_o is either a direct child of A_o , or a direct descendant through ancestors who have no siblings.
2. The document order of elements in the input is preserved in the generated elements in the result, such that relationships along the document order and sibling axes are maintained.

The intention is to retain hierarchical relationships through processes, especially in ancestry. If x in the input has an ancestor y and y projects to y_o in the output, then y_o should have some ancestor x_o which was generated from x . Figure 50 shows two transformations of a tree where the upper case result elements are generated from their corresponding lower case equivalents: b) is approximately isomorphic – c) is not.

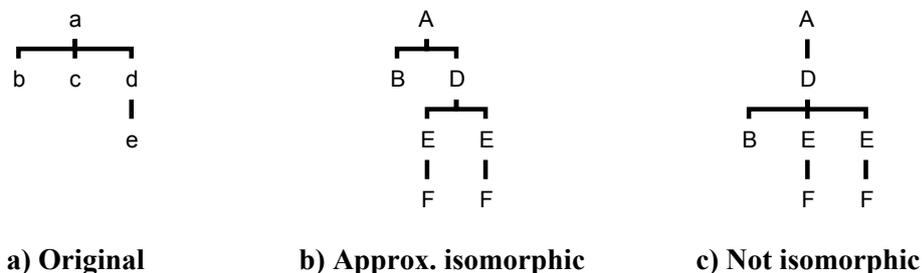


Figure 50. Approximate tree isomorphisms

²Corresponding to the higher-order behaviour of functions such as *map* in FP (Chapter 11).

Good XML citizen

A tool that processes XML structures behaves, to its own semantics, as a *good XML citizen* if it:

1. Does not object to the existence of information from other semantic spaces in the input XML tree. Such information may exist as extra attributes, which may be in no or a foreign namespace, or elements that *are* in foreign namespaces, and
2. When producing an XML result, which is approximately tree-isomorphic to the input, this additional information has been written into equivalent positions (attributes, elements) on the result as that had in the input. In particular attribute \rightarrow element and child \rightarrow parent relationships are maintained for the additional information.

This behaviour permits information to be added to a tree such that hybrid actions involving processes in different semantic spaces can be supported, without having to modify tools to accommodate such extra specific information. Figure 51 shows two transformations of a tree: the tool producing b) was a good citizen; that generating c) was not.

<pre><a> <b n:name="fred"/> <n:c /> <d> <n:e /> </d> </pre>	<pre><A> <B n:name="fred"/> <n:c /> <D> <n:e /> </D> </pre>	<pre><A> <D/> </pre>
a) Original	b) Good	c) Bad

Figure 51. Good XML citizen transformations

8.2 Variable-data functional semantics

DDF was designed from the outset to support high-flexibility automatically-generated variable documents. As Chapters 4 and 6 have shown, the design has two major parts – a cascaded XSLT model to describe variation of ‘presentational intent’ with bound data, and an extensible layout system capable of describing a very wide variety of document graphical forms. This section discusses in greater depth the types of data variation that DDF can handle through several paradigms. This will form a basis for a more advanced ‘functional’ interaction between the data and layout which is explored in Chapter 9 and leads to ‘higher-order’ documents.

The survey of prior art (Chapter 3) has discussed three general models for data-variability for documents: simple queries into database records, used for systems like mail-merge; sets of named variables which are bound through generally simple one-to-one mappings from data, usually on a record-by-record basis, and which can be interpolated by name into the document; and fully-programmatic mechanisms. Clearly DDF has the latter capability, through the use of XSLT. But there are also paradigms that can easily and readily simulate the other forms. Sometimes this can exploit the document logical structure layer to provide canonical representation, but these forms can be simple enough not to need such additional complexity.

A simple ‘database’ query as typified by mail-merge is a suitable starting point:

```
Dear <title> <lastName>,
    Account: <accountNo>
    We have a great offer for you.....
```

Assuming that the data record is a flat XML tree with names similar to the interpolating elements in the document, then the mapping is pretty trivial³:

```
<fo:block >
  <fo:block>Dear <xsl:value-of select="title"/>
    <xsl:value-of select="lastName"/>, </fo:block>
  <fo:block>Account: <xsl:value-of select="accountNo"/>
</fo:block>
  <fo:block>We have a great offer for you....</fo:block>
</fo:block>
```

The semantics of the **xsl:value-of** will yield an empty string in the absence of a value, so the default behaviour is as would be desired. If an editing system similar to [63] is used, then the **xsl:value-of select="var"** elements could be projected through as **<fo:inline><var>..** with suitable trace pointers, so the authoring document can use simpler reduced variable indicators. Similarly an editor can support the insertion of a variable value through user interface interaction that adds the appropriate XSLT structure to the source and which then gets reflected in the visual form⁴.

When the data is not arranged so conveniently, then some simple data mapper might suffice. A common example is the use of the comma-separated variable (**.csv**) format. In this case we can either arrange that the structural layer is used as a buffer, creating an XML structure with

³**fo:block** structures within an **fo:block** denote sub-paragraphs or line breaks.

⁴The editor described could easily show variables with differing appearance (colour, font, surround) without disturbing the inherent style of the eventual text.

suitable names, or the interpolating **xsl:value-of** elements will use indexing statements (**tokenize(.,',')[3]**). A better solution is to use a model-based representation for the named field mapping, such as shown in Figure 52:

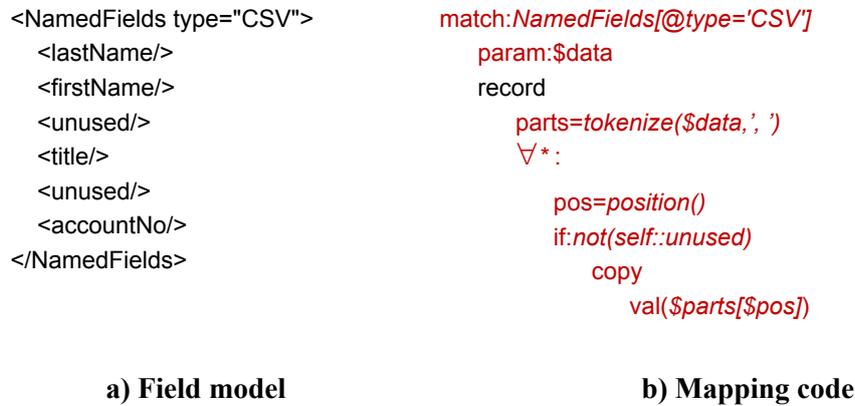


Figure 52. Named field mapping

Obviously this model could either be interpreted at each document instance or compiled into the document by partial evaluation (Figure 53):

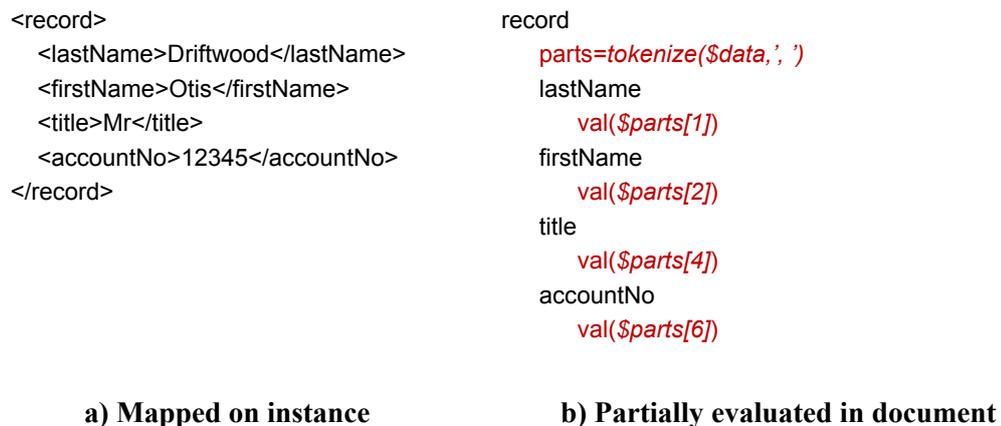


Figure 53. Model projected on data or document

Supporting an extensible set of named variables (e.g. as used in *Dialogue*[103]) has two possibilities, depending upon whether the document can be compiled against a model or not. There obviously has to be some list of the variables, their types and their source mapping. In our small example this could be something like:

```
firstName: type=string source=CSV[2]
lastName: type=string source=CSV[1]
title:    type=enumeration("Mr.|Mrs.|Miss|Ms|Dr.|Prof.") source=CSV[4]
accountNo: type=database("accounts")
           query=(last=lastName,first=firstName; => number)
```

With a reasonably small set of such variable types, user interfaces for creating and modifying them are possible. Given such a model we could represent variable interpolations within the document itself either as reserved words such as **\$title** or preferably XSLT structures that link to the variable model and give a suitable visual appearance as discussed above. A preferred option compiles the model into the document, making the variables first-class **xsl:variables** and then interpolating them in the normal manner. This has several advantages:

- Variables have suitable scoping – local variables can be supported within a subtree or subsection, while global ones can still operate.
- Types can be identified and checked both at compile-time and run-time.
- Expressions of several variables can be supported easily, including variables created as functions of other variables.

For our example the following series of variables might be generated:

```
<xsl:for-each select="$lines">
  <xsl:variable name="firstName" as="xs:string" select="f.csv(.,1)"/>
  <xsl:variable name="lastName" as="xs:string" select="f.csv(.,2)"/>
  <xsl:variable name="title.Enum" as="xs:string*" select="'Mr.', 'Mrs.', 'Miss', 'Ms', 'Dr.', 'Prof.'"/>
  <xsl:variable name="title" as="xs:string" select="$title.Enum[number(f.csv(.,4))]/>
  <xsl:variable name="accountNo" as="xs:string" select="f.database('accounts', ('last', $lastName, 'first', $firstName))/number"/> ....
</xsl:for-each>
```

8.3 Documents as passive arguments

There is an interesting subclass of variable documents that treats other documents as data. Chapter 11 discusses where these argument documents are considered to be *active*, i.e. some of their variability is subsumed into the document that consumes them, but here they are treated as *passive* data sources. In essence they are only more complex forms of data, and assuming they are represented in XML this poses few problems, other than perhaps those of scale⁵.

A simple case is generating an external outline or table of contents for a given document. There has, of course, to be some knowledge of what type of construct is used to describe a heading or section and where the useful data is located in and around that construct. If the document has a logical structure written in XHTML, then **//section/h** will get all the headings,

⁵Consuming PDF is an entirely different order of complexity.

and a template-based tree descent which copies only **section | h** and their descendants will form up a correct nesting without full content. If however result page numbers are needed, then the final paginated presentation needs to be accessed. We need to know how a heading appears in the result for this type of document (assuming it can somehow be distinguished from other text) and how one extracts both the heading and the ordinal position of the page element within which it sits.

Cases such as a précis involve some of the content (figures, paragraphs) of one or more documents being extracted and generated into another document. Here there is an additional issue: collecting any required resources or properties to complete a correct presentation of the extracted portion *as it appears in the source document*, as opposed to where only the information is used, as in making an outline. The issues are similar to those investigated by Bagley[8], having to collect relevant style state that appears elsewhere in the document to make the extracted components be self-contained, though complete tracing of external resources, such as images, may involve some complexity.

Of interest to the computer scientist is using this approach for documentation of a set of programs, or indeed variable documents. Here having *everything* described in XML is more than a distinct advantage – primary parsing is automatic and XPath makes collection declarative. Here are a few examples of how simple this can be:

- An index of all the XSLT functions can be formed by using **//xsl:function** to select each of the function definitions. A matching template can then collect its name (**@name**), any type (**@as**) and all the parameters (**xsl:param/(@name,@type)**) to build up a set of result structures that can then be sorted and/or grouped and used as presentation. (XHTML is good as it makes building Web-documentation easy.) Similar methods can be used for **xsl:template**.
- Library resources imported by documents or programs from other files can be tracked by following the inclusion statements (**xsl:import href="file"**) recursively opening the document pointed at. Relative source locations can be resolved and loop recursions can be avoided easily. Building an index of ‘in which file is this resource defined’ is trivial.

- All function calls are contained in XPath expressions (**pic:pic-width(..)**), so unless such patterns appear in embedded strings, regular expressions (**\\ilc*:\\ilc*\\()**) can find these references and make call-maps for sections of code – automatic generation of hyper-links around the map is straightforward.

These all look remarkably like documentation programs, but the point is that they can be contained in the variable documents: binding and evaluating the documenting document over the source of interest produces the desired documentation:

```
<xsl:function name="f:parse-svg-path" as="element()*"
  doc:doc="Parse an SVG path definition into its instructions. The result is a sequence of
  elements, the name of which is the instruction (e.g. L, m, z, C), and the text value is the arguments
  for that instruction.">
  <xsl:param name="path" as="xs:string"
    doc:doc="Typically the @d attribute from an svg:path"/>
  <xsl:for-each select="tokenize(replace(normalize-space($p),'(.)([A-Za-z])','$1|$2'),'\\|')">
    <xsl:element name="{substring(.,1,1)}">
      <xsl:value-of select="normalize-space(replace(substring(.,2),',''))"/>
    </xsl:element>
  </xsl:for-each>
</xsl:function>
```

<i>element()*</i>	f:parse-svg-path (<i>path as xs:string</i>)	Parse an SVG path definition into its instructions. The result is a sequence of elements, the name of which is the instruction (e.g. L, m, z, C), and the text value is the arguments for that instruction.
	(<i>xs:string</i>) path	Typically the @d attribute from an svg:path

Figure 54. Simple auto-documentation

Very modest documentation paradigms can make this more valuable. Using a reserved namespace for documentation attributes and elements (**doc:**) encourages in-file documentation almost in a literate programming style[97], without needing the ‘tangle’ operation – the step described here is effectively the ‘weave’. Any XSLT element could be decorated with a **@doc:doc** attribute with some simple textual description⁶ – this can be used for argument descriptions (**xsl:param/@doc:doc**). Top level **doc:doc** elements could contain extensive (HTML) structured documentation sections; with a protocol where they immediately precede the entity they describe, retrieval is through a selector **preceding-sibling::*[1][self::doc:doc]**.

⁶XSLT ignores foreign namespace attributes and top-level elements – this is crucial for some of the higher-order effects described in Chapter 9.

Chapter 9

Variable Layout as a Higher-Order Function

A consistent model for evaluating a document layout that still contains variable elements is examined in this chapter, providing a foundation for continually active, or partially bound, variable documents. We show how layouts can be declared in ways where they can be idempotent under layout resolution, re-satisfied when information within them changes, and contain elements that support controllable *topological* modification, i.e. subcomponents being added, deleted or modified as a document is progressively bound to a sequence of data.

The previous chapters have shown a document acting as a simple function of its variable data, generating a ‘fixed’ value for a specific data instance. This mapping can be quite complex, especially in layout, which is designed for smooth extensibility. But functions can exhibit further complexity, as manifested in functional programming, including two behaviours characteristic of the evaluation of *higher-order* functions:

- They can produce a function as output.
- They can accept and process one or more functions as input.

This chapter examines the possibilities of the first of these: a variable DDF document being evaluated over some data and resulting in another variable DDF document, which may have viewable presentation. This opens the possibility of creating ‘continual documents’ that modi-

fy themselves under strict document-declared control as data is bound to them progressively, whilst still providing viewable and meaningful presentations at each stage. Such behaviour would be almost impossible to implement in conventional document systems. Documents that take other documents as input are discussed in Chapter 11.

The key is that by arranging that document variability is expressed as a program embedded within the document and held in the same XML syntax as the rest of the document, for some purposes the *program can be treated as data*. This means that program can be altered to accomodate extended semantics (such as continual activity through multiple bindings) and then embedded in the resulting document. Hence the document (or document and processor) are behaving in a fashion that is of higher-order than that of a simple variable document.

Adding three concepts to the layout model (*attributive* layout directives, *foreign* elements and their correct handling, and *meta-layout* operators) and defining semantics for retaining fragments of XSLT within presentational layout trees, is sufficient to support this ‘variable document as output’ functionality. Small scale examples will appear here; Chapter 12 will illustrate application-scale use and will explore some design paradigms for such a document. This is the key set of ideas of this thesis.

9.1 Layout and approximate tree-isomorphism

The model for layout required by a document described so far is strictly the invocation of a deeply nested function call, with the addition of XSLT-like semantics for single-assignment variables, iteration and choice. Ignoring the additional semantics, a call is of the form:

$$f_i(f_j(x_a, x_b, \dots), f_k(x_c, \dots), \dots)$$

where f_i are the layout functions, each with properties, and x_a are leaf primitives. (In this case SVG elements that contain no further embedded layout functions – SVG elements that *do* contain such functions can be considered to be identity copies.) There are two important points:

- Being considered a nested invocation of *functions*, free of side-effects and which produce unique single ‘values’ for the same arguments, regardless of external context, means that operations such as partial evaluation are possible.

- If the functions construct an output tree that reflects their own calling structure, i.e. producing a result $g_i(g_j(x_a, x_b, \dots), g_k(x_c, \dots), \dots)$ where the brackets denote tree structure and g_i is an SVG node with properties that are canonical for the result (i.e. a rectangular position/extent) and contain any additional information left over from the f_i invocation (i.e. unknown properties), then higher level functional operations can operate on *arbitrary* constructs below.

Retaining the parent-child relationships of the functional declaration into the result has proved critical. Whilst all the children of some layout could be flattened, the **svg** and **g** constructs of SVG can mark result groupings¹, and they constitute a node on which to attach *group properties*. Such properties can thus be exploited by ancestor layouts – the most powerful example shown so far is the convention of using **@name** to identify pieces, making it possible to describe relationships between named subcomponents and to extract post-layout properties (usually position) for such. Such isomorphism also allows us to support a degree of *idempotency* in variable document layout as the next sections will show.

9.2 Layout with embedded function

Generation of *grounded* layouts from declarations of intent has already been discussed, where the layouts were either built from document variability or which were statically invariant in the original document. Documents of interest might also need further variability contained in their presentational output, for later binding of data or generation of a more specialist document. Examples include:

- Specialising a generic template by binding partial information. For example a generic set of ‘invoice’ templates could be bound to data for a particular company, generating a set of appropriate document templates for use in their variable document workflows. Some parts of this (e.g. company name and address fields) can be laid out completely. Others, such as customer name, still require that interpolative statements be left in the layout.
- A document that has ‘continual life’ – where data is added by stages that could be widely separated in time, but the document is still ‘presentable’ at any point. A medical record is an example, which will feature later in the thesis.

¹They map easily into PDF for example.

These new possibilities raise, in turn, three separate issues. Firstly new interpolative statements (**xsl** elements) may be generated or left within (**svg**) layout results; secondly, layout *consequences* must be evaluated correctly around such elements and thirdly, layout *intents* must be retained on results to support subsequent re-satisfaction.

9.3 Attributive layout and embedded program

Chapter 6 described required layouts as reserved tree nodes with children beneath, acting as a canonical form for any localised layout. But some types of layout are essentially localised geometric transformations of a single piece, and could be considered additional *properties* of a node. An example is rotation, which could be declared in two ways:

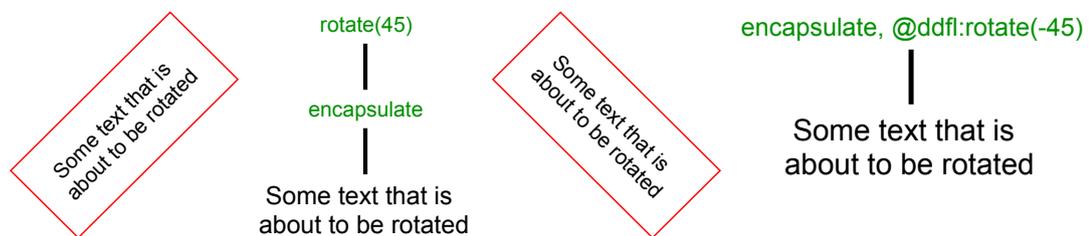


Figure 55. Simple rotational layouts & source declaration

In the second form we attach the attribute **@ddfl:rotate** to the ‘encapsulating’ node. Giving such declarations higher precedence than the node itself, (caught by ***[@ddfl:rotate]**) causes the result to be *post-processed* by a layout agent – in this case to rotate the result. This is a more natural declaration for a single piece: it can be considered an extended property of the element, and can avoid the need to make the tree another node deeper. Multiple use of this approach (e.g. for actions such as scale, mirror, rotate and anchor) requires a priority order to be established for non-commutative actions.

Using this approach for more general declarations of layout intent opens interesting possibilities, especially for describing and evaluating partial and multi-stage layouts. A flow would usually be defined as in Figure 56a:

<pre> <ddfl:layout function="flow"> <svg:rect fill="blue" width="8" height="8"/> <ddfl:layout function="flow" direction="x"> <svg:rect fill="red" width="12" height="7"/> <svg:rect fill="green" width="15" height="12"/> </ddfl:layout> </ddfl:layout> </pre>	<pre> <svg:svg ddfi:layout="flow"> <svg:rect fill="blue" height="8" width="8"/> <svg:svg ddfi:layout="flow" direction="x"> <svg:rect fill="red" height="7" width="12"/> <svg:rect fill="green" height="12" width="15"/> </svg:svg> </svg:svg> </pre>
--	--

a) Element combinators

b) Attribute declaration

Figure 56. Element & attribute defined flow layout

But if the layout processor recognized an attribute `@ddfi:layout="flow"` as a synonym (Figure 56b) then we could attach that to `svg:svg` elements yielding a layout *result*:

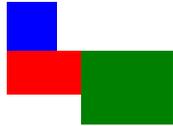
	<pre> <svg:svg ddfi:layout="flow" height="20" width="27" x="0" y="0"> <svg:rect fill="blue" height="8" width="8" x="0" y="0"/> <svg:svg ddfi:layout="flow" direction="x" height="12" width="27" x="0" y="8"> <svg:rect fill="red" height="7" width="12" x="0" y="0"/> <svg:rect fill="green" height="12" width="15" x="12" y="0"/> </svg:svg> </svg:svg> </pre>
---	---

Figure 57. Simple attributive flow layout – resulting graphics

Why is this important? To put it simply, the resulting SVG tree still contains *all the information necessary to re-evaluate the flow if something inside changes*. If we took the result of Figure 57 and passed it through the layout processor again, we would get exactly the same result. If somehow we alter the size of one of the rectangles within that result or add something new inside the flow, then we can reprocess the presentable SVG for layout to get the correct new arrangement.

KEY: Layout intent as attributes. Layout intent can be added as attributive decoration to a grounded SVG graphical result, leading to *idempotency* in layout. Previous or modified results can be reprocessed yielding the intended output without recourse to an independent definition.

Suppose that somehow a ‘foreign’ object which happens to be a section of XSLT had been inserted in the middle of the inner flow and had not been evaluated:

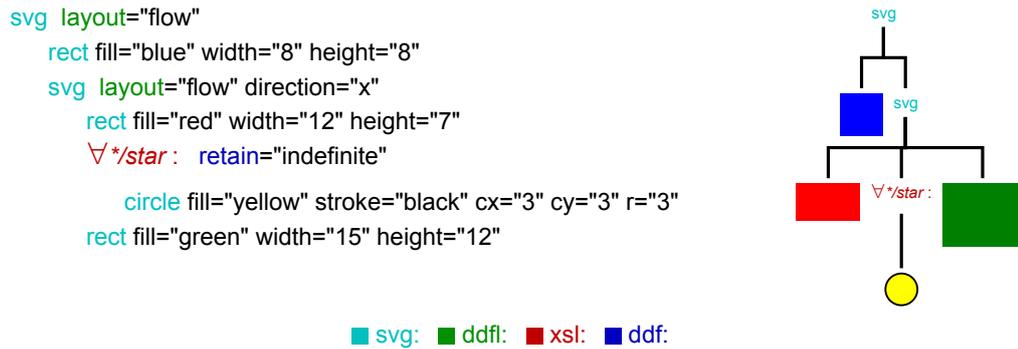


Figure 58. Attribute defined flow layout

If this flow is evaluated by the layout processing agent for flows, and provided that agent is a good XML citizen, i.e. not only does it not signal an error at the included XSLT structure but deep-copies it, unmodified, into the same place in document order in the result and ignores it when calculating positions for all ‘proper’ **svg:*** children, the result will be:

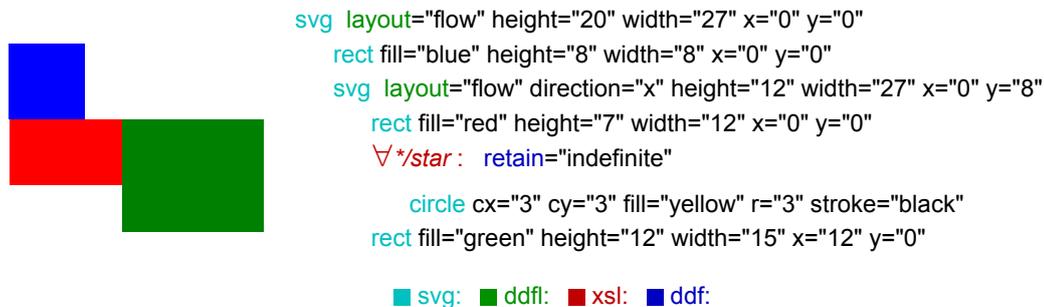


Figure 59. Simple attributive flow layout – constant result

The flow has been evaluated for layout as if the XSLT wasn't making any difference, but the placeholder still remains. Furthermore the result still contains flow declaration information. If the final SVG imager (renderer or PDF generator) ignores the XSLT completely then the result is correctly presentable. And the resulting structure is nearly identical to the input. Further re-evaluation as layout will continue to produce the same result, hence *idempotency*.

KEYS: Good XML citizens and tree-isomorphism. Non-layout elements and declarations can be accommodated and retained within layout trees, preserving position within the tree, provided layout is tree-isomorphic and layout agents behave as *good XML citizens*.

If subsequently this SVG tree is evaluated on XSLT semantics, the placeholder may generate new content, which on layout will ‘expand the flow’:

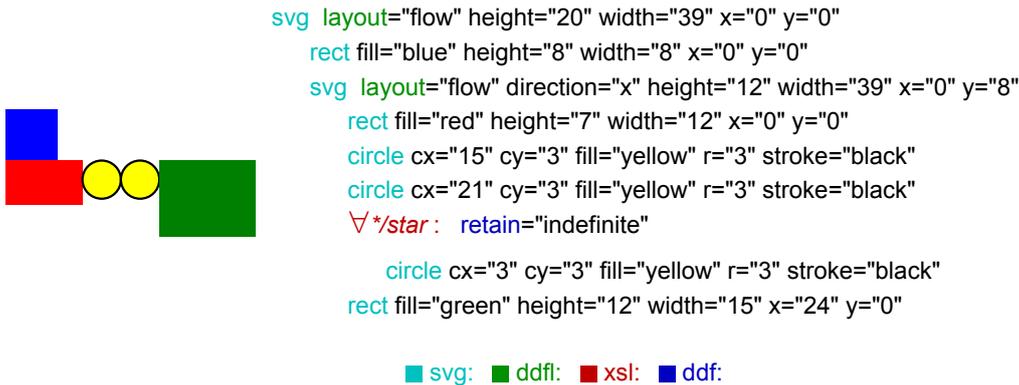


Figure 60. Simple attributive flow layout – data-modified graphics

There were evidently two ‘stars’ in the data, which the XSLT fragment iterated over to produce the two circles. The generating XSLT fragment remains – the directive `@ddf:retain="indefinite"` caused a self-propagating construct to be added. How this is done is described later, but it can be coherent and is crucial to producing continually active documents.

KEY: XSLT defines variability. As the data-processing ‘program’ is defined completely in an XML syntax, it can be embedded in other XML structures and manipulated by XSLT in a similar way, even to the point of generating self-propagating code.

Figure 61 shows diagrammatically what is going on in a different simple layout with two places interpolating data – the first conditionally into a text block, and the second as the colour of a rectangle, both being within flows.



Figure 61. Layout with two sections of variability

Binding and laying-out this out, to two distinct data records in progression, yields Figure 62. A tree representation (Figure 63) shows what happens most clearly. The nodes are coloured to represent the namespaces of elements involved, with two special forms for `svg:svg` ele-

ments: those that are expansions of **fo:block** elements have an **fo:** coloured border, and those that are decorated with **@ddfl:layout** properties are similarly bordered with the **ddfl:** colour and presented as a circle.

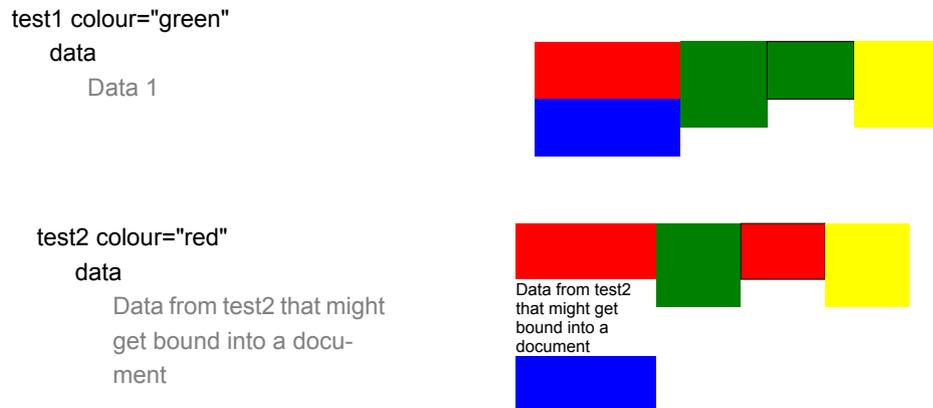
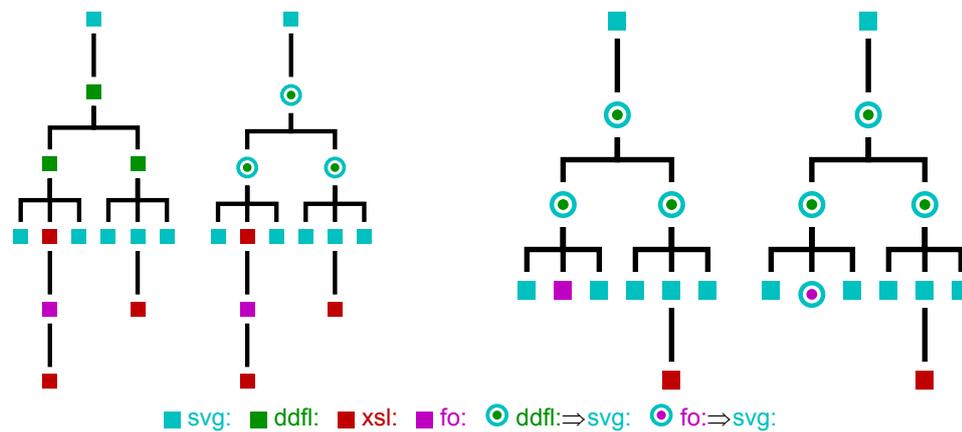


Figure 62. Two different data bindings for Figure 61 & resulting graphics



a) After XSLT(test1) & layout

b) After XSLT(test2) & layout

Figure 63. XSLT & layout evaluations for binding test1 followed by test2

Figure 63a shows the layout after binding the XSLT to the *first* data instance and resolving its layout; the result has then been further evaluated over the second instance in Figure 63b. The **fo:block** is unexpanded during the first pass as the condition (**xsl:if test="test2"**) is unsatisfied at this point and the construct is marked to be retained – in this case the subtree passes through unchanged, without disturbing the flow. The **xsl:attribute** that sets the rectangle colour has triggered but, again, is marked to be retained. On the second binding the conditional **fo:block** can trigger and interpolate its value, resulting in an expanded text block, that i) does influence the flow within which it lies and ii) is now no longer variable – the parent and child XSLT (**xsl:if**, **xsl:value-of**) have been interpolated and removed.

Later sections will show how adding a hybrid system of XSLT and specialist *meta-layout* declarations can support topological modification of graphical presentations. But this is only possible due to these **key** ideas, which will be examined in more detail:

- SVG trees can be decorated with attributive declarations to describe intended relationships between subparts of the trees.
- Elements and attributes in foreign namespaces can exist within SVG trees and support variable graphical behaviour *provided* that all tools act as *good XML citizens*.
- XSLT program fragments, being entirely described in XML, can exist within, and be manipulated along with, other XML structures.

9.4 One-to-many mappings

We have seen how a layout that is a one-to-one mapping in tree terms could be declared purely with an attribute – the input tree produces a single tree as output which is topologically similar to the input. A tree node with n children produces an output tree, also with n children, probably in the same document order.

For constructs like flows this is adequate, but not all layouts are one-to-one in form. A paginator was described in Chapter 6, which is a one-to-many mapping. The defining layout tree has a node declaring pagination with parameters² and a set of n children that should be packed in sequence into containers. The result of this is not a single tree but a sequence of trees each filled with the relevant children. In graphical terms we have:

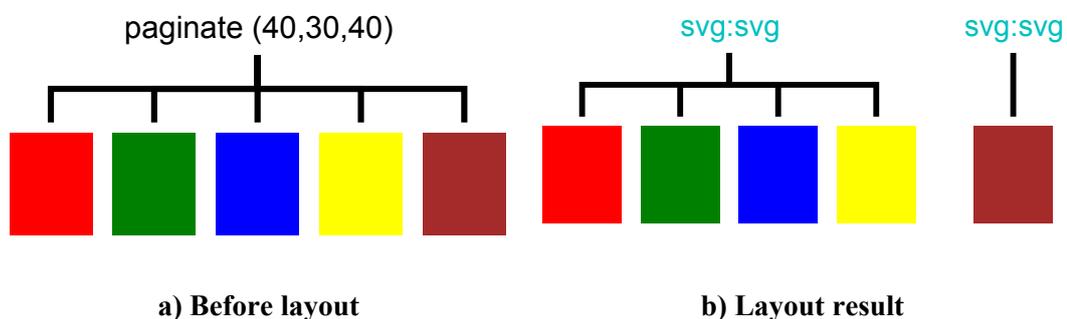


Figure 64. One-to-many pagination layout

²This argument assumes that the containers are described as parameters not children – when they are ‘reserved children’ a modified approach applies, described in the next section.

Now if we wish to declare the ‘pagination’ constraints across the result, where should they be placed and how should the collection be gathered for resatisfaction? One possibility, in the spirit of ‘parental responsibility’, would be for parent layouts to check for one-to-many result groupings in their children, before evaluation. However this requires every layout to do so. Another possibility exploits the common paradigm within layout agents that they evaluate all children in a uniform manner by the action **apply-templates select="" mode="ddfl:layout"** and then operate on the resulting laid-out children.

We assume that all the results from a one-to-many mapping are consecutive siblings of each other in the resulting tree (thus preserving approximate tree isomorphism). We arrange for just *one* of the siblings to be responsible for gathering all the components from itself and its siblings and ensuring that all the other siblings return a null result. A suitable test is **empty(preceding-sibling::*[@ddfl:layout=\$function])** where **\$function** is the name of the layout – this will be true for the first element of the mapping result, and false for all the rest.

The test can be refined further by identifying members of each one-to-many mapping with some marker, such as **@ddfl:as="element()"** and giving each group a unique **@ddfl:id**, so that several similar mappings can act independently, even when results are contiguous siblings. Figure 65 shows a suitable meta-layout agent for collecting and evaluating these groupings.

```

match:svg:svg[@ddfl:layout][@ddfl:as='element()*] mode="ddfl:layout" priority="2.5"
  function=@ddfl:layout
  id=@ddfl:id
  if:empty(preceding-sibling::svg:svg[@ddfl:layout=$function][@ddfl:id=$id])
    (element()) temp=
      layout({@ddfl:layout})
        copy-of(@* except (@ddfl:layout,@width,@height),*|text())
        copy-of(following-sibling::svg:svg[@ddfl:layout=$function][@ddfl:id=$id]/(*|text()) except *
          [@ddfl:artifact])
    => $temp mode="#current"
  param(attributes)=@ddfl:layout,@ddfl:as,@ddfl:id,@spacing|@direction

```

Figure 65. One-to-many layout resatisfaction code

Figure 66 shows the source of two independent paginations in sequence, both of which contain additional element generators. In Figure 67 the addition of circles into the set of children forces later, ‘older’, siblings to move into the next container and in one case forces creation of an extra page.

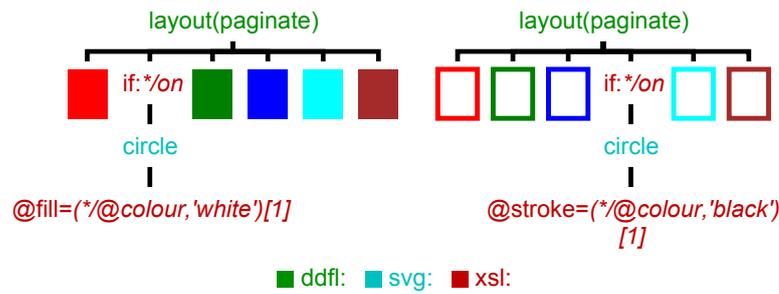


Figure 66. One-to-many pagination with continuous alteration.

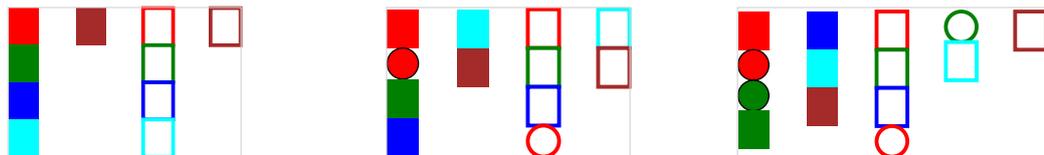


Figure 67. Three successive bindings to the paginations of Figure 66

9.5 Foreign namespaces within layout

The previous examples are simple cases where a layout can be resolved when some of the child elements are *not* graphical or layout constructs. To do this consistently and correctly, and for layouts more complex than a simple flow, certain protocols must be followed. The aim is to describe generic methods that absolutely minimise the changes needed in layout agents and maximise the cases in which resatisfaction can occur correctly.

Once again it is imperative that tools behave as *good XML citizens*. This means that:

- The presence of unknown attributes on an element (regardless of namespace) or child elements in *foreign namespaces* does not cause an error³.
- Unless there are specific reasons not to, transformational tools generating XML results from such input, which have approximate XML isomorphism in their mapping, should copy over such unknown information (attributes and elements) to equivalent positions in their results. Whether such copies are deep or shallow may depend upon semantics. Information that may be used at ‘higher level’ (acyclic interdependencies, editing) is thus retained. In this chapter it supports extended functionality.

In the DDF framework two specific areas *must* obey these strictures. Layout agents must do this as a matter of course and how they do this consistently is the bulk of this section. But

³Systems with strict schema awareness will come unstuck here – it is my opinion that schemas are best employed for input data verification, rather than deep runtime analysis

any *visual observer* of a DDF document is similarly bound – the ‘no error’ condition is relatively simple⁴, but foreign elements must be **ignored** in something like a PDF generator. Usually such ignorance should be deep: it is difficult to envisage a situation in an observer where it should be shallow (i.e. ignore the element but descend into the children), though a suitable case might occur during layout invariant optimisation.

The process of handling foreign attributes during layout is fairly straightforward. Most agents when constructing the output element, copy across all attributes appearing on their input element *except* the ones that define their known parameters. (Resatisfaction now makes it important to retain even those.) But handling foreign elements is not quite so simple.

As we have seen, an XSLT element should be considered to have no geometric extent but still be copied into place in something like a flow. But not all layouts are that simple and the effect of the XSLT element may not be simply confined to its geometric influence. Figure 28 in section 6.1 described a square arrangement, which counted the number of children to determine an appropriate number of columns and rows. Buried XSLT subtrees within that set will make that count too high – such elements should be excluded from the set of ‘proper children’.

Similarly any activity that effectively iterates across part or all of the child set needs to be ‘foreign aware’. Some statements like **sum(preceding-sibling::*/@height)** or **min(*/@x)** may work correctly assuming foreign elements do not have such properties, but **if(position() mod 2 = 0) then 'lightblue' else 'white'** will err on producing alternating background colours with foreign elements present. In effect some of the common XSLT and XPath constructs that will be present within *layout agents* will need replacement. Figure 68 shows a few examples and some supporting canonicalising functions.

ddfl:foreign() is a predicate function identifying ‘foreign’ elements – elements of SVG that are decorated with the property **@ddfl:foreign** are considered to be in this class, regardless of any other semantics. **ddfl:position()** and **ddfl:last()** provide ‘foreign safe’ equivalents to **position()** and **last()**, though they do have to have a node argument.

Some sections of result SVG are part of the *group* appearance – examples include background fills and partial borders. Sometimes these can be quite extensive, as in ‘folder’ encapsulation. Whilst these elements are held within the parent **svg:svg** in the result (they must be to cor-

⁴Using the Batik SVG renderer would cause a problem – unknown attributes cause error termination.

rectly translate or scale with the whole group as well as display), they are not true children of the original layout. They must be identified as such (**@ddfl:artifact** is used) and are excluded from any re-evaluation of the layout – indeed they will be deleted and rebuilt if demanded. The **ddfl:foreign()** function detects such elements.

∇*:
BODY

∇*:
choose
 when:f:foreign(.)
 •
 otherwise
 BODY

a) Construct.

b) Foreign aware equivalent.

```
function name="ddfl:foreign" (xs:boolean)
  (element()) param:node
  exists($node/self::xsl:*) or exists($node/self::fo:*) or $node/@ddfl:artifact or
  $node/@visibility='hidden' or (exists($node/@ddfl:foreign) and boolean
  ($node/@ddfl:foreign))
function name="ddfl:position"
  param:node
  count($node/preceding-sibling::*[not(ddfl:foreign())]) + 1
function name="ddfl:last"
  param:node
  count($node/parent::*[not(ddfl:foreign())]) + 1
```

c) Support functions.

Figure 68. Foreign element aware constructs

Simple one-to-many mappings can be supported by the layout agents marking the result children appropriately and then using generic ‘collect all my siblings’ actions. But for some layouts more is required, such as practical paginators which have reserved children describing the containers in which to flow or templates that include background content and flow containers (e.g. **ddfl:container**, **svg:rect[@ddfl:container]**, **ddfl:template[*[@ddfl:container]]**). For these to be resatisfiable, their information must be transferred to the result, often as element structures. This requires the layout agent to follow additional protocols.

These reserved definition children should only be copied *once* into the result set, and described as foreign (**@ddfl:foreign**) – this is most easily accomplished by copying in only with the first of the result children (**if(position() = 1) then \$reserved else ()**). If the true result children are encapsulated with other background material from one of these definitions (as would

be the case with a template) then the situation is somewhat more complex but there is a suitable protocol involving labelling such material as artifactual, ensuring removal and possible regeneration at a subsequent evaluation (Figure 69).

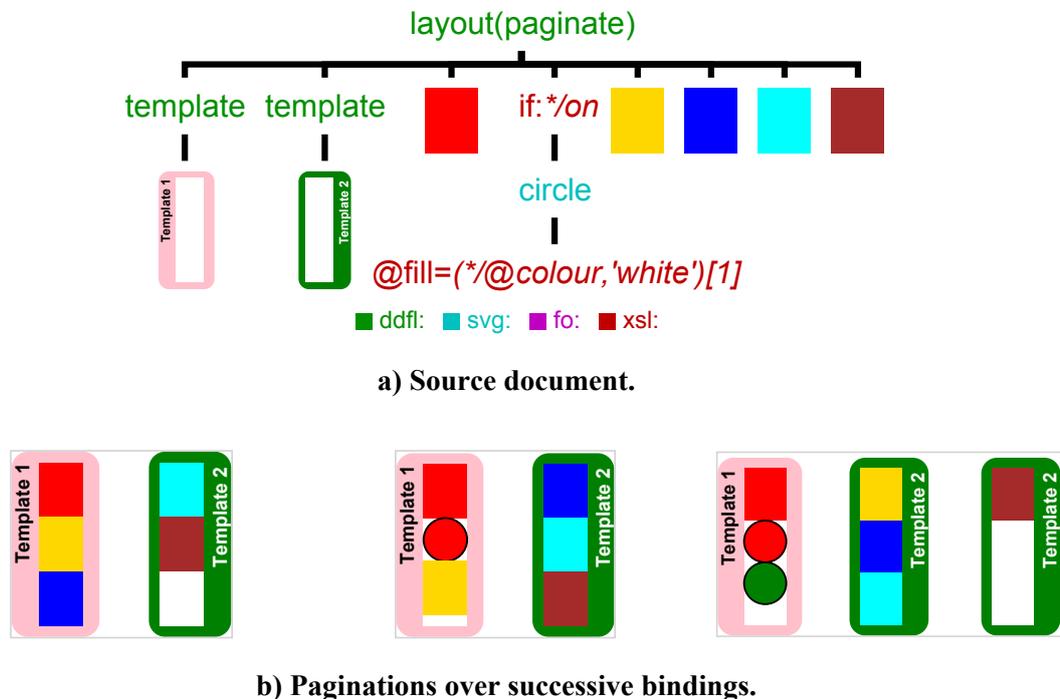


Figure 69. Resatisfying pagination with background templates

All these examples actually exploit one significant factor – the re-processing of the children themselves is independent of the parents. This is usually true, but one important case does not satisfy that condition – text flow within containers with varied widths. As we show in Appendix B the technique here is to use a presentational variable of fixed name (**text-column-width**) set by the paginating layout agent to the width of the current container, and to reference that width in an **fo:block** before expansion – to resatisfy such systems we will need to retain and then resatisfy the defining block.

For several reasons preclude supporting continued embedded program statements within the layout of, or re-satisfaction of layout over, **fo:block** text paragraphs: i) the line-wrap code is sufficiently complex already, ii) features such as hyphenation add to the complexity of resatisfaction and iii) paragraphs are often evaluated within a parent-derived context (e.g. container width in paginated flows). The approach taken is to replace such blocks *in toto* using the mechanisms described later, rather than partial layout.

This method allows SVG trees to be processed and re-processed to satisfy attached layout declar-

ations – to exploit this for practical extended functional purposes we need a mechanism that permits alteration to these trees, preferably through embedded fragments of additional code.

9.6 Modification of the SVG tree

The example above has shown *addition* to the SVG tree by dint of the execution of a fragment of XSLT embedded within the tree, leaving extra SVG material that may influence the overall layout. (Whether some XSLT remains within the tree after execution is immaterial here, but becomes important in the life of a continual document.)

But information within the tree also needs *deletion* and *replacement*. One of the critical properties of XSLT, in common with all functional programming, is that there is *no* ‘delete’ operation – all actions either leave the result tree unchanged or add to it at their point of operation. Removal of parts of a tree *by XSLT* can only be accomplished by *not copying* those sections to a entirely new tree.

Updating an attribute property (but not removing the attribute itself) can exploit the ability of **xsl:attribute** to overwrite an attribute on the result tree parent, as in the previous example where the colour of a rectangle was updated from the bound data. A general solution for modifying or deleting elements however requires some external agent, operating at a stage separated from the XSLT execution. One possibility is to do this during the resolution of layout, with some *meta-layout* function surrounding content and XSLT code to generate replacements: evaluation of the function by the layout processor makes a suitable choice between children, somewhat similar to the **only-child** function described in section 6.3.

Whilst a number of types of such function could be imagined, what is a minimal set that would allow document layouts to alter in the ways required: replacement, deletion and removal of further variability? Four functions have been explored – **hideSVG**, **removeSVG**, **null** and **last-SVG**. As will be shown it is important that which of these functions is to operate on a particular part of the layout tree can be changed – section 9.8 explains the mechanism.

An SVG subtree can be hidden by **hideSVG**. The result of layout evaluation of this function is an **svg:svg** with no properties (decorated **@ddfl:foreign**), all its children remaining unevaluated. Until the parent function (i.e. **hideSVG**) is altered to something else, they will never exert any influence on, or see the light of day without, their parent layouts⁵.

removeSVG deletes any contained child SVG subtrees. Its evaluation results in an **svg:svg** (again decorated **@ddfl:foreign**) with no dimensional properties and containing only ‘foreign’ subtrees – i.e. all proper SVG is deleted. Foreign children may generate new SVG subtrees later, but these will always be removed until the function (i.e. **removeSVG**) is altered.

A subtree can be deleted permanently by the layout function **null**, which always yields an empty sequence. After execution, it and its entire subtree disappear completely and have no influence on any parent both now and for the rest of eternity. This is obviously a ‘final’ rather than an initial operation during a series of document layouts.

The most useful operation is modifying or replacing an element by surrounding it and some replacement generator, and possibly other elements in foreign namespaces, with the function **lastSVG**. The result of the function is an **svg:svg** group with the (dimensional) properties of the *last* **svg:svg** from evaluating all the children. This group contains, as result children, all other foreign elements and *only* that last **svg:svg**, arranged strictly *in document order*. The result is not ‘foreign’ and therefore may contribute to, and be visible in, its parent layout. Figure 70 shows the implementation code (**hideSVG** and **removeSVG** are simpler versions).

```

match:ddfl:layout[@function='lastSVG'] mode="ddfl:layout"
  (element(*) children=
    => * mode="#current"

  transfer=@transfer
  last=($children[not(ddfl:foreign(.))][last()])
  svg
    $last/(@width,@height),@transfer,@x,@y
    if:$transfer='position'
      $last/(@x,@y)[not(. = 0)]
    @ddfl:layout=@function
    ∇ $children :
      choose
        when: not(self::svg:*)
          •
        when: is $last
          copy
            @*
            if:$transfer='position'
              @x=0
              @y=0
            *|text()

```

Figure 70. Layout function to support tree replacement

⁵Systematic use of SVG's **@display** property is an alternative and discussed in section 13.3

The meta-layout function **lastSVG** is used for replacement in the following way, illustrated through an simple example shown in Figures 71, 72 and 73. A piece *P* that may be replaced is grouped under **lastSVG** with some retained generating construct (XSLT) that may build a replacement under suitable conditions – the generator is placed *after P*. Assume that during some XSLT phase the new constructor does not fire, remaining dormant. The evaluated layout is an **svg:svg** group with *layout(P)* and the generating construct as children, and the dimensional properties of *layout(P)*. For its parent layout it presents the same extent as the visible piece. Nothing will change while the XSLT remains dormant.

```

layout(flow) x="0" y="0"
  rect fill="red" width="50" height="30"
  layout(lastSVG)
    block width="50" font-family="Helvetica" font-size="6"
      This is an fo:block that might get replaced eventually.
    if:test2 retain="until-triggered" evaluate="false"
      block width="50" font-family="Helvetica" font-size="6" font-weight="bold"
        val(test2/data)
  rect fill="blue" width="50" height="30"

```

■ ddf: ■ svg: ■ fo: ■ xsl: ■ ddf: text

Figure 71. Original document

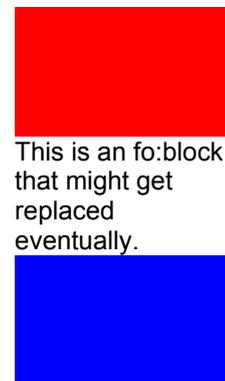
```

svg layout="flow" height="87.58" width="50" x="0" y="0"
  rect fill="red" height="30" width="50" x="0" y="0"
  svg layout="lastSVG" height="27.58" width="50" x="0" y="30"
    svg element-type="text-block" font-family="Helvetica" font-size="6" height="27.58" width="50"
      svg
        ...This is an fo: block
      svg
        ...that might get
      svg
        ...replaced
      svg
        ...eventually.
    if:test2 retain="until-triggered" evaluate="false"
      block font-family="Helvetica" font-size="6" font-weight="bold" width="50"
        val(test2/data)
  rect fill="blue" height="30" width="50" x="0" y="57.58"

```

■ svg: ■ ddf: ■ xsl: ■ ddf: ■ fo: text

Figure 72. First stage binding and layout



When the XSLT eventually triggers it produces a new element (the replacement **fo:block** with a data value interpolated for its text in the example of Figure 73) which is then evaluated as

a layout instruction. Now the result of evaluating **lastSVG** focusses on the *new* piece of SVG (the evaluated text block) which appears *after* the original in document order and consequently the old section is discarded and the dimensional properties of the new-to-parent layout are projected onto the result. Whether the XSLT fragment is retained after first triggering is a matter for its properties – in this case it disappears after execution.



Figure 73. Replacement of text at second binding and layout

Thus far it appears that these four meta-layout functions are sufficient to support programmatic alteration of the SVG tree from embedded instructions, provided we can switch between them. For example, to display some graphic when a data-condition is true we need to use **lastSVG**; when not present **hideSVG** will mask the element. How this is done in a consistent and continued manner is the subject of the next section.

9.7 Retained XSLT

We need to be able to execute and retain sections of XSLT within these layout trees. Three different embedded XSLT constructs have been shown in the examples above. There are other forms to be considered and as many of the standard ‘pull’ constructs as possible should be supported. The major problem is to ensure that we can control when pieces of XSLT both might possibly execute during the XSLT evaluation phase and also appear in some form within the output of that phase. Normally the ‘retained’ form is unmodified (and therefore capable of repeat performance during a subsequent XSLT evaluation), but there are circumstances where the form of the XSLT fragment might be altered.

Systems where program generates program are complex. We often need program fragments to both generate output as well as copies of themselves. The DDF framework has a compiler that converts the document into an executable XSLT program, so code modification can be carried out by that tool, controlled by directives attached to fragments⁶. The code required is principally a small set of additional templates, operating in mode **ddf:modal**, pre-empting those shown in Figure 21.

The first directive describes the mode of *retention* – whether and under what conditions a section of code should appear in the output as well as having been possibly invoked during processing. There are at least five different simple forms:

- **unretained** – execute once during any first XSLT pass, interpolate and disappear. The usual XSLT behaviour.
- **retain while condition (continue-while)** – execute the construct. Leave a continued form of the construct whilst some condition holds.
- **retain indefinitely (indefinite)** – interpolate if possible and remain under all circumstances – probably positioned *just after* any interpolation in document order, though order might be controllable⁷.
- **retain until positively triggered (until-triggered)** – the construct remains dormant until a positive test or selection occurs – then interpolation occurs and the construct disappears.
- **retain until negatively triggered (while-triggered)** – interpolation occurs and the construct is retained at each evaluation while a condition holds true. The construct disappears (and is not executed) as soon as the condition *fails*.

The condition to be tested can either be described directly (as an XPath expression in **@ddf:test**) or, for some XSLT instructions (**xsl:if**, **xsl:*[@select]**), it can default to the XPath interpolation involved in the absence of such a test. The test conditions must be accessible from the context node being examined at the time of XSLT evaluation. An example occurs in Chapter 12 where program elements are retained while the data contains a ‘to be continued’ marker.

⁶Attributes in foreign namespaces can be attached to XSLT elements without interference to XSLT evaluation.

⁷Equivalent to **continue-while** with a **true()** condition, but sufficiently common to treat specially.

To make these effective XSLT operations we need to convert them to compound constructs that involve generating XSLT in the output of an XSLT pass. For example **xsl:if test="expression" ddf:retain="until-triggered"** would convert to:

```

choose
  when:expression
    body
  otherwise
    XSLT:if test="expression" retain="until-triggered"
      body-warpedXSL

```

where **XSLT:** will be generated as **xsl:** in the output and **body-warpedXSL** is the body of the **if** ‘warped forward’ so that all XSLT elements contained are generated in the output rather than evaluated in place. This ‘warping’ operation is a sufficiently common requirement in the compiler that a specific push mode (**warpXSL**) performs this action. A moment's thought will show that if the test is triggered only the consequence remains. If not then an equivalent **xsl:if** structure will appear in the output. On a subsequent binding pass, provided the document is again recompiled, similar behaviour will ensue⁸. In the absence of triggering the construct is *idempotent*. The compiler can perform this conversion using the following template:

```

match:xsl:*[@ddf:retain='until-triggered'] mode="ddf:modal"
  XSLT:choose
    XSLT:when test="{(@ddf:test,@test,@select)[1]}"
    •
  XSLT:otherwise
    => • mode="warpXSL"

```

9.8 Hybrid XSLT/meta-layout action

Using a hybrid approach between retained XSLT and meta-layout agents to achieve the structural changes to the tree is best illustrated by an example where a graphic construct appears and disappears depending upon the bound data. With four stages of data binding, the second and fourth of which contain an element **<on/>**, the results look like Figure 74.

The graphic is constant so all that is necessary to do is to alter the ‘visibility’ to the layout system depending upon satisfying the ‘display’ condition. However this is where we meet an

⁸A DDF document is **not** an XSLT stylesheet, so some form of conversion is always required before execution of XSLT semantics and ‘constant’ structure within the DDF document is desirable.

issue with the XSLT processing model of generating a result tree with functional dependency on an input tree. An *element* once written into the result tree cannot be altered by any instructions buried within its defining tree – child elements can be added to the empty ‘shell’ but the element itself cannot be changed.

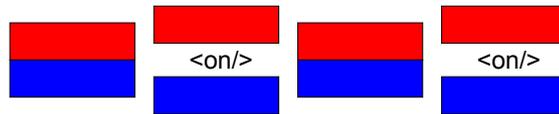


Figure 74. Four successive bindings of a document

The attribute properties written onto the element (before children) cannot be deleted but fortunately their values *can* be overwritten – as we will see this is a key feature, but the means of overwriting is strictly limited. Such modifications must happen *before* any child elements or text nodes are added; any variability can depend only upon the input tree, not the element and its properties just written. For example it is not possible to read and modify a property, e.g. **attribute name="font-size" select="@font-size + 4"** within an **fo:block** will not read the font-size already defined, but rather look for some property on the input data, if any⁹.

However the ability in XSLT to overwrite an attribute value is *just sufficient* for our needs, letting us change the name of the required layout function between **hideSVG** & **lastSVG** and setting and unsetting the ‘foreign’ status:

This is a hybrid action: the XSLT phase can only *add* to the tree within which the XSLT fragment is embedded, but it can add information such that a subsequent process (i.e. the layout) will *not copy* parts or *modify* such parts during the copy, according to those instructions. But this signal can only be varied based on the input data, not the direct state of the tree itself – hence we cannot build a flip-flop: the graphic presentation does not have a state that is readable by XSLT embedded within it.

KEY: Hybrid action. Modification of processed layout is performed by the XSLT phase adding new components or changing signals in the layout tree, and the layout phase when evaluating *meta-layout* functions, choosing *not to copy* some components based on those signals.

⁹This is not strictly correct – there are methods to read the defining XSLT stylesheet (**doc("")**) which might permit a convoluted mechanism to do this in some strictly static circumstances, but this not at all general.

Conditional revelation of a variable element (i.e. the graphic is not constant) uses the same general approach (Figure 76) but with **removeSVG** and **lastSVG** controlling the surrounding layout and a guarded generator for the graphic:

```

layout(flow) x="0" y="0"
  rect fill="red" stroke="black" width="50" height="15"
  svg layout="hideSVG" foreign="true"
    choose retain="indefinite"
      when: */on
        @ddfl:layout=
          lastSVG
        @ddfl:foreign=
          false
      otherwise
        @ddfl:layout=
          hideSVG
        @ddfl:foreign=
          true
    block padding="1" width="50" text-align="center" font-family="
    Helvetica" font-size="10"
  <on/>
  rect fill="blue" stroke="black" width="50" height="15"

■ ddf: ■ svg: ■ xsl: ■ ddf: ■ fo: text

```

Figure 75. Programmatic hiding and revelation of graphical content

The order of the constructs is *critical* – anything that alters attributes *must* follow the parent element head immediately – it is an execution error if any attribute is written after the construction of a child element or text node¹⁰. These constructs must therefore be written with exceptional care, or preferably generated from some ‘higher-level’ directive, either through a macro package or with a specialist module attached to the compiler.

These compound hybrid forms have been described as additional **ddfl:** instructions, but ones that are recognised by the DDF compiler through a plug-in module. (This is the only point thus far where the DDF compiler has any knowledge of the existence of the layout processor – even to the namespace for **ddfl:**.)

The paradigm shown in this example is supported by the **<ddfl:revealSVG test="show">** instruction which the compiler will expand to the necessary form. The expansion is smart enough to determine whether the graphics involved are data-invariant (i.e. **empty(*!(xsl:* | @***

¹⁰This is to support indefinite serialisation in demand-driven processing – when an element or text constructor is encountered, the opening parent head node can be closed permanently. This restriction also means that multiple retained **xsl:attribute** elements must be handled as a complete group, ensuring that the interpolations all appear before any retention statements. Code ‘robustness’ is discussed in section 13.1.

`[contains(., '{'})])` and uses the most efficient construct. The `ddfl:revealSVG` instructions can be nested.

```

layout(flow) x="0" y="0"
  rect fill="red" stroke="black" width="50" height="15"
  svg layout="removeSVG"
    choose retain="indefinite"
      when: */on
        @ddfl:layout=
          lastSVG
        @ddfl:foreign=
          false
      otherwise
        @ddfl:layout=
          removeSVG
        @ddfl:foreign=
          true
    if: */on retain="indefinite"
      block padding="1" width="50" text-align="center" font-family="Helvetica" font-size="10"
        @fill=(*@colour, 'white') [1]
        <on/> in
        val(*@colour)
  rect fill="blue" stroke="black" width="50" height="15"

```

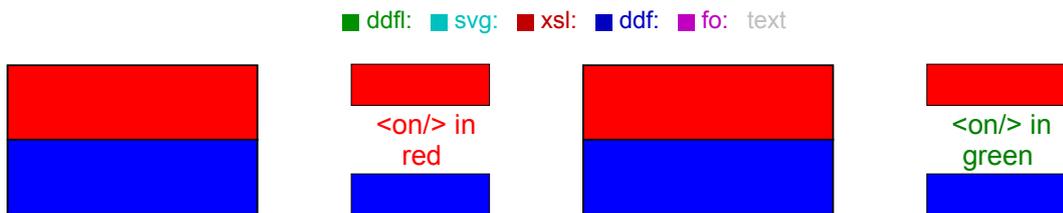


Figure 76. Four successive bindings with a data-variable graphic component

9.9 Conclusion

There are parallels in the methods described in this chapter with the decorated SVG approaches of Marriot *et al*[3, 74] – they permit SVG attribute properties (position and size mostly) to be replaced by acyclic expressions or ‘attached’ to edges of constraint graphs that are then solved dynamically to satisfy the layout intentions. Thompson, King & Schmitz[49, 93] similarly add additional declarative structures in SVG to describe event-based adaptation. Both of these involve constant tree topology.

Here we decorate a canonical form of SVG with layout properties in a foreign namespace, such that the SVG can be modified appropriately to re-satisfy those intentions in the presence of alterations. Such alterations could be confined to changes in property, but the most interest-

ing are due to modifications in *topology* caused by embedded information or operations in other spaces, such as evaluations in XSLT.

The model developed here is a two-pass hybrid one – a programmatic phase using XSLT and a layout resolution that turns declarative layout intent into grounded graphical constructions.

There are five important points:

- Information for both phases is in the form of XML tree components that may be, and usually are, totally interspersed.
- Both phases tolerate the existence of information for the other phase, behaving as good XML citizens.
- Both phases may leave information in their results for later resatisfaction or re-evaluation, either as attributes or as elements.
- Re-evaluation information is constructed either by a compiler building ‘self-propagating’ code or by layout processing agents adding information during result computation.
- Modification of processed layout is performed by the XSLT phase adding new components or changing signals in the layout tree, and the layout phase choosing *not to copy* some components based on those signals.

As the implementation of both of the phases is entirely functional, consisting of tree construction with *no* in-place modification or reassignment, then there is a degree of robustness about the approach. It may be possible to develop a limited calculus for determining the consequences of this method, that may have parallels with systems of analysing XML deltas[95].

The ideas presented in this chapter are the key ones from this thesis. In the next three chapters they will be used to explore partial evaluation, ‘higher order’ documents taking other variable documents as arguments and an extended example of a ‘continual life’ document.

Chapter 10

Partial Evaluation and Constant Folding

This chapter examines some of the ways documents can be evaluated partially, either because data is incompletely bound, or as an optimisation step to perform as much work as possible before data is bound. Two specific approaches are presented: i) arranging that a document can respond correctly to incomplete binding, by suitable declaration of guard conditions and ii) folding constant (invariant) sections of the layout tree, by identifying such and evaluating to grounded graphics.

DDF documents may have very large and heterogeneous trees defining layout constructions. At late stages of binding the medical record example that we shall meet in Chapter 12 has some 1700 elements defining its layout. At another extreme, the presentation declaration for this thesis has around 33,000 elements with more than 100,000 attributes. When there are very large datasets being processed, it may be prudent to find if significant amounts of work can be performed ahead of time.

10.1 Partial data binding

A possible variable document is a generic template that can be specialised through one or more intermediate stages before final production of large numbers of variants. Some of these can be through simple ‘theme’ bindings, often straightforward styles, but of more interest is when

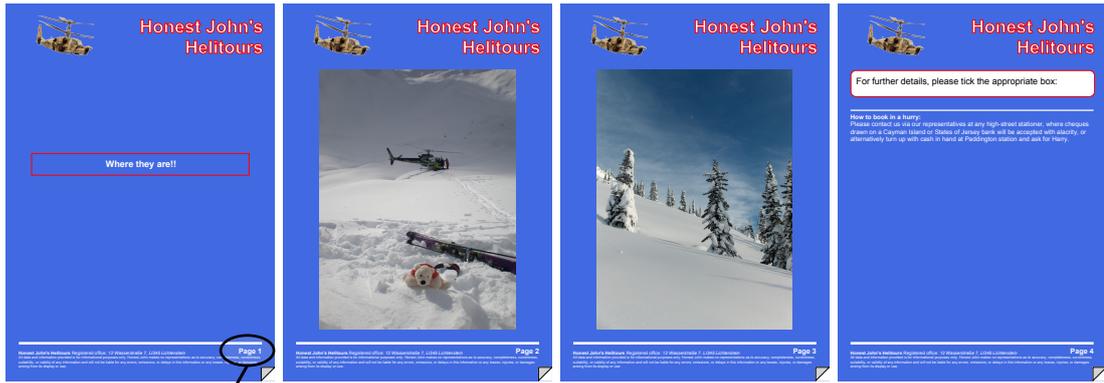
some, but not all, of the data is bound. For example the travel brochure example can be considered to have two distinct data sections – that related to the company, and that related to the customer and their specific offers. We could imagine a two-stage process of binding: first with the company data to give a new ‘template’ and then across a vector of customers at some later date.

Whilst we could construct a template that deliberately built another, it would be preferable to support such partial data binding automatically, or, at the least, with a small number of embedded directives attached to the document to indicate areas of possible partial binding. Some directives to control retention of programmatic constructs were introduced in section 9.7 – these, along with a few other modifications supported by the document compiler, can produce the required effects and lead to more fully-automated systems. The first experiment uses embedded directives supported by a ‘compiler-driven’ approach that operates automatically.

Again the travel brochure is used as the example and it is assumed that the company data is bound first. All variability ultimately arises from XSLT fragments – in this document there are three distinct top-level types: **xsl:template**, **xsl:function** and **xsl:attribute-set** and a series of embedded interpolating or choice statements inside the result XML trees (**xsl:value-of**, **xsl:for-each** etc.) To successfully bind the ‘company’ information during the first phase, we can just let the normal evaluation proceed.

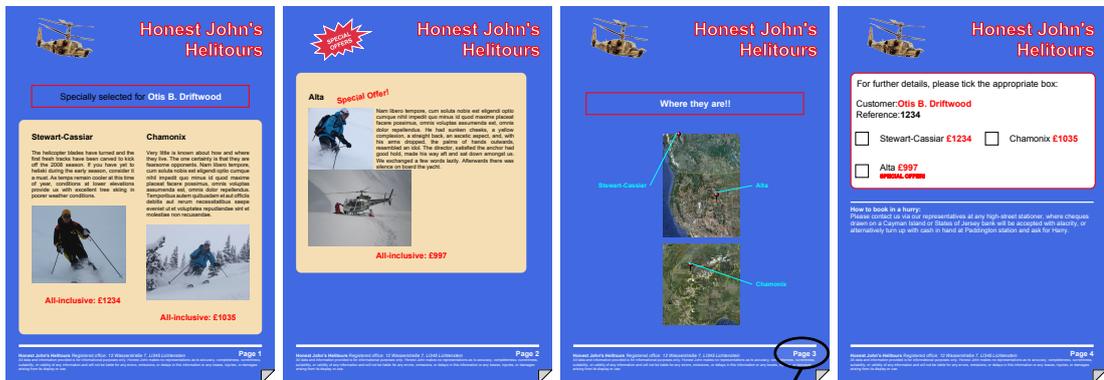
But we also need to arrange that ‘not company’ statements are not executed, and retained for some future occasion. (As most document interpolations are ‘positive’, not involving negative tests, if we let them proceed we usually get null-sequence results – this doesn’t break the document, but removes any further interpolation.)

Figure 77 shows a correct result through the two stages: in a) the eventual leading pages (which contain the customer name and various resorts over a number of pages) have not been generated. The first page shown is the background for the ‘maps’; the last is for ‘further interest’, populated with static data and a layout that will subsequently contain customer details and resort check-boxes. As the document is required to be a multiple of 4 pages in length, two fillers have been added during the first binding. Figure 77b shows the completed document after a customer and offers have been bound – the addition of two pages has made the fillers unnecessary and they have thus been ‘removed’. Note that the page number for the ‘maps’ display has altered, as its ordinal position in the document has now changed.



Page 1
 completeness, currentness,
 accuracy, or damages

a) First to company...



Page 3
 completeness, currentness,
 accuracy, or damages

b) ... then to customer

Figure 77. Two-stage binding of a variable brochure

Our data model splits into two halves – **company** and **(customer | resort)**, so interpolations concerning the second group can be ‘guarded’ as outlined in section 9.7. Declarations of the form **ddf:retain="until-triggered" ddf:test="exists(brochure/customer)"** are suitable. The ‘top-level’ constructs (templates, functions..) are retained forward by declaring **ddf:retain-pattern="..."** for those forms needed – they have no effect unless ‘called’ from result trees, so will not harm the document if not required. Figure 78 shows the points that contain such directives.

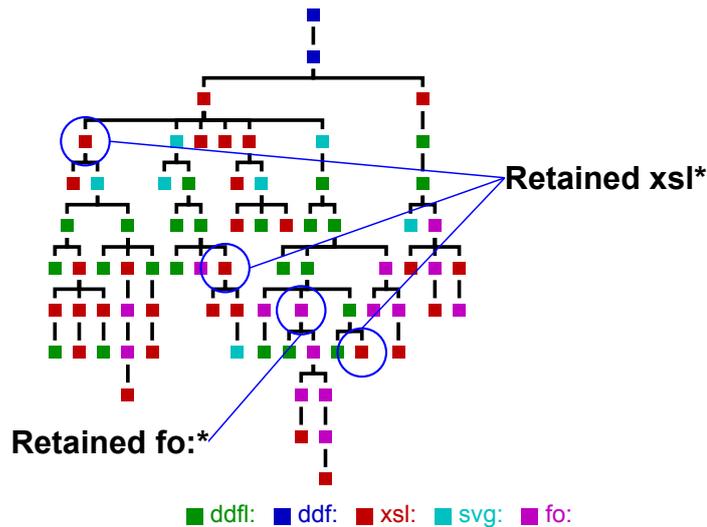


Figure 78. Retention declarations for main page generators

In most cases an **xsl:*** element is being guarded – the test is that executing the **@select** XPath produces a non-null result. But one of the text blocks contains interpolants over customer data:

```

block retain="until-triggered" font-family="Helvetica" font-size="20pt" margin="2" padding="2" text-align="left"
  @width=$background//*[[@name='blank']]/@width
block
  Customer:
  inline fill="red" font-weight="bold"
    val(brochure/customer/name)
  block
  Reference:
  inline fill="black" font-weight="bold"
    val(brochure/customer/ref)
  
```

Partial evaluation of line-wrapping text blocks will be difficult and of little value – everything *after* a retained variable will need to be re-satisfied¹. Assuming that every interpolant is expected to produce a value, then the condition for evaluation of such a block is that all the interpolants yield non-null results. Thus for our example, a suitable form is:

¹And with an optimising line-wrap, such as Knuth-Plass, the entire block would have to be re-evaluated.

```

if:exists(brochure/customer/name) and exists(brochure/customer/ref) retain="until-triggered"
block font-family="Helvetica" font-size="7.056" margin="2" padding="2" text-align="left" width="190"
  block
    Customer:
    inline fill="red" font-weight="bold"
      val(brochure/customer/name)
    block
      Reference:
      inline fill="black" font-weight="bold"
        val(brochure/customer/ref)

```

■ xsl: ■ ddf: ■ fo: ■ ddfi: text

That transformation can be carried out by the document compiler, matching cases of **fo:block** [**@ddf:retain**]. Supporting these directives gives a result, after evaluating over company data, as shown in Figure 79.

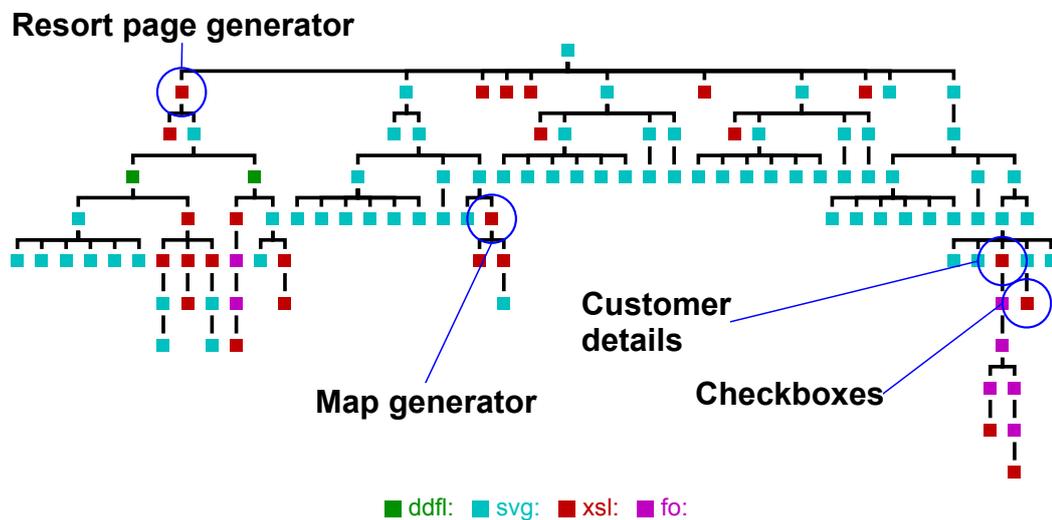


Figure 79. Retained variability after company binding

The layout agents involved (flow, grid, encapsulate) can all tolerate embedded **xsl:*** fragments so they will adapt correctly during subsequent data bindings. The brochure may contain a number of ‘filler’ pages to round the total page-count up to a multiple of four. The document design makes the number of fillers required be determinable through XSLT operating on the data binding. On binding to the company only, an additional two fillers are needed – none are required subsequently for the particular customer shown. This optionality could be supported by some page-level layout function that generated or removed the filler pages; however using SVG's **@display** property on complete **svg:page** elements is an easier method. The page numbering function, which interpolates page references, is a one-to-many mapping: re-satisfaction is supported as described for pagination in section 9.4 and demonstrated in Figure 77.

We have now shown that a series of appropriate directives added to a document can control some partial binding of data. The rest of this section explores how these can be derived automatically. But there is a fundamental issue about inferring the *intent* of constructs in the document that requires information from some external *oracle* to resolve ambiguity.

Consider the interpolant **brochure/customer/title** when used within a text block. In partial binding, what can we infer from the absence of **title** for a customer – that there will never be a title for this person or *it is just not bound yet?* This cannot be determined from the document code itself, unless it has been decorated with hints during its design. Similarly large programmatic sections processing a **saleitem** would merely yield a null result in the absence of any **saleitem** during first binding.

One possibility is to use a model of the data, such as a *schema*, which describes ancestral relations between named elements, text nodes and (attribute) properties, and optionality in the structure – whether and how many child elements of a given type are required. However a single schema doesn't usually describe partial binding – the use of multiple schemas is needed for that, though they will have probable overlap. Figure 80 is a very simplistic model for the structural level of the brochure at two stages of binding²:

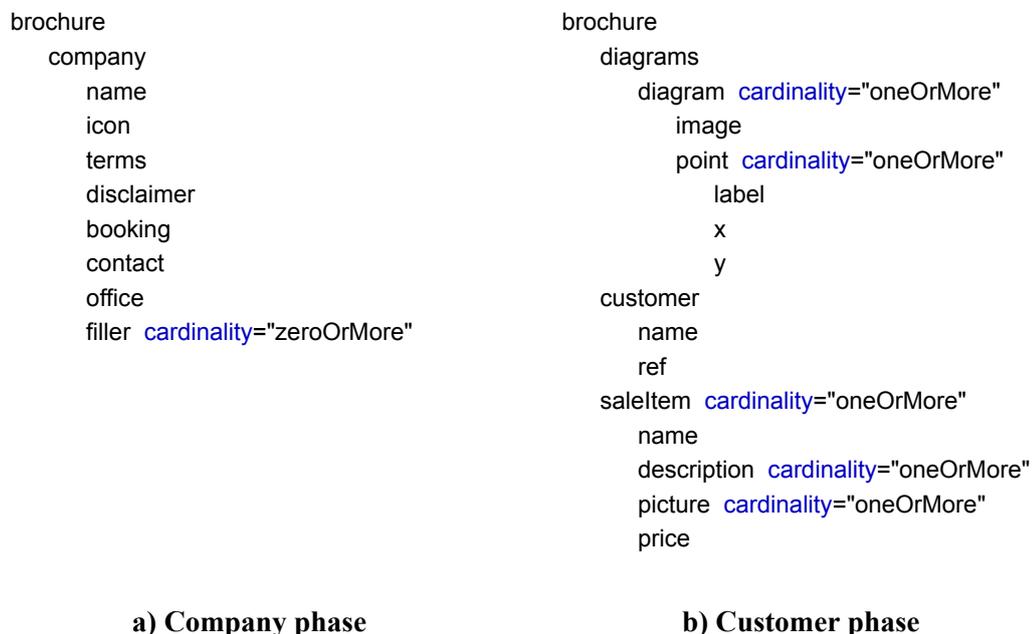


Figure 80. Simple schemas for brochure data

²Using a standard schema language, such as RelaxNG, would have been too cumbersome for this illustration. Focus has been on the structural layer since most variability is in the presentation, which derives from information held in **ddf:struct**.

Using these schemas and work-flow knowledge that data for the company is bound first, followed by customer, we can infer that any interpolating statement (XPath expression) that involves a descendant axis of **company** can be left to bind to conclusion on first phase; those descending from **customer | diagram | saleItem** will need guarding. But if the schemas are not as separable (i.e. *customer* and *company* schemas provided separately), such inferences become much more difficult. Such schema-driven approaches are not pursued in this thesis.

An alternative oracle could be for the designer to provide high-level hints, strategies or heuristics, probably placed on the document. As an example, most documents are built around ‘positive’ and ‘one-off’ interpolations – the presence of data causes content to be generated and then not altered thereafter – this data may be invariant (e.g. a ‘terms & conditions’ block when **TandC**) or variable (**xsl:value-of select="customer/ref"**). If we assume that *all* the interpolants must generate content, the following cases will need ‘until-triggered’ guards:

- **fo:block[./xsl:*]**. Text blocks with embedded variability. The test is, as described above, that every embedded XPath interpolant produces a non-null result. The test needs only be carried out at the uppermost **fo:block**.
- **xsl:for-each(-group) | xsl:apply-templates**. The tests for these are their **@select** XPath expressions. If we assume their bodies (or invoked templates) only take information from the selected *subtree* (i.e. do not investigate high ancestry, and are unaffected by ‘cousin’ subtrees) then guards are not needed at lower levels, nor need they be ‘promoted’.

Using these simple tactics we can transform the presentation section of our (undecorated) brochure document to that shown in Figure 81, where the added guards are identified. Running this modified document through the two successive phases of data binding yields the same results as shown earlier. Tests can be refined to support document-borne hints (e.g. **@select="brochure/customer/title" @ddf:as="text()?"**, which declares that the title is effectively optional).

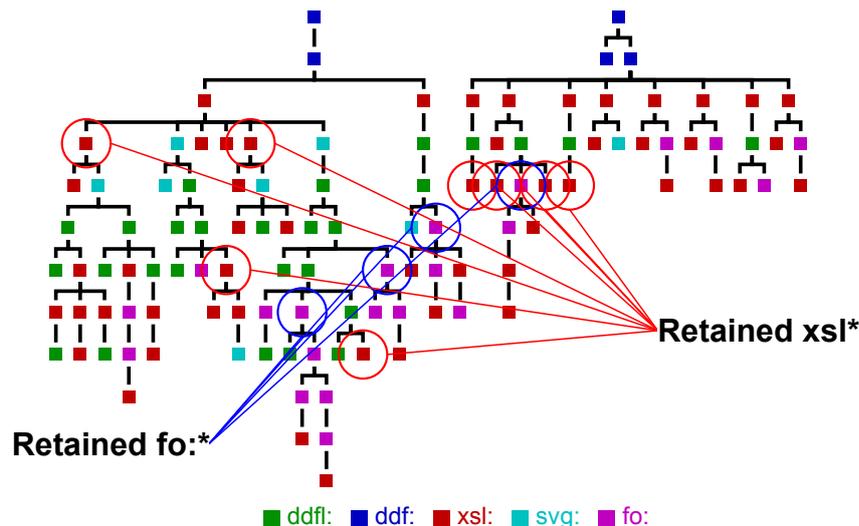


Figure 81. Automatically generated retention declarations

Heuristic approaches are also possible. A large tree existing under **xsl:for-each select="saleItem"** implies that we *are* expecting to produce something for a **saleItem**, whereas the interpolation of **@special-offer** in:

```

<fo:block >
  <xsl:if test="@special-offer">
    <xsl:attribute name="background-colour">red</xsl:attribute>
    <xsl:attribute name="fill">white</xsl:attribute>
  </xsl:if>
  <xsl:value-of select="description"/>
</fo:block>

```

appears to be highlighting the appearance of the text under special conditions and thus can be inferred as an optional statement.

10.2 Constant folding of invariant layout

When the set of document instances to be produced is very large, and the variation between instances modest, then it may help to pre-evaluate as much as possible of the layout in some pre-processing or compilation stage, or at some equivalent point during a multi-stage binding. How effective this can be depends upon the type of document, the size of anticipated instance sets and the susceptibility to such pre-evaluation of both the document's presentation model and the implementation technology that generates the final presentation.

To extract the maximum evaluation possible requires a detailed knowledge of the semantics of the layout combination. For example a uni-directional 'flow' of a sequence of parts can

(by associativity) be replaced by a flow of partial flows³; completely bound flows can be evaluated and treated subsequently as an atomic piece. Macdonald [70] describes some of the problems involved in geometric layout of partially bound assemblies, focussing on exploiting invariances such as this.

But something as simple as a self-sizing table cannot be broken down completely into independent sub-tables, as the size of cells can depend upon the size of new-cells ‘yet to be bound’. So in these cases we may have to resort to speculative evaluation, providing test choices to determine whether pre-evaluated or re-evaluated alternatives should be used. These speculative possibilities are not explored in this thesis, though the general mechanisms to support this could be built relatively easily given the current framework.

However it is possible to identify, relatively simply, *some* parts of a variable document's layout that can be pre-evaluated correctly, by focussing on invariance. In compiler technologies this is *constant folding* – evaluation of partial terms that are constant with respect to any variability a program may have. Again the travel brochure is the example, though the specific document is slightly simpler and has more inherent layout invariance⁴. Variable data is interpolated in two places: the customer name into part of the header and resort/holiday descriptions into the (finite) flows of the containers.

Figure 82 shows two parts of the presentation of this document: the page ‘background’, which has been assigned to a variable, and the second page, which uses this background and interpolates the layout for the third (and fourth) ‘resorts’ over that background.

³Though this violates the goal of *tree-isomorphism*, by introducing additional layers – sibling relationships may now become cousin or uncle/aunt.

⁴This section is a mild reworking of the DocEng2010 paper[61].

```

background=
  svg height="297" width="210"
    rect fill="royalblue" height="297" ...
    block fill="white" font-family="helvetica" ...
      Honest John's HeliTours
    block fill="white" font-family="Helvetica" ...
      Honest John's HeliTours ... Registered office:...
page
  layout(encapsulate)
    copy-of($background)
    image xlink:href="cref://inputrun-CLI-step.w188aaa.d193e73.d281e12/special.png" height="
40" ...
    block background-color="royalblue" border-style="solid" ...
      inline font-weight="bold"
        How to book:
      block
        Please contact us via our representatives....
    layout(flow) background-color="wheat" direction="x" ...
    => brochure/resort[position() gt 2]

```

■ ddf: ■ svg: ■ fo: ■ unknown: ■ xsl: text

Figure 82. Presentation for the brochure background and second page

The computational cost of a layout depends on the complexity and the number of items to be processed. In practice however the cost of normal document layout is dominated by line wrap in text paragraphs – the numbers of characters dwarfs all other entities in the document and every character must be treated, at least to determine character width.

The presentational model is a mixture of elements directly in layout and geometric spaces (**svg**:,**fo**:,**ddf**:) and in ‘programmatic’ spaces (**xsl**:). The effect of data variation within this model *only* manifests itself through interpretation of those programmatic (**xsl**:) elements. Figure 83 shows code declaring a simple flow with some elements repeating for each customer, with the *name* variable for each customer :

```

<ddfl:layout function="flow">
  <svg:svg name="A">...</svg:svg>
  <xsl:for-each select="customer">
    <ddfl:layout function="flow" direction="horizontal">
      <fo:block name="C">Most valued customer:</fo:block>
      <svg:rect name="D" width="2" height="40"/>
    </ddfl:layout>
    <fo:block name="E">
      <xsl:value-of select="name"/>
    </fo:block>
  </xsl:for-each>
  <fo:block name="B">The company assumes no responsibility</fo:block>
</ddfl:layout>

```

Figure 83. Interspersed layout and program

For this flow, the elements A and B are completely constant, the text block E is variable and whilst the number of elements C and D is variable, they each contain invariant parts. A and C, the **svg:*** elements, are in final grounded forms, but the **fo:*** blocks still require layout and thus those that are invariant can be processed irrespective of any data values. Moreover the horizontal flow above C and D is thus independent of data, so it too could be evaluated.

These invariances identified are independent of the semantics of the layout – if the parent of C and D were some other layout function it would *still* be invariant and thus evaluable.

KEY: Invariant sections can be identified without any detailed knowledge of what the layout functions actually are.

It is sufficient to be able to identify ‘variable-containing’ regions and cross-tree linkages to such regions. This leads us to a tentative conservative tactic:

*If a ‘geometry’ node has no **xsl:** descendants or properties, or descendants with such properties, then its layout is **invariant** to data, and may be evaluated and replaced by the grounded form.*

Influence from the **xsl:** namespace can occur in three separate places:

- Descendant nodes: **xsl:***
- Attribute value templates (**name="{XPath}"**), which similarly interpolate a variable value to the property.

- Use of attribute sets (**xsl:use-attribute-sets**), which add attribute properties to a node from defined named sets of attributes. The values of such attributes can be static or XPath expressions and functions of data context, though the latter use is uncommon.

The first two conditions can be detected succinctly by XPath predicates of the general form **empty(descendant::xsl:* | descendant::*/@*[contains(.,'{}')])**. Practice is not so simple: there are more complex issues that need attention:

- The use of a **descendant::xsl:*** test can be expensive ($O(d*n)$ where n is the number of nodes in the whole tree and d is the average tree depth).
- Some elements (e.g. **fo:block**) cannot be evaluated *by parts* easily (partial line-wrap is very difficult) and so must be processed in their entirety or not at all.
- Blanket detection and exclusion of **xsl:use-attribute-sets** is too conservative – most use of attribute sets involves static values, independent of data. Limited analysis and interpolation of attribute set values may be needed.
- The DDF layout model includes single-assignment presentational variables. Whilst their values may be invariant, their interpolations must be identified and analysed.

The last two issues are examples of *cross-tree* influence – where an element refers to information stored elsewhere in the tree, normally through a name or other form of identification. (The **svg:use** construct is similar.) Knowledge of such cases is needed to make this technique conservatively successful.

The implementation chosen was to employ a bottom-up pre-traversal of the tree, which identifies invariance (absence of attribute value templates, no **xsl:** children and all children being invariant, any referential element pointing to invariant definitions) and in the same pass interpolates the values of attribute sets. Then the main invariance processing pass is performed, top-down, whilst interpolating presentational variables: invariant sections are evaluated; variable ones are copied and then processed recursively.

Processing attribute sets

Interpolating **xsl:use-attribute-sets** directives will need evaluation of some part of the XSL model. If we assume that attributes in sets have simple static values, such as **<xsl:attribute name="foo">bar</xsl:attribute/>** then interpolation is relatively straightforward. We collect

all applicable attribute sets, every attribute of which is guaranteed static, and then process references, interpolating attribute values and removing the attribute-set references only if every required set is available in the static collection⁵. The code is comparatively simple:

```

required-sets=tokenize(@xsl:use-attribute-sets,'\s+')
if:every $r in $required-sets satisfies $r = $static-attribute-sets/@name
  copy
    ∇ $required-sets :
      thisSet=$static-attribute-sets[@name=current()]
      ∇ $thisSet/xsl:attribute :
        @{@name}=•
      copy-of((@* except @xsl:use-attribute-sets), *|text())

```

Presentational variables

Single-assignment presentational variables can be very effective in building coherent documents: in our example the page background is defined as such and reused. But such a variable could contain data-dependent properties or components making its layout variant. Thus any *interpolation* of that variable's value, in some other layout, denies that layout invariance too.

Rather than excluding any case of presentational variable interpolation, a more reasonable approach is to analyse the invariance of each instance and then examine all variable use. If any variable use within a layout refers to a *variant* presentation variable, then invariance is denied to that element and thus it will not be evaluated. A simple implementation focusses only on direct interpolations of single variables (**\$background** rather than **\$blank/@width**), recording the values of invariant variables during recursive tree descent. Invariant variable declarations are removed from the result and **ddfl:copy-of** directives that refer to invariant variables are interpolated directly. This actual interpolation is inexpensive; the real value is to permit larger dependent layouts to become invariant and thus pre-evaluated.

Results

Figure 84 shows the elements of the main example document displayed as a tree, where sections identified as being invariant are highlighted in pink.

⁵A minor modification can remove invariant attribute set references from a mixed collection e.g. "dynamic1 static dynamic2" → "dynamic1 dynamic2"

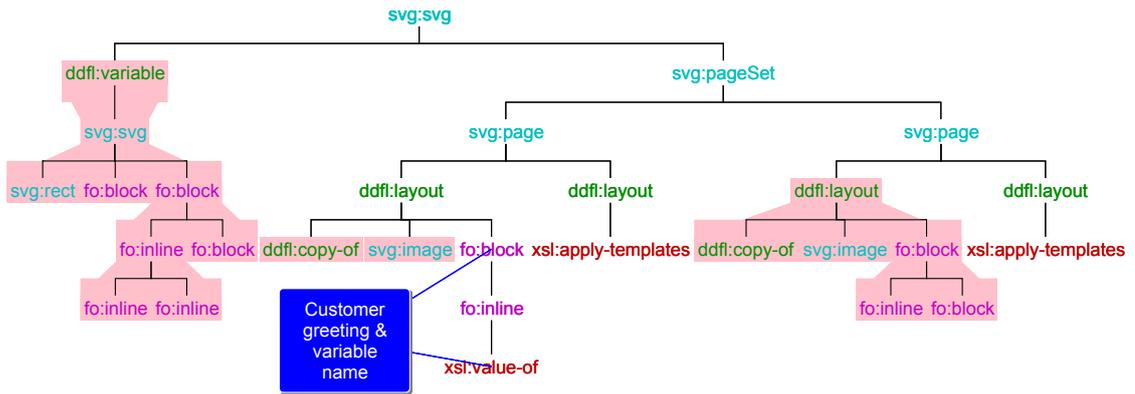
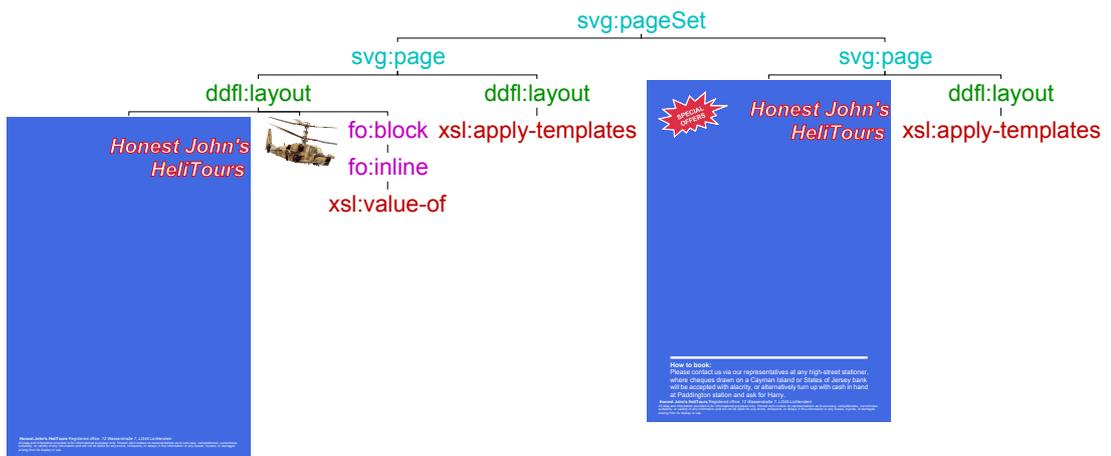
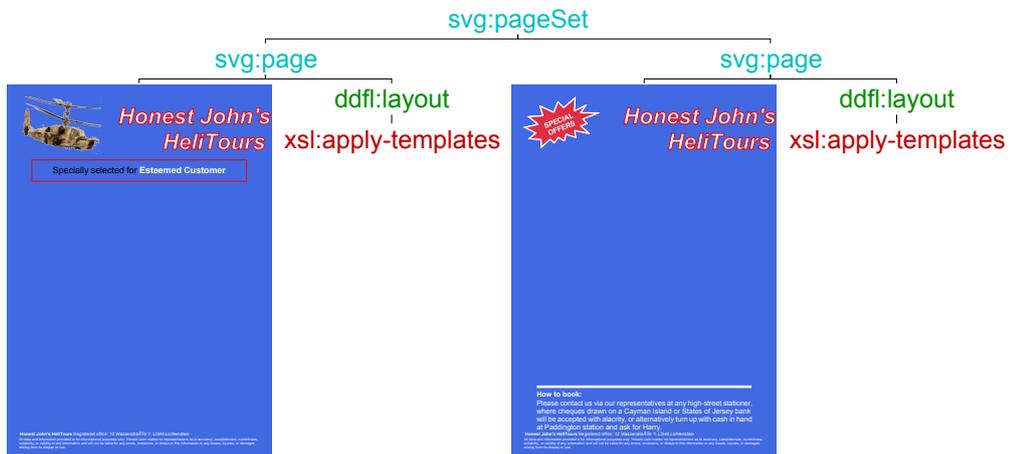


Figure 84. Document template with invariant sections identified



a) Variable customer greeting



b) Static salutation

Figure 85. Main document template after invariant processing

The technique has identified the background variable and its use on the first page as invariant and the entirety of the first layout on page 2 as similarly invariant. The ‘customer greeting’ (*Specially selected for Otis B Driftwood*) is a variable **fo:block** element on the first

page, which has been labelled in Figure 84. Figure 85a shows the result of processing all invariant parts available – clearly the customer greeting cannot be evaluated. If however it were static (e.g. *Specially prepared for a Valued Customer*), then that section becomes invariant and hence its parent layout as well. The result is shown in Figure 85b – the effect here is rather minor, but there could be consequential propagation of invariance – this technique would find it automatically.

This section has outlined a basic and very conservative approach to identifying invariance. Deeper analysis of XSLT and XPath semantics, and the (XPath) semantics of using presentational variables, could present further opportunities to extract and evaluate layout invariance, still without having knowledge of the semantics of layout functions themselves.

This technique could be used in other XML-based layout models provided there is a canonical rendered form that can be consumed subsequently by the layout without further substantial processing. (In this case SVG is both the final output form and consumable as laid-out input, subject perhaps only to translation, or some other operation that does not require the inner-tree to be processed.)

To determine *whether* this approach is worthwhile requires performance measurement on a large number of sample documents. This would need i) identification of the extent of invariance within them, ii) determining the computational cost saved by pre-evaluating those sections against the additional storage size of those rendered results and iii) measuring the cost of the pre-evaluation (which would only be important if production runs are relatively short).

Chapter 11

Active Documents as Variable Data

This chapter examines variable DDF documents consuming other DDF documents as *active* arguments, analogous to the usual meaning of ‘higher-order’ functions in functional programming. Examples in simple combination and applications such as imposition are discussed. It is concluded that the complexity involved makes such an approach so complex and special-purpose, that specific XSLT programs are a better choice.

A function behaves in a higher-order when it takes another function as an argument, and manipulates it (normally by evaluation) within some context. For functional programming languages this is a key capability, supporting concepts such as *maps* and *folds* which can be used to generate much larger robust complex functions.

Can something similar be achieved in documents? Can a variable document be written to consume other variable documents as ‘data’ and still be a variable document? This chapter illustrates that *documents as active arguments*¹ could be supported and discusses the necessary types of syntax, semantics and implementation alteration. Most of the discussion centres around the phase of XSLT semantic interpolation rather than layout evaluation.

Whilst higher-order functions in FP are usually *very* generic, exploiting highly regular rep-

¹As opposed to *passive* arguments, such as in auto-documentation where program elements in the argument documents are not carried forward as *program*.

representational models of the functions they consume (such as the Curry representation of multi-argument functions by ‘nesting’ single argument ones), within documents this is likely not to be so: as a consequence the forms described here will be limited in generality. The conclusion will argue that these limitations are such as to make this approach probably not worthwhile and special-purpose XSLT programs are a better choice.

(It has been suggested that some of the work in this chapter is closer to a highly-typed system than a generic approach to higher-order. There is some truth in this, as the higher-order documents considered here have very clear models of the argument documents they consume.)

Another difference is that higher-order functions in FP hold and execute (possibly partially) their argument functions during execution within the context of the execution of the higher-order function itself. In this work the document arguments are not normally executed, but rather manipulated and restructured by the higher-order document to produce a ‘source-form’ result document that can be executed at some subsequent time. Thus the ‘higher-order’ analogy might be tenuous, but the approach has some parallels.

Three types of manipulation are described: simple combinators, compound ‘expressions’ and an example of document imposition.

11.1 Simple combinators

One of the simplest examples of a higher-order function is a binary combinator – taking two arguments each of which is a function and producing a function that in turn will generate a single structural result. In Haskell this might be as simple as:

```
type Simple = (Int → Int)           binary double triple 4
binary :: Simple → Simple → (Int → [Int])    ⇒ {8,12}
binary p1 p2 = λx → [(p1 x), (p2 x)]
double x = x + x
triple x = x + x + x
```

An analogous situation for a document is a function that would combine two other documents each as a separate page, but which performs this combination before any data binding occurs:

```

data Page =
  Page {contents::String} | EmptyPage
  deriving (Eq, Show)
type Doc = String →[Page]
twoDocs :: Doc →Doc →(String →[Page])
(twoDocs d1 d2) = λs →(d1 s) ++ (d2 s)
doc1 s = [Page ("some process on:"++s)]
doc2 s = [Page ("another process on:"++s)]
twoDocs doc1 doc2 "variable text"
⇒[Page {contents = "some process on:vari-
able text"},
  Page {contents = "another process on:vari-
able text"}]

```

We can produce a similar effect with a DDF document using some of the technical additions that were developed for continual documents in Chapter 9 and a series of tricks and some special-purpose data structures. Figure 86 shows three single documents each of which produces an SVG page set². Each operates on elements of the same variable data schema (company information, customer details and resort/holiday descriptions), so in theory we should be able to combine these parts to make meaningful composite, but still variable, documents.



Figure 86. Three separate variable documents

Combining a pair of these documents with a similar ‘binary’ combinator will need to implement the information flow shown in Figure 87, where the appropriate sections of XSLT code are combined into the corresponding sections in the final result.

The immediate problem is that the usual ‘information flow’ within a DDF document starts at the data section, and passes through the logical structure to the presentation. Now **data** will contain the argument documents – we have to project and process the **struct** code from both of those into the **struct** in the result. Given the XML nature of the argument documents and suitable XPath selectors this is relatively simple, when the code combination is pure concatenation. The actual source of the binary combinator is shown in Figure 88. The **struct** component primarily creates an implicit root matching template which will contain the children of the **struct** root templates of the argument documents (the purpose of the **TP** child is described shortly).

²These are simplifications of the larger common brochure example.

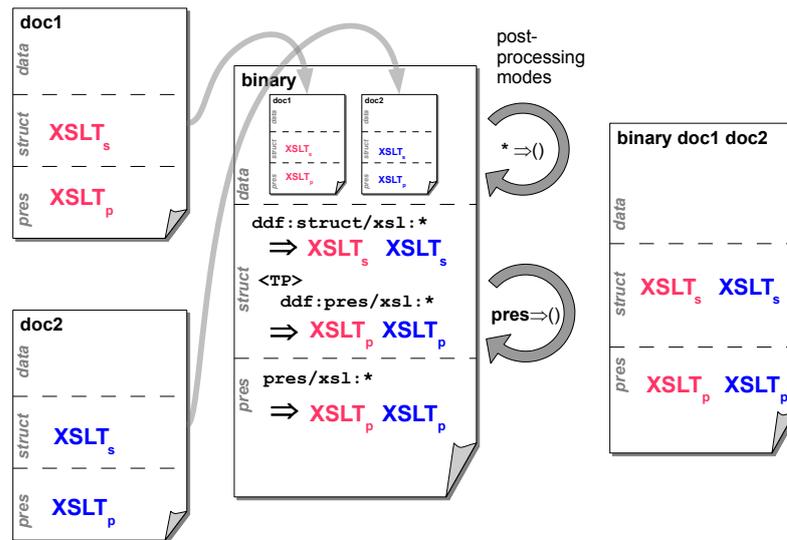


Figure 87. DDF 'higher-order' information flow

```

doc
  data
    match:* mode="ddf:data-data"
  struct
    match:/
    copy-of(//ddf:struct/xsl:template[@match='!']/*) retain="indirect"
    TP
    copy-of(//ddf:pres/xsl:template[@match='!']/*)
    match:TP mode="ddf:struct-struct"
  pres
    match:/
    svg
    pageSet
    copy-of(TP//svg:page)

```

■ ddf: ■ xsl: ■ svg:

Figure 88. DDF combinator document

If we wanted to differentiate between the two documents (perhaps reversing the order) then a slightly more complex XPath will suffice (e.g. **for \$d in (//doc2, //doc1) return \$d/ddf:struct/xsl:template[@match='!']/***).

The **pres** code in the argument documents has eventually to reach the **pres** section of the result – but the source for projection into presentation is in the **struct** section. Hence without modification to the DDF document, we have to both pick up new presentation code from the structure and then *remove it from the structure*. The **TP** structure is used as a placeholder into which the presentation root templates are stored temporarily. If we did not finally remove this structure then the two codes would interfere in the compound document. A post-processing

mode (**ddf:struct-struct** etc) needed for the continual document, can contain suitable removal code. In a related way we also need to remove the data documents from the data section, otherwise they would be considered data in any eventual evaluation – simple code in the **ddf:data-data** post-processing mode can do this too. The attribute declaration **ddf:retain="indirect"** ensures that the XSLT elements collected are retained in the final document output – this feature was required for continual document support.

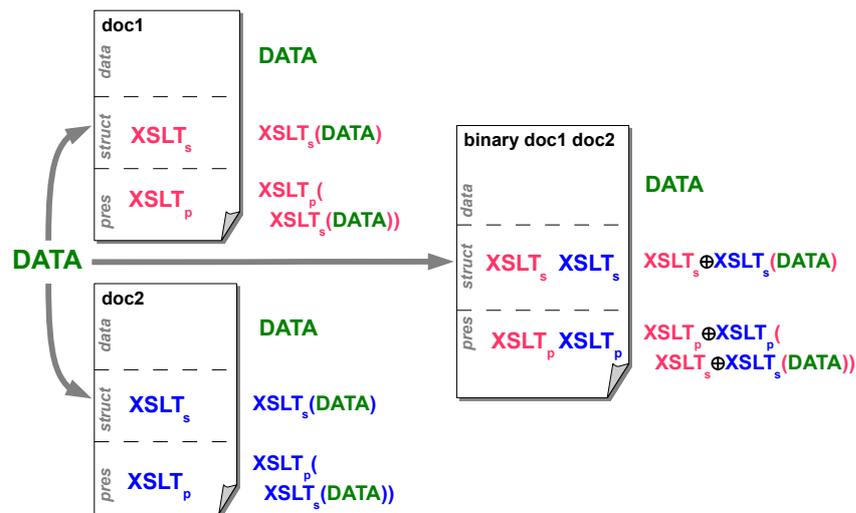


Figure 89. Document evaluations

What we now get is a document that will respond to data binding in similar ways to its constituents, as shown in Figure 89, where the \oplus symbol represents the effect of code concatenation. Figure 90 shows the result of projecting such a compound document on data.

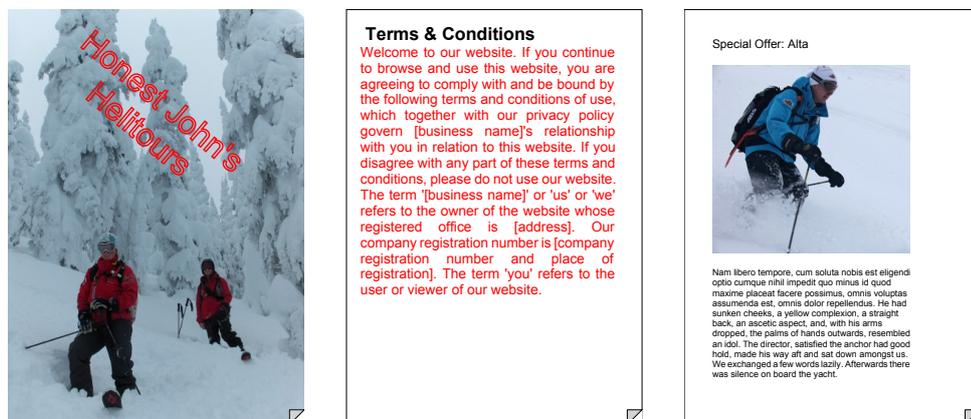


Figure 90. A combined document

This approach works for the current example, but is clumsy and will certainly break with more

complex document arguments. (The evaluation of ‘binary’ is confined to the compile-run phase within evaluation of a DDF document – no layout is performed – technically it should be subjected to the same treatment as processing layout invariance.) There are a number of issues:

- Structures introduced as programmatic tricks should be unnecessary for simple cases.
- Combinations are likely to be more complex than simple program concatenation, even though with ‘push-driven’ XSLT code (template matching), much of the code is order insensitive and ‘top-level’.
- Programs can *interfere* through clashes in naming³ or in priorities of push templates.
- Documents as arguments can carry referential contexts – item ‘pointers’ (e.g. images) or imported libraries. How should these be resolved or imported? What happens when they are completely common, e.g. two documents both reference the same library?

All these suggest that the DDF document will need some limited additional semantics to deal with these issues. Several of these are similar to those outlined in Chapter 9.

11.2 Higher-order syntax for DDF

Any syntax and semantics chosen should simplify the easy case we have encountered in our first example and build foundations for more complexity. This section will show the types of construct that will be needed, and that whilst ‘higher-order’ documents may need compiler-like abilities to analyse their argument documents fully, a small number of standard constructs and functions can help produce some reasonably simple forms.

Pulling presentation code through the structure is awkward and unnecessary. We could indicate that a particular code section in **pres**, or even the whole of that section, should treat **data** as its source space, rather than the default **struct**. A simple attribute annotation (**ddf:source="data"**) will suffice for now, where top-level program merging is involved. This requires a minor modification to the DDF compiler, to add a new possible capture mode (**ddf:data-pres**) and an additional interpolation of the result of this push-processing into the presentation result:

³Similar issues could arise in any FP system – modules or packages are one of the solutions, restricting clashes to package names.

```

<xsl:variable name="ddf:pres">
  <xsl:apply-templates select="$ddf:struct" mode="ddf:struct-pres"/>
  <xsl:apply-templates select="$ddf:data" mode="ddf:data-pres"/>
</xsl:variable>

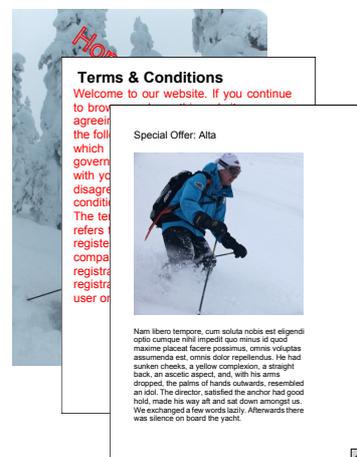
```

This assumes that any instructions to pull from data space will concatenate *after* those from structure space. In practice I suspect both source spaces will be used together rarely and, for such circumstances, ordering between the spaces is inconsequential. The post-processing ‘clearance’ of the data space could be indicated similarly by a simple declaration (**ddf:clear="*"**), coupled with minor additions to the compiler. A more general solution treats the clearance directive as an XPath pattern, so selective deletion is supported. Now the combinator document simplifies:

```

doc
  data clear="*"
  struct
    match:/
      copy-of(//ddf:struct/xsl:template[@match=']/**)
  pres
    match:/ source="data"
      copy-of(//ddf:pres/xsl:attribute-set)
  svg
    pageSet
      copy-of(//ddf:pres//svg:page)

```



■ ddf: ■ xsl: ■ svg:

Figure 91. Simplified DDF combinator & bound result

We now are implementing an information flow within the document that is a rather more complex graph (Figure 92). As well as the conventional routes **data** → **struct** → **pres** there are also two additional transfer modes, **input-data** that can be used for preprocessing, and of interest here, **data-pres**, which allows presentation elements to be made directly from data. In this case it is used to transfer XSLT programmatic fragments from the **pres** spaces of the input argument documents to the **pres** section of the result, without the need for assisting structure. In addition there are a set of ‘working’ and ‘result’ spaces which allow ‘after-read’ actions. In these examples they are used to remove data (such as the argument documents themselves) from the result after necessary information has been extracted. This is achieved by *not copying* controlled by **ddf:clean="pattern"** directives that the compiler understands⁴:

⁴Templates can also be written directly in these modes if desired, but compiler directives are more coherent.

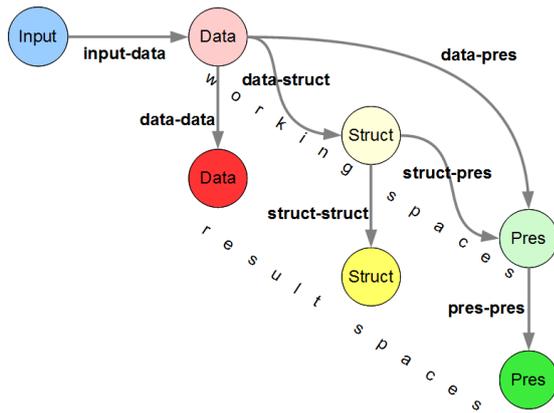


Figure 92. Information flows within 'higher-order'

11.3 Resource name conflicts

More interesting argument documents will contain global resources and features such as attribute sets and functions, which are accessed through embedded references within the tree. We can transmit such elements by conventional XSLT statements in the higher-order document:

```

doc
  data clear=""
  struct
    match:/
      copy-of(../ddf:struct/xsl:template[@match=']/]*)
  pres
    match:/ source="data"
      copy-of(../doc1//ddf:pres/xsl:attribute-set)
      copy-of(../doc2//ddf:pres/xsl:attribute-set)
  svg
    pageSet
      copy-of(../doc1//ddf:pres/svg:page)
      copy-of(../doc2//ddf:pres/svg:page)

```

■ ddf: ■ xsl: ■ svg:

Figure 93. Combination with (styling) attribute sets

The relationship between these resources and their exploitation points is not that of ancestor/descendant but through a referential name. When combining argument documents there may be clashes between resource names used in each. Suppose two documents both use a stylistic **attribute-set** called *para*, differently bound in each. Processing outlined so far would place both styles in the combined document. In XSLT multiple attribute sets *can* have the same name – the effect is that of merging their attributes. Thus our example evaluation would not break but also would not be as intended⁵:



a) Documents with same-name styles

b) Style clash in combination

Figure 94. Style conflict in document combination

To process this properly we need to i) identify name conflicts and ii) resolve them, presumably by renaming the set and its references in one, or both of them. The first can be tested through document-embedded statements (**any \$n in distinct-values(xsl:attribute-set/@name) satisfies count(xsl:attribute-set[@name=\$n]) gt 1**), but the second needs tree rewriting code, best added as a semantic feature. It is easier to implement blanket renaming of such internal references (including functions) under a directive request. This is not trivial – both name and reference must be altered: references can be buried in other forms of expressions⁶.

```

doc
import href="../../../HigherOrder/ho-library.ddf"
data clear="*"
struct
  match:/
  XSLT2:template /
    copy-of(../ddf:struct/(xsl:template[@match=']* | *[not(self::xsl:template)]))
pres
  match:/ source="data"
    copy-of(../doc1//ddf:pres//xsl:attribute-set) rename-prefix="doc1"
    copy-of(../doc2//ddf:pres//xsl:attribute-set) rename-prefix="doc2"
  XSLT2:template /
  svg
  pageSet
    copy-of(../doc1//ddf:pres//svg:page) rename-prefix="doc1"
    copy-of(../doc2//ddf:pres//svg:page) rename-prefix="doc2"

```

■ ddf: ■ xsl: ■ unknown: ■ svg:

Figure 95. Directed resource renaming

⁵Cases where one imported style overrides another can be satisfied by suitable XSLT/XPath programming.

⁶References to attribute sets are in whitespace-separated lists, functional references are in XPath expressions.

This works but the renaming has to be declared twice – once on the element that includes the attribute set, the other over the scope of application of those attributes. A more complete method might be to permit declarations of renaming spaces such as `<ddf:rename-scope path="//doc1" rename-prefix="doc1"/>` which could be compiled into suitable templates.

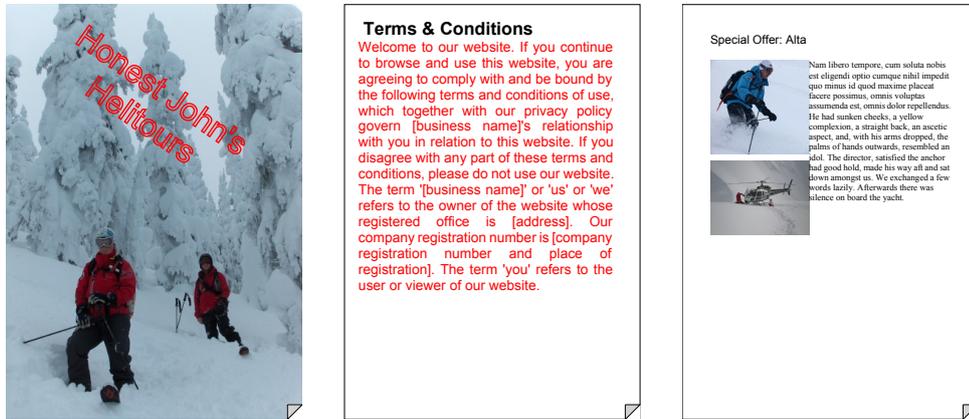


Figure 96. Combined document with style renaming

11.4 Compound documents

One of the powerful features of higher-order functions is the ability to build compound functions. How a binary page-level combinator document can be defined has already been shown. If this is correct we should be able to apply the process repeatedly somewhat like:

```
doc3 s = [Page ("a 3rd process on:"++s)
(twoDocs (twoDocs doc1 doc2) doc3) "variable text"
=>[Page {contents = "some process on:variable text"},
Page {contents = "another process on:variable text"},
Page {contents = "a 3rd process on:variable text"}]]
```

If we do that we get Figure 97.

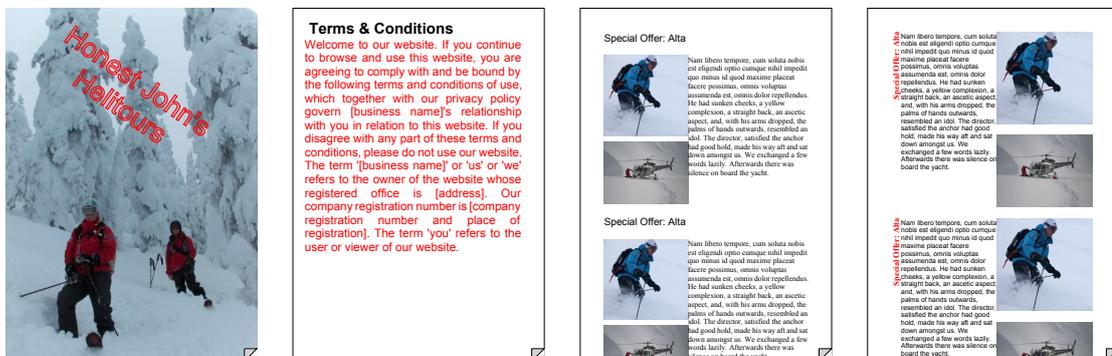


Figure 97. Compound application

Whilst the presentation is correct, there is a duplication of the resort. Two of the argument documents contain identical code to pull in the resort data item (`<xsl:copy-of select="//resort[@special-offer]" />`). In the combined document the resort-gathering code is included twice in the **struct** space:

```
struct
  match:/
    copy-of(../company../customer)
    copy-of(../resort[@special-offer])
    copy-of(../resort[@special-offer])
```

There is now duplication of ‘function’, whose resolution needs deeper knowledge of the document semantics. (Attribute set clashes were corrected by a strategy of global resource renaming.) We must first assume that duplication of structure is **not** required – this is reasonable but may be subject to declaration. We must then eliminate duplicate programmatic constructions in the two document program inclusion statements. But this has to be performed during execution of the higher-level action:

```
match:/
  XSLT2:template /
    copy-of(../ddf:struct/(xsl:template[@match='']/* | *[not(self::xsl:template)]))
```

Here we are amalgamating all the ‘top-level’ trees within **struct** space, or children of a ‘root’ template therein, to form a new root template. We need to eliminate *identical* trees from that collection. A suitable function (**ddf:distinct-trees()**) can be written and included in a ‘higher-order’ library document:

```
match:/
  XSLT2:template /
    copy-of(ddf:distinct-trees( ../ddf:struct/(xsl:template[@match='']/* | *[not(self::xsl:template)]))
  )
```

Invoking it removes the duplicates and produces the anticipated result:

For this type of simple document combinator we should expect associativity⁷, which we can check through testing that **binary(d1,binary(d2,d3)) = binary(binary(d1,d2),d3)**. This is indeed the case.

⁷This would not be so if the binary combinator added some additional components, such as a title page.

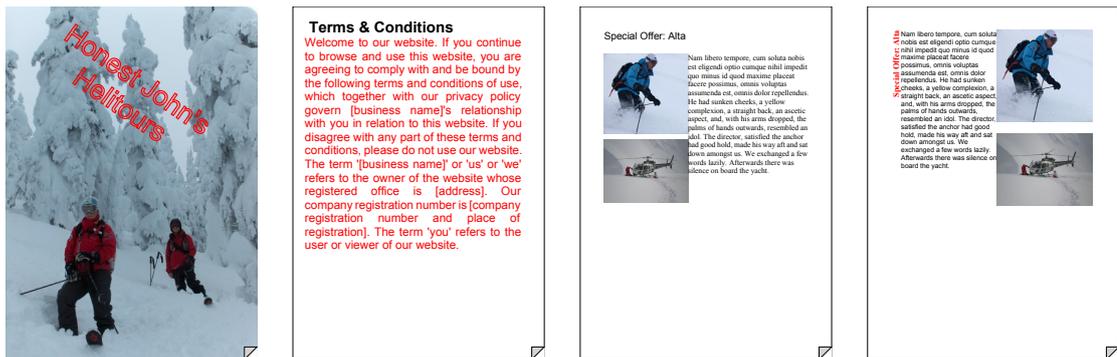


Figure 98. Compound application with duplication removed

Compound inclusion - document 'imposition'

Imposition is a common process in printing which places groups of pages onto larger sheets for eventual printing on a press. This process can include a number of other actions, such as adding guide marks for the printing process, superimposition of watermarks or similar and page rotation and order re-arrangement for eventual collation of the finished work by folding and cutting before binding.

Can a document be created which carries out this imposition for a sequence of variable documents, operating on the same data – i.e. **imposition(doc[])(data)**? The imposed document set is variable and can be evaluated in one action on bound data to produce the ready-for-print result.

The main issue is whether the set of argument documents each define a fixed or variable number of pages. If all are fixed, the approach is comparatively easy – complete combination can be carried out *before* any processing of the constituent documents, i.e. during the evaluation of the ‘imposition’ document and no ‘imposing’ code remains in the resulting unbound document. If some documents have a data-dependent number of pages then the imposition must be carried out during the evaluation of the imposed document set, sometimes even *after* layout of the constituent parts.

We first examine saddle-stitch binding for a set of fixed-page documents, often referred to as ‘booklet-making’. Pages must be in multiples of four, so the imposing code (Figure 99) adds extra blank pages as necessary. The set is then reordered to group into sheets, both front and reverse⁸. Each sheet is then composed by positioning pages. The final result when bound to some data is shown in Figure 100, where ‘deliberately blank’ has been added to those extra pages and page numbers have been imposed.

```

XSLT2:template /
  svg
  pageSet
    pages=
      ∇.//docs/* :
        copy-of(../ddf:pres//svg:page) rename-prefix="{name(.)}"
    pages4=
      $pages/*
      n=4 * ((count($pages/*)+3) idiv 4)
      ∇(count($pages/*)+1) to $n :
        page width="210" height="297"
        group:$pages4/* by:min(( position() - 1) idiv 2, (last()-position()) idiv 2 )
        group:ddf:rotateRight(current-group()) by:(position() - 1) idiv 2
        page width="420" height="297"
        svg x="0" y="0"
          current-group()[1]/(@width,@height,*)
        svg x="210" y="0"
          current-group()[2]/(@width,@height,*)
        line x1="210" y1="0" x2="210" y2="297" stroke="black" stroke-width="1"

```

■ unknown: ■ svg: ■ xsl: ■ ddf:

Figure 99. Imposition code

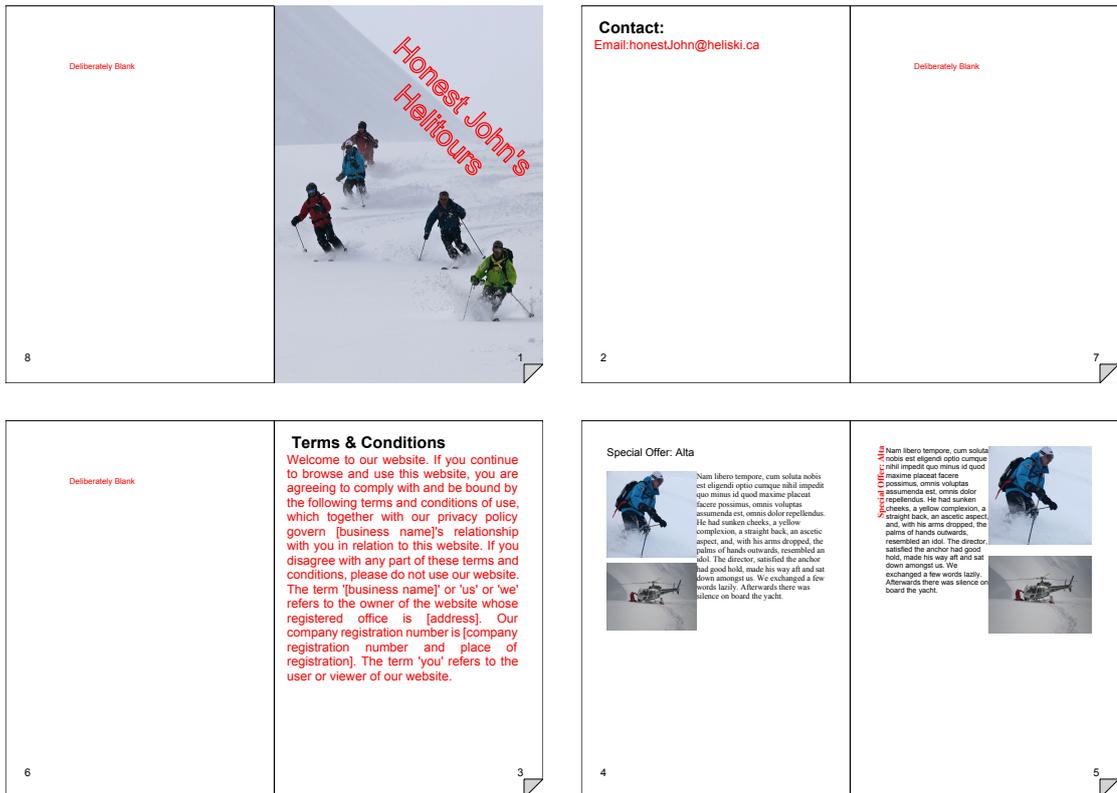


Figure 100. Imposition for saddle-stitch binding with fixed-page argument documents

⁸The integer function $\min((n-1)/2, (last-n)/2)$, when $last \bmod 8 = 0$, yields a value which is the same for all pages on the same sheet and is used as the grouping discriminant. `ddf:rotateRight()` ‘barrel shifts’ a sequence of items on place right.

When the number of pages is variable the code in the imposition document becomes more complex. Firstly we must recognise situations that *generate* a variable number of pages: there are two broad kinds. An XSLT construct can produce a data-dependent sequence of pages (e.g. `<xsl:for-each select="resort"><svg:page>...`). To find such cases we must examine the XSLT ancestry of code pages, or in the worst cases examine the execution flow. A layout function can also produce pages – most commonly a paginator. The imposing document must have sufficient knowledge of layout syntax to identify candidate constructs.

Having found variable page generators we must place them (and any static pages in proper sequence position) under a suitable imposition computation. When the variability is a consequence of layout (e.g. pagination) then that imposition must be performed *late in the layout phase*. If however all variability is caused by XSLT, then the imposition can be determined before final layout phase. Using push mode templates, iterators such as match the pattern `xsl:for-each[svg:page]` can be copied in *in toto*, whereas other isolated `svg:page` elements can be copied in sequence. Figure 101 show two different data bindings which result in a variable number of pages in an 8-up imposition (which requires page rotations) – here the argument documents being imposed together all have page variability defined in XSLT.

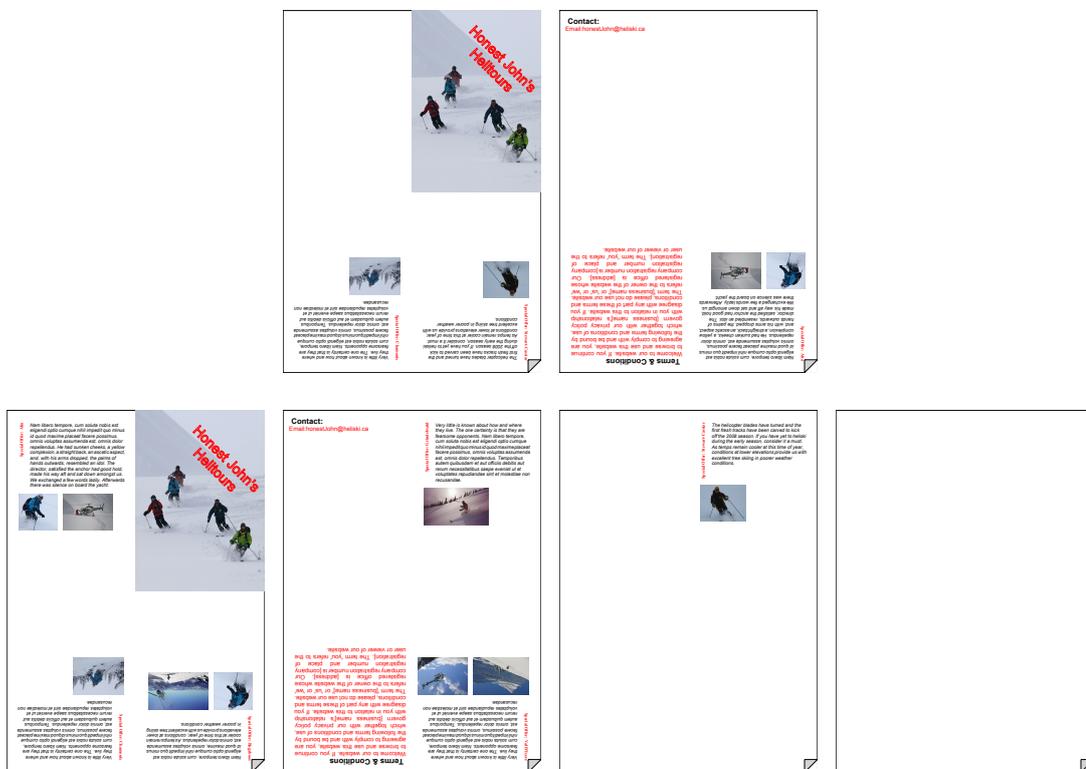


Figure 101. An 8-up imposition with variable-page documents

11.5 Conclusion

The examples in this chapter have been relatively simple higher-order tasks, focussed on page combination of the results of projection of argument documents. But there has to be some careful programming, along with specialist support functions and directives that the compiler will exploit. More complex examples could include a document that consumes another variable document to produce a new version that will generate an outline or some form of précis of the content that the argument document would produce.

But it should be clear that these ‘higher-order’ documents need either i) to have an intimate knowledge of the ‘type’ of the documents they are consuming or ii) relatively simple canonical forms for those argument documents. In the examples shown the canonical form, for presentation, is the **svg:page** and the semantics of processing structural and presentational mappings are at best concatenation, with some appropriate measures to avoid clashing of named resources and removal of duplicated function.

The implementation of ‘higher-order’ semantics being explored here was for the higher-order document to consume sections of the argument documents through its embedded XSLT and generate new sections of potentially modified, and perhaps partially evaluated, XSLT in the result. For any usefully complex argument documents this becomes principally a programming exercise, *not* a simple property of some typical document component. The only other approach is strictly to hold ‘encapsulated’ documents within the consumer for later evaluation and extraction of results – as such a good description of work-flow may be more suitable.

The conclusion from this chapter is that documents of the complexity of DDF consuming active variable documents in a higher-order sense are possible in some limited circumstances, but are not especially practical. Writing specific XSLT programs for the small number of cases anticipated (such as imposition) will be much simpler for the document engineer. Higher-order behaviour in the sense of producing variable document *results*, as discussed in Chapter 9, is much more promising as we shall see in the example of Chapter 12.

Chapter 12

Example Document - a Medical Record

To illustrate a ‘highly functional’ document this chapter presents a larger example of a simple medical record, treating it as a ‘continual’ document: one that may continue to accept data, grow and change as new information is added progressively, but for which *it is meaningful, and useful, to observe its presentation content* at stages during its binding ‘life’. The general idea was discussed in a previous chapter. The medical source data and the record's appearance at several stages of binding are discussed first, followed by details of the implementation in terms of the document design document and the use of various of the ‘higher-order’ features described earlier. A number of different design/code paradigms are illustrated that demonstrate the utility of some of these features.

The example is a multi-part, multiple page medical record for a single patient¹. As new data is added, from a variety of sources, content is added at several parts of the document. For example a new temperature reading may add to a chart *and* a table. Laboratory tests might add both a complete new page and an additional entry in an earlier summary table, as well as to the all-important accounting page. The overall design for the document is:

¹The same basic example, using a more *ad hoc* approach, based *only* on additive operations, was presented at DocEng07 though the paper used a simpler diary example[64].

- A title/summary page, containing simple patient details coupled with a summary for each day in the subsequent record.
- A page containing two data/time graphs (for blood pressure and temperature). New samples will be added to this graph as they are bound.
- A page for each completed day, containing relevant information from tests and such like.
- A page detailing patient account charges.

Some parts of these pages will have additional material added, modified or removed at various stages in the document's life.

12.1 Data and the document life

The variable data to be bound is a simple XML structure (**medical-record**) containing several **days** with simple descriptions as shown in Figure 102.

```

<medical-record patient="Solomon Grundy">
  <patient>
    <name>Solomon Grundy</name>
    <date-of-birth>2011-09-11</date-of-birth>
    <next-of-kin>Tom Thumb</next-of-kin>
    <finance>
      <credit-card>12345678901234</credit-card>
    </finance>
    <admitted>2011-09-12</admitted>
  </patient>
  <contd/>
</medical-record>

<medical-record patient="Solomon Grundy">
  <day date="2011-09-12" day-no="0">
    <reading type="temperature" time="06:00:00" charge="23.45">36.4</reading>
    <reading type="temperature" time="08:00:00">36.9</reading>
    <reading type="bloodP" time="08:00:00">90/65</reading>
    <test type="reflex">Reflexes normal</test>
    <test type="catscan" image="catscan8cg.jpg" charge="123.45">Cross-section of his skull, showing a brain present</test>
    <notes>The patient appeared grumpy and threw his mobile phone at the consultant</notes>
  </day>
  <contd/>
</medical-record>

```

Figure 102. Sample patient details and medical data

There are a number of main design decisions. Firstly we assume that there is a positive marker for ‘continuation’ in the data² – in the presence of such a marker the document will be in a form such that new data can continue to be bound to it. In its absence, the document ‘finalises’. The specific marker chosen is **contd**, and is assumed to be the last child of a new data binding. (Later on the reverse process, i.e. continue until told to stop, is explored.)

Here is the document at each of four stages of progressive data binding. (A larger version of the final document can be found in Figures 116 and 117 of Appendix A.)

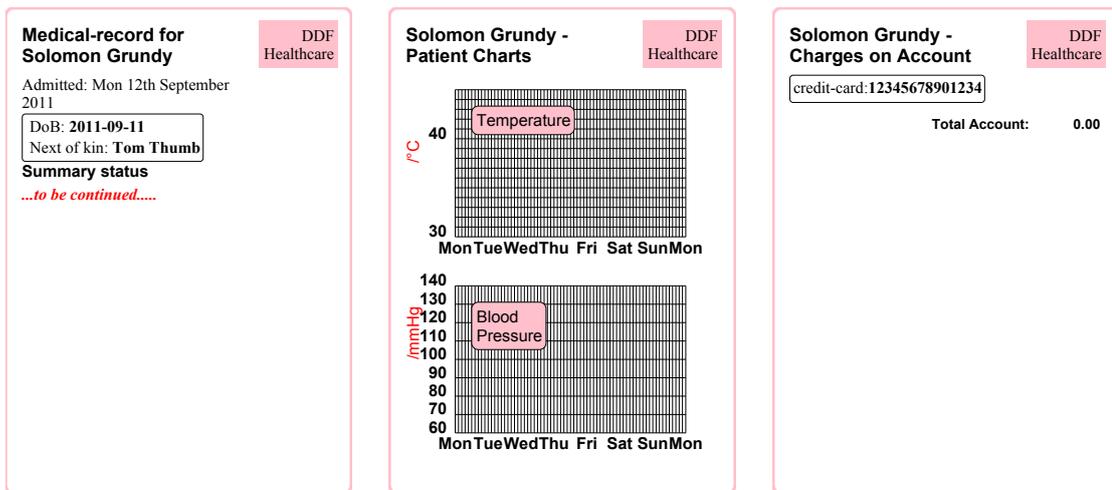


Figure 103. Medical record after binding patient details

The very first binding, in Figure 103, establishes the patient details and as the input was marked to be continued, indicative markers appear in the appropriate positions in the output as well as programmatic continuing instructions in four places³ for the next binding. After the second binding we have content for Monday and Tuesday appearing in all four sections with a presentation shown in Figure 104.

²Not the continuation of functional programming, but it is a suitable word for this context.

³At the foot of the summary, in the graphs, between ‘day’ pages and just before the account total on the charges page. A visible marker appears in the summary where the continuation point is displayed as an indicator.

Medical-record for Solomon Grundy DDF Healthcare

Admitted: Mon 12th September 2011

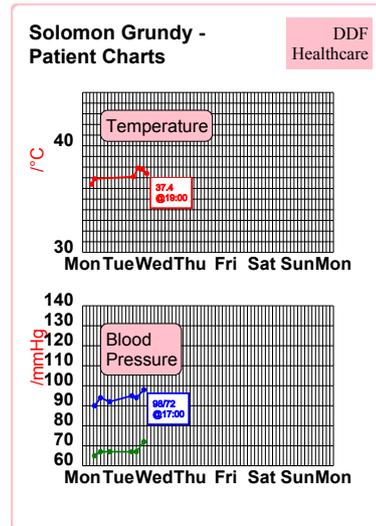
DoB: 2011-09-11
Next of kin: **Tom Thumb**

Summary status

Mon: 36.9°C
36.4°C The patient appeared grumpy and threw his mobile phone at the consultant

Tue: 37.9°C
37.1°C The patient was interviewed by the local constabulary

...to be continued.....



Patient notes for Mon 12th DDF Healthcare

Tests & Reports

- 36 I
- 36 I
- 90 BP
- 94 BP
- 92 BP
- Re I
- 95/67 @ 9:00
- 94/67 @ 12:00
- 98/72 @ 17:00

Gener: Reflexes normal

The patient's mobile

General Notes

The patient was interviewed by the local constabulary

Solomon Grundy - Charges on Account DDF Healthcare

credit-card:12345678901234

Mon: reading temperature @ 6:00	23.45
12th test catscan	123.45
Tue: reading temperature @ 10:00	10.00
13th reading temperature @ 13:00	10.00
reading temperature @ 16:00	10.00
reading temperature @ 19:00	10.00
Total Account:	186.90

Figure 104. Pages of the medical record after binding two days' data

We can now proceed to bind data for the second part of the week: these retained and embedded program sections now operate, generating material for Wednesday, Thursday and Friday (but *not* Monday and Tuesday as that presentation already exists in the document at this stage). Since we claimed the week still continued in the new data this processing again leaves additional continuation program points, as shown in Figure 105.

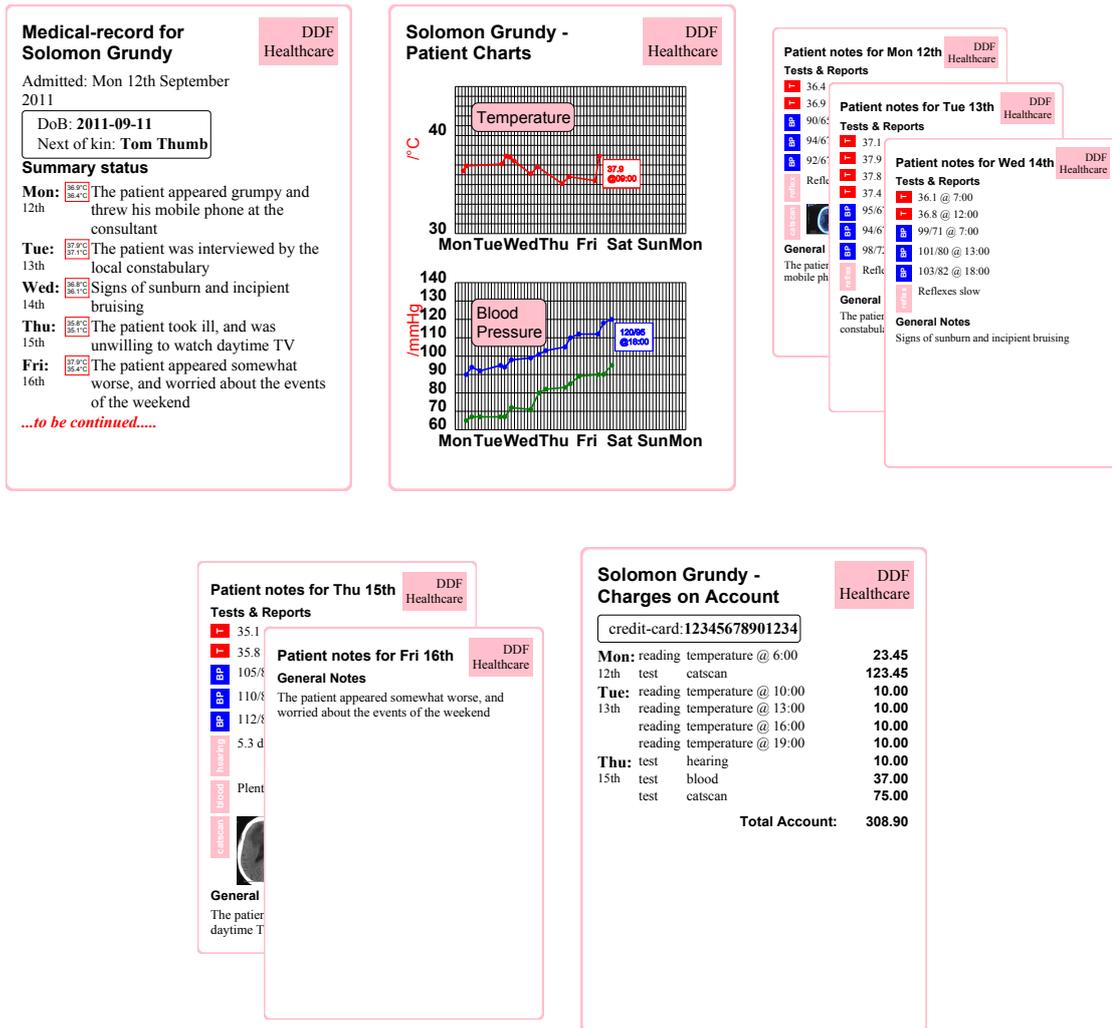


Figure 105. Record pages after further binding

The third stage completes it with the weekend's data (this time without a continuation marker), giving a final result shown in Figure 106. This has no remaining program and will not alter with the addition of any further data.

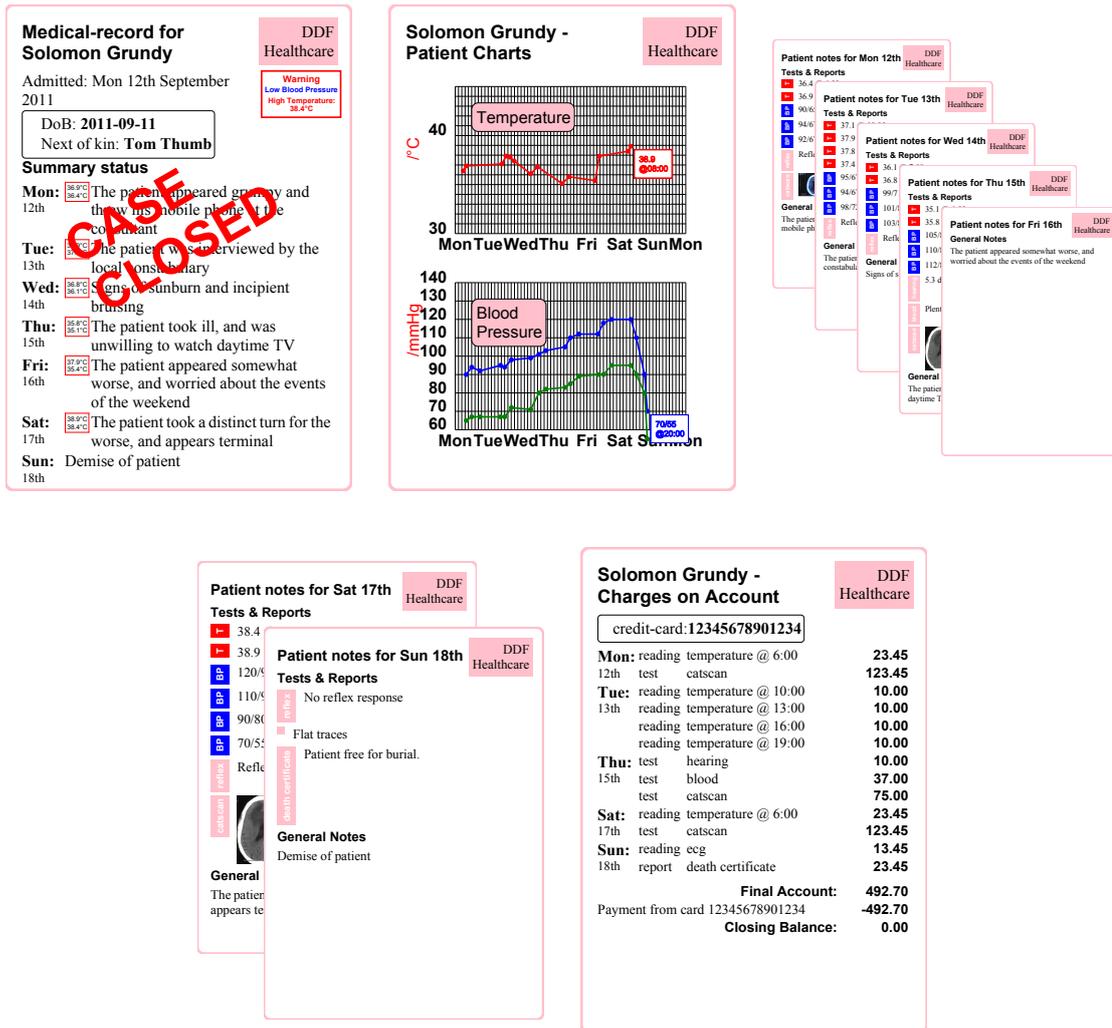


Figure 106. Record pages after final binding

In a sense this document is performing in a manner somewhat similar to being bound into a `scan1()` higher-level function:

```
scan1 (+) [1,2,3,4,5]
=>[1,3,6,10,15]
```

However the result of each stage should be a *function* that should be applied to the 'next' argument. Expressing it accurately in Haskell may not be possible, needing a recursive functional type i.e. **Type FN = (a → FN)**.

12.2 Implementation

The features and constructs described in Chapter 9 are used heavily. To start, appropriate pro-

cessing of the input data is arranged. Firstly the document accumulates all the original data for the record so, at some later stage, information bound earlier in the record's life can be interpolated. However doing this requires some mechanism to identify *new* data that has just appeared in the current binding. If not then repeated sections are likely to appear in the presentation. The method chosen is to inject a marker **new**⁴ in the **ddf:data** section in front of the data from the most recent binding. A predicate [**preceding::new**] will then be true *only* for all new information.

Secondly some embedded XSLT fragments must be retained after execution when there is a **contd** marker in the data. Since this condition may be used often, the test is bound to a variable **continueData = exists(*|contd|contd)** that will be interpolated. To indicate the retention the **@ddf:retain="continue-while" ddf:test="\$continueData"** construct is attached to such pieces – the effect is for them to execute and then, provided the test is satisfied, remain embedded in the same form for a subsequent execution.

These ancillary markers (**new,contd**) appear in the data for purposes of being read for the presentational XSLT projection but they need to be removed before the next data binding, otherwise old data will be wrongly read as new, or continuation will be misconstrued. These can be removed mostly cleanly in a data-finalising pass controlled by a directive pattern **@ddf:clear="new|contd"** attached to the data section.

Figure 107 shows the result of the first binding of data (the patient details), which contains the two markers (before their subsequent removal) and the retained sections of XSLT code. The first is the binding to the **continueData** variable, which will eventually yield false when **contd** no longer appears; the second is the retained interpolator which adds new data, preceded by the **new** marker, and ‘self-propagates’ the interpolation. This approach can be a general paradigm for ‘marked continued’ data streams bound part by part.

Now the data is appearing in the correct fashion, attention can turn to generating the presentation. Usually a DDF document would employ the **ddf:struct** intermediate space to build a logical (and canonical) structure – often for purposes of reuse. In this case it is ignored – it complicates the explanation and the exploratory programming and there is little to be gained in this simple example from using such a canonical layer. By marking the **ddf:pres** section

⁴Which of course should *not* occur in the data – a namespaced marker is perhaps more robust. **contd** is an agreed marker within the context of the application.

with the directive `@ddf:source="ddf:data"` presentation will be generated directly from variability in the data.

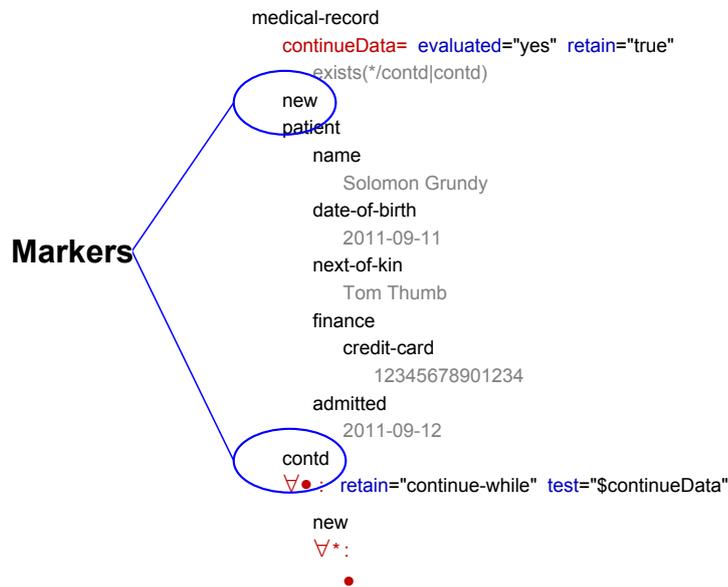


Figure 107. Record with contained data & implementation markers

The presentation generator is written as a mixture of pull and push mode operations. Some graphic entities are ‘self-contained’, such as the ‘day pages’, and producing these and other parts defined completely by simple small subtrees (e.g. **notes**) are most easily done with XSLT templates. To retain these templates in the output for use in later bindings they can be marked `@ddf:retain="true"` – the compiler will arrange that a copy appears in the output.

The major structure is most built in ‘pull’ mode – a set of **svg:page** parts each containing reference to a common background (the presentational variable **\$background**) and some layout functions (mostly flows) for grouping parts. Within the layouts of these pages are interspersed sections of XSLT that will generate additional or replacement material. The new ‘per-day’ pages are inserted by a simple template-invoking operation:

`⇒ medical-record/day[preceding::new] mode="day" retain="continue-while" test="$continue"`

This statement is decorated with a ‘remain’ condition that is true when the medical record continues (**\$continue**). Once that no longer appears this invocation will be removed thereafter and at no further binding will new pages appear. Within the cover page the per-page material is similarly placed within a vertical flow, using a similar embedded invocation but this time in **summary** mode, with a corresponding and retained template.

On this page there are two special additional parts – warnings and a closed indicator⁵. The ‘CASE CLOSED’ banner that appears in the final instance is an **xsl:if** section at the end of the page whose continued existence is guarded by the **until-triggered** condition on a test of **not(\$continue)**. Once the case is closed the banner appears and as this surrounding XSLT disappears, the case can never subsequently be ‘unclosed’!

The warning section in the top RH corner appears only when a condition is detected in the new data – thus it needs to be able to switch on and off. The complicated arrangement of altering the layout function (between **hideSVG**, **lastSVG** and **removeSVG** depending on visibility condition and the invariance of the graphics), shown in Figure 75 in section 9.8, is used to do this, exploiting the **ddf:revealSVG** compound instruction to arrange them correctly. This can be nested as here, so we generate the warning graphics as follows:

```

readings=medical-record/day[preceding::new]/reading retain="continue-while" test="$continue"
svg layout="flow" x="155" y="40" id="warning"
  revealSVG test="med:warn($readings,'bloodP') | med:warn($readings,'temperature') "
    svg layout="flow" padding="1" encapsulate="border-color:red;border-style:solid;border-
width:1"
      block fill="red" font-size="16pt" use-attribute-sets="warning"
        Warning
      revealSVG test="med:warn($readings,'bloodP') "
        block fill="blue" use-attribute-sets="warning"
          Low Blood Pressure
      revealSVG test="med:warn($readings,'temperature') "
        block fill="red" use-attribute-sets="warning"
          High Temperature:
          val(med:warn($readings,'temperature') [1])
          °C

```

■ xsl: ■ ddf: ■ svg: ■ ddf: ■ fo: text

Figure 108. Conditionally revealed warnings

The outermost is guarded by either warning condition (blood pressure low, temperature high – the function **med:warn()** checks the records returning any readings that are ‘of concern’) and generates the group with warning label when required. The first of the inner warnings has static content so merely hides or reveals. The second needs to generate variable content (one of the errant temperatures) so has to perform replacement – this requirement is detected automatically by the compiler in expanding **ddf:revealSVG**.

⁵The ‘to be continued markers’ shown in the documents are generated by similar methods, but are not central to the document design – they are merely conveniences for the reader.

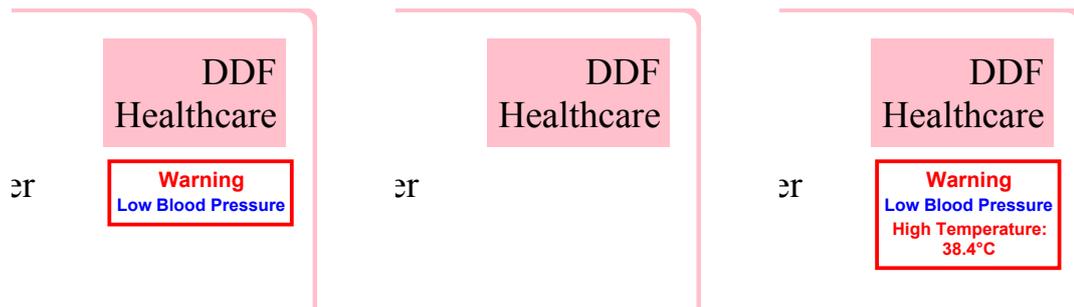


Figure 109. Conditionally altered graphics over three stages of binding

The page showing the blood pressure and temperature traces uses a graph-generator that after initial invocation will contain a section of retained XSLT that continues to plot the additional data points for the given trace. There are two possible means of construction. Firstly we could design a specialist layout agent (**graph**) that mapped $data[],XSLT \rightarrow graphics$, but a more generic method uses an XSLT generator $data[],XSLT \rightarrow DDFL,XSLT$. This is employed here: a named XSLT template generates background, any extant data points and appropriate embedded program to continue drawing the traces.

This template has two sorts of parameters: the usual graph properties (scales, axes, colour, caption etc.) and four XPath parameters:

- i) When to continue, to be used for retention tests. For this case we use the same pattern **exists(*/*contd|contd)** that was used to calculate **\$continue**.
- ii) A path to collect new data readings (**@readingsPath**), e.g. **medical-record/day[preceding::new]/reading[@type='bloodP']**.
- iii) A path that will generate an X and Y pair for a reading (**@xyPath**). This is executed relative to each reading and is expected to produce a sequence pair of **xs:double**, where the units are the graph units – scaling into the eventual scaled SVG co-ordinates is handled by an affine transform precomputed from the graph properties. In this case the X part of this computes a number of days by finding the duration between the original patient admission and the parent day for the reading (**days-from-duration(xs:date(../*@date) - xs:date(preceding::patient/admitted))**) and then adding time as a fractional day.
- iv) An optional path that will produce a sequence of lines of text for the final ‘point label’, e.g. **string(.),concat('@',replace(@time,':\d\d\$', ''))**

The last three XPaths are associated with an individual line trace, so multiple lines can be drawn,

as is the case for blood pressure where systolic and diastolic pressures (conventionally written 120/92) are plotted separately.

The graph backgrounds (axes, labels, grids) are written once in the initial binding, leaving an XSLT group embedded in each of the graphs, that will act as a self-repeating generator for the data points and line. Figure 112 shows diagrammatically what is happening in the SVG tree for the temperature graph after each of the bindings. There are no readings for the first so the generator remains dormant. After the third binding there are 12 readings (the generator has produced 12 line-point pairs) and since there is possibility of continuation the generator remains *in situ*. After the final binding this disappears along with most of the other XSLT constructors.

At top level, within the SVG tree containing the graph and just inside each of the graph line groups, are a small number of XSLT variables that have been initially evaluated and retained *as values* as shown in Figure 110.

```

continueXPath=$continueX eval="true"
(xs:double*) transform=$h-scale,0,0,-$v-scale,0, $g-height + ($v-scale * $min-y) eval="true" retain="
continue-while" test="$continue"
readingPath=@readingPath eval="true" retain="continue-while" test="$continue"
xyPath=@xyPath eval="true" retain="continue-while" test="$continue"
labelPath=@labelPath eval="true" retain="continue-while" test="$continue"

```

a) Variables before evaluation

```

continueXPath= evaluated="yes" retain="true"
exists(*|contd|contd)
(xs:double*) transform=20,0,0,-6,0,270 evaluated="yes" retain="continue-while" test="$continue"
readingPath= evaluated="yes" retain="continue-while" test="$continue"
medical-record/day[preceding::new]/reading[@type='temperature']
xyPath= evaluated="yes" retain="continue-while" test="$continue"
days-from-duration(xs:date(..@date) - xs:date(preceding::patient/admitted)) + (hours-from-time(xs:
time(@time)) div 24),.
labelPath= evaluated="yes" retain="continue-while" test="$continue"
string(.,concat('@', replace(@time,':', '\\d\\d$', '')))

```

b) 'Fixed' values

Figure 110. Embedded before and after evaluated variables

The outer two are the continuation XPath expression and an affine transform whose value is required in later stages to correctly position the points depending upon the initial ranges and size of the graph requested at the first document evaluation – the transform is calculated dur-

ing that evaluation. The other three are the data-gathering XPath expressions described above and bound within the relevant trace. It could be possible to interpolate these values as scalars into XPath expressions that use them, but this requires complete analysis and partial evaluation of such expressions – other parts of such expressions are the data values themselves and therefore not bound to a single value. This expediency of marking values that are needed, and will be evaluated and retained as static bindings, overcomes this problem. As they are ‘foreign’ to SVG they sit happily within the trees.

Since graphs with continuous lines are preferred, i.e. the first of the next set of readings connects to the last of the previous, we need some way of recording what the last point was. Recall from Chapter 9 that XSLT code cannot access the (result) tree within which it operates – this rules out searching for the previous **svg:circle**. However an XSLT variable with a known name (e.g. **\$last**) could be read within the XSLT execution. A suitable mechanism is:

- Initialise an empty **xsl:variable name="last"** in front of the first constructor.
- Read the value of **\$last** and if it is non-null, use this value as the beginning of the first ‘joining line’.
- Record the last reading (x,y) as the sequence value of an **xsl:variable name="last"** that will be written into the final output (using the automatically ‘warped’ **XSLT:variable** mechanism.) This will then be a preceding sibling of the next graph line constructor. It will disappear after interpolation during the next subsequent binding and evaluation (i.e. after **\$last** is read as described in the previous bullet point).

This method works as can be seen in the figures (it works within tree nesting scope hence each of the two traces in the blood pressure graph are ‘separate’) and can be used with appropriate care as a paradigm to record, within the tree, some graphically-related information that will change at each iteration, such as tracking a maximum value.

The final accounts page, whilst perhaps a little unrealistic (would auditors sign off on accounts being generated entirely in documents?) is quite straightforward – new account charges are added on a daily basis, and the account total is recalculated from the entirety of the data by code shown in Figure 111. (This is a case where retaining all the data is helpful.)

```

layout(lastSVG) role="total"
  √medical-record : retain="continue-while" test="$continue"
  total-account=sum(day/*/@charge)
  layout(flow) x="0" y="0" direction="x" spacing="4" padding-top="5"
    block use-attribute-sets="charge-amount" width="144"
      val(if($continue) then 'Total' else 'Final' )
      Account:
    block use-attribute-sets="charge-amount" width="40"
      val(format-number($total-account, '#0.00' )

■ ddf: ■ xsl: ■ ddf: ■ fo: text

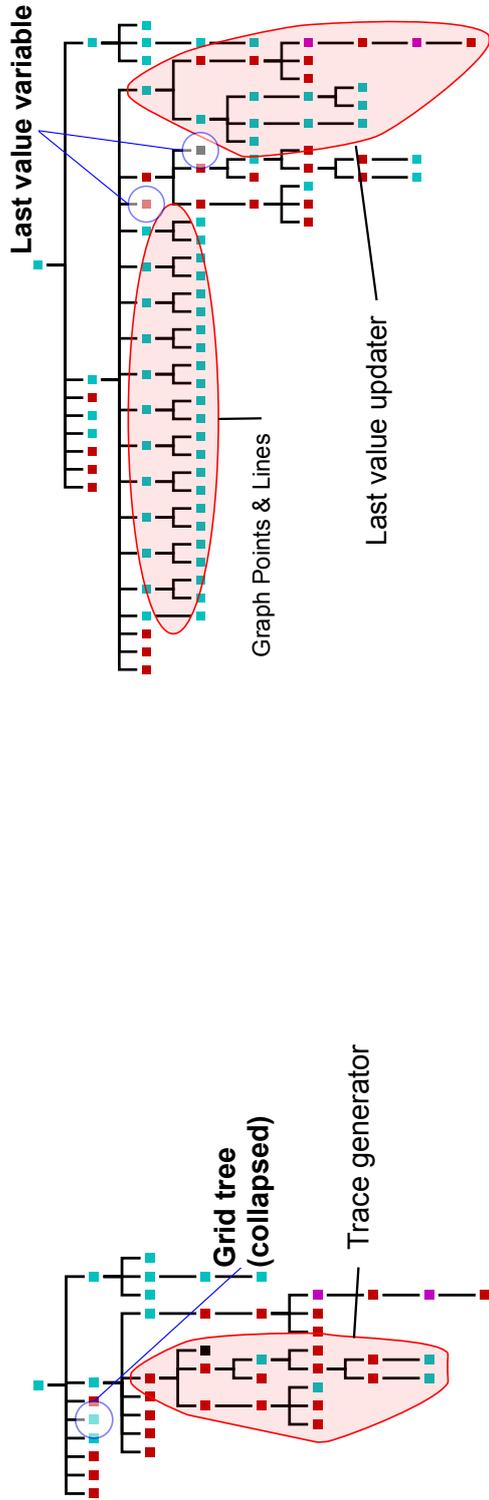
```

Figure 111. Constructing the accounts charges page

At the start of this chapter a design decision was made that the document would operate in a ‘positive continuation’ manner – unless a specific condition is satisfied in the input data (using **contd** as the marker) the document will no longer be ‘active’. But the inverse might also be possible – continue unless told to stop (‘**case-closed**’). Figure 113 shows the document in reduced tree form, with all the (10) points that contain reference to continuation (**\$continue**) marked, as well as the defining points for the variable⁶.

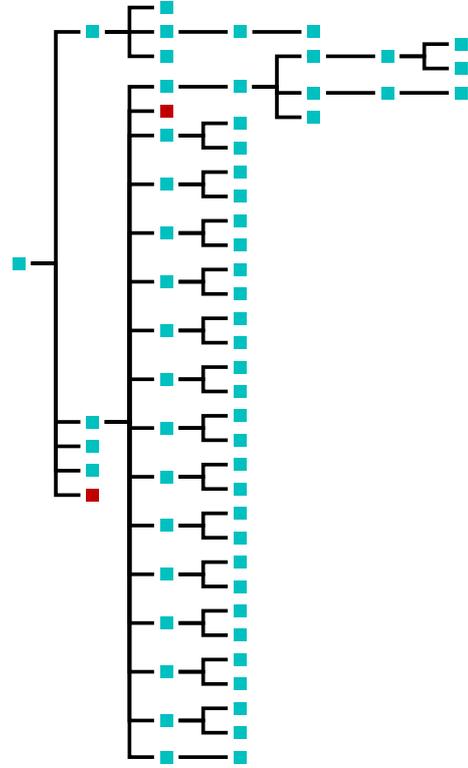
Is it possible to merely negate these conditions to get the effect we wish? If the condition **exists** (***/contd|contd**) is changed to **empty**(***/case-closed|case-closed**) and a different set of data with such positive closure is presented, then indeed it does perform as expected – the ‘continuation condition pattern’ could be a parameter of the document.

⁶The graph generator was designed such that it takes the continuation condition as a parameter, so the only reference to this condition is in the main document.



a) Bound to patient

b) Bound to Mon-Tues & Wed-Fri



c) Bound to entire week, no continuation

Figure 112. Graph generator at three stages during progressive binding

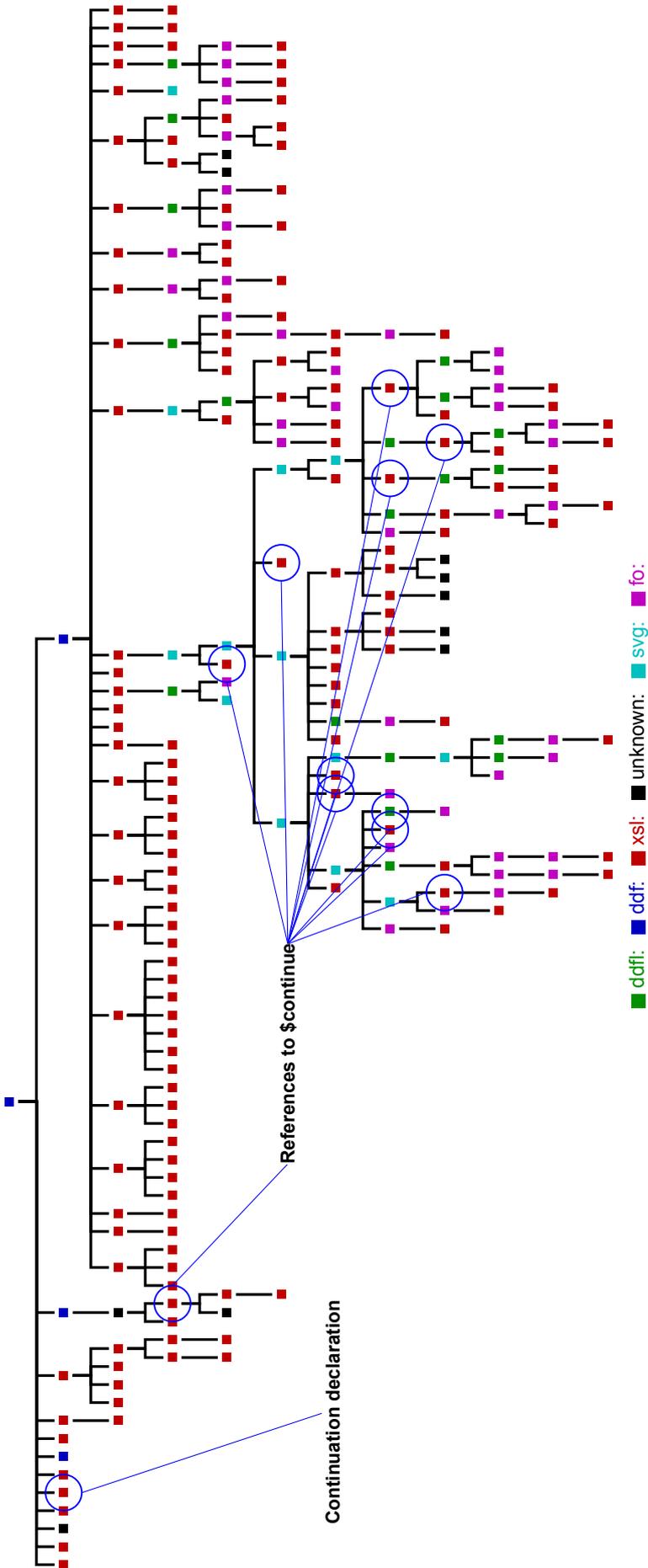


Figure 113. Continuation points within the document

12.3 Conclusion

This example has shown that it is *possible* to define and process certain types of continually active document within DDF with a coherent model of interspersed sections of SVG, XSLT and DDFL layout declarations. By contrast with the DocEng07 paper [64], which worked by operating in ‘push mode’ and overlaying graphics on top in some places, this example uses ‘pull mode’ semantics extensively, and adequately supports deletion and replacement of graphical content from local positions *within the tree*.

Most of the techniques described in Chapter 9 have been employed, albeit in some rather complex arrangements. Some areas, such as arranging suitable retention of program components and arranging for superfluous program sections to be deleted when no longer required are somewhat fragile and would need a more systematic analysis to be uniformly practical.

The techniques outlined could be used as paradigm and code-generation tactics for smarter document compilers. Some of these possibilities are discussed briefly in Chapter 13. This example re-emphasises the four key findings:

- SVG trees can be decorated with attributive declarations to describe relationships between subparts of the trees, leading to a degree of *idempotency* in layout.
- Elements and attributes in foreign namespaces can exist in SVG trees and execute to support extended variable graphical behaviour *provided* that all tools act as *good XML citizens*.
- XSLT program fragments, being entirely described in XML, can exist within and be manipulated along with other XML structures.
- Constructs for embedded XSLT behaviour in SVG can be generated and processed consistently within such a framework and support extended functional behaviour with a hybrid system of a small set of specialist *meta-layout* constructs and the use of a transformational compiler to modify the embedded XSLT according to appropriate directives.

Chapter 13

Discussion and Conclusion

To conclude, we summarise the work reported thus far in the thesis, discuss some general features of the framework and its use, re-iterate the key findings of the research, outline ideas for any redesign of the architecture, suggest further technical work that might be of significant interest and present lessons, both positive and negative, learned within the entire DDF story.

The previous chapters have described the context of variable data documents, relevant prior art in document engineering and an experimental XML-based architecture (DDF) for defining flexible and extensible forms of such documents. The basis of DDF as a single-document combination of XSLT, SVG and a declarative description of hierarchical layout intent, together with partitions for holding data bindings and logical structural models, has been explored and illustrated in some detail. Implementation techniques in terms of document compilation and an extensible pattern-driven layout processor, almost exclusively using XSLT technology, have been explained and their effectiveness demonstrated through example and the production of this thesis itself, using the very technologies that it describes.

Much of Part B is equally relevant to static documents, but the thesis title “Documents as Functions” being re-used for Part C reflects the research focus of this work: exploring possibilities that such an architecture brings to variable data documents by treating them more overtly

as *functions*. Such use would either be as contributory parts to the construction of other variable documents, or as documents that can be partially, or repeatedly, bound to data. Most of the focus has been on the latter point – partial binding and eager evaluation of documents – either because only partial data is available, but a visual result is still required, or to optimise latency in final document production by pre-evaluation of invariant and repeating details.

Chapter 9 showed how the original design of DDF in a wholly-XML format, together with *good XML citizen* behaviours in processing tools (some of which manipulate programmatic elements embedded in DDF documents), could accommodate mechanisms to support idempotency and resatisfaction in document layout, retention and modification of data-sensitive programmatic generators, and in-tree modification of presentation.

These mechanisms were used in section 10.1 to describe a document that could be evaluated by parts over its anticipated data, with a complete visible presentation at each binding. Finding invariant sections within the layout of DDF documents was examined in section 10.2. Both of these are relevant to optimisation of large-scale variable data document production by employing re-use architectures, such as PPML's *reusable_objects*, where pre-computed and pre-rasterised results can be incorporated, directly within the press, as parts of documents with other variable components generated on a per-instance basis.

Building variable documents that construct more complex or larger variable documents by ‘higher-order’ techniques was examined in Chapter 11, specifically looking at how some simple ‘function as argument’ ideas from functional programming might be implemented using DDF documents and the techniques presented in Part C. The basic finding was that whilst some page-level combinations were possible, the documents involved were too complex for this approach to be preferable to purpose-built document-transforming programs.

That the techniques are promising is shown in Chapter 12, which builds a self-contained continually-active variable document, which modifies its presentation in a controlled and robust manner as data is bound to it, in a possibly infinite sequence of steps. This is an example of a *highly flexible* variable data document, which current commercial tools would be incapable of supporting without a ‘rebuild the document from scratch every time’ strategy, from data stored in some repository¹. Here the *document* is the focus of design – it alone holds the data and is the sole generator of the step-varying presentation, and it may lead to novel designs for *active documents*.

13.1 General discussion

Robustness and resilience

DDF was in part given functional foundations to increase the robustness of the documents and the ability to extend their functionality in a predictable manner. Generally this has been the case, especially in adding new layout capabilities or building ‘library’ documents. There are many times when simply adding a few XSLT templates within a DDF document has built the required new functionality – a good example was adding a ‘macro’ facility so that compound constructs, such as the complete tree/image/XML pictures shown in the thesis, can be defined in the input data stream. The interpolation of calls on these macros is handled by code that is defined in a library document (**ddf-struct-macros.ddf**) which expands macros between the *data* and *struct* spaces. Of course there are some cases where there is somewhat less robustness, or more accurately resilience, than might be desirable:

The use of push-driven template processing in document authoring requires some discipline in deciding on resolution priorities and, when matching cases are spread across several documents (e.g. in overloading), conflict can be difficult to control or even recognise. Disciplined use of modes can assist, but in document templates as large as that for this thesis, a few such problems are inevitable.

As far as GUI-based authoring is concerned, the editor discussed briefly in section 4.3 had few problems of reliability. Assuming documents being edited were ones that were authored ‘from scratch’ on the tool, then the types of layout that could be created or modified were strictly defined by the ‘editability’ configuration and, apart from issues of variables giving a null result on a specific data instance (see Figure 31 for a possible solution), document editing rarely failed. And the editor also had the possibility of providing a direct (graphical) view of the document XML structure to provide structural feedback to the author.

Failures in the processing of documents in the framework appear in three forms. In some, rare, cases the failure is ‘silent’ and possibly a smooth degradation. Several paradigms can be employed to support this, such as providing default values for a paragraph width from the layout agent processing text (and raising a warning). Within constrained layout (section 6.4) a set of constraints can be written that are legal but have conflict under certain circumstances

¹This is the approach taken by *DialogueLive*.

– the constraint solver always produces a ‘minimum error cost’ solution but this can sometimes lead to very strange layouts. Other failures can be caught by type and syntax checking by the XSLT processor (Saxon provides line number identification which could be intercepted and traced back to the offending original DDF source document).

Finally failures can occur at runtime (in the intermediate XSLT or the layout processor), normally associated with either a type exception or an error in XSLT construction ordering, such as writing attributes after children. The latter is rather difficult to track down in large documents and processors, even by the author. There are a small number of checks built into the layout processor ‘runtime’, such as reference to non-existent presentational variables or requests for unknown forms of layout combinator. But in general there has not been a systematic approach to failure detection and amelioration in the current experimental software suite.

In a system as large as the DDF implementation toolset, there are of course some sections where coding has to be very accurate – minor changes in recursive data structures can cause collapse of the intended functionality, or make debugging very difficult. In part this can be caused by the lack of a strong typing system in XSLT – an example is the use of XML trees to describe edge properties of SVG graphic trees for the tree-layout described in section 6.3 where strong typing may well have helped. Similarly the support for presentation variables (section 6.4) required more difficult coding than would have been the case if XSLT supported ‘sequences’ of ‘sequences’.

Document code that is complex because it declares interaction between processes in two different semantic spaces needs some care. The hybrid XSLT/layout models of section 9.8 show such capabilities but are very sensitive to the correct ordering of XSLT instructions. Defining compound instructions (such as **ddf:revealSVG**) which are compiled into suitable constructs decreases errors in document construction.

In other cases the domain problem just grew. The ‘deep quoting’ of documents that this thesis contains, put much strain on trying to keep track of relatively-referenced resources (i.e. images), despite the context-mapping system described in section 5.1. Everywhere some section of a variable document instance was quoted, not only did that section have to be collected, but the context map for that document collected and merged with current context, with tag clashing being inevitable. An alternative approach is postulated in section 13.3.

Efficiency

The research described was aimed primarily at building smoothly extensible functionality into a variable document architecture, rather than pure execution performance. Given the generality, it should be slower in execution than a specific-purpose document format and associated support processor, such as XSL-FO processed by Apache FOP.

One data point of interest is that this thesis, some 260 pages long, with a source of ~ 60k words, ~120 figures (being a roughly equal mixture of images, SVG/XML graphics, many of which are examples computed during the construction, and PDF pages), takes just over 6 minutes to generate to final PDF² – most of this time is taken up resolving the layout tree, which is upto 32 levels deep, has ~27k elements and ~95k attributes. Anecdotally this about an order of magnitude slower than TeX or FOP would process a report document of this size.

In terms of the necessary information processing, by extracting all the text paragraphs and evaluating them separately *en masse*, we can show that some 2 minutes is taken up purely with text expansion, regardless of any compounded layout. In practice many of the layouts involve repeated expansions of text – this is an area where there is considerable possibility of tuning the layout agent code to ensure that intermediate results are preserved. In particular the interpolation of presentational variables (used to correctly provide a width for a paragraph in pagination, Appendix B) currently involves some potentially redundant re-computation. There is plenty of scope for tuning the supporting code and exploiting ‘already evaluated’ markers on results.

13.2 Key findings

Several key ideas have been raised through Parts B and C which have proved crucial to the flexibility of the document designs discussed. Collectively they are:

- **Universal use of XML.** All structures: data, program, layout intent and grounded graphics are described in XML syntaxes. The use of a common-tree-based syntax automatically provides features for containment, scoping and meta-level behaviour.

²See Appendix E for more details.

- **XSLT defines variability.** Using a ‘functional’ programming language whose main data type is an XML tree and which is also written in XML, supports the description of extensive document variability directly within a document as executable code fragments. These can be embedded in other XML structures and manipulated by XSLT in a similar fashion, thereby providing a route for extending the XSLT model into self-propagating code and partial evaluation, controlled by directives.
- **Layout as hierarchical combinators.** By describing a document's presentational layout as a tree with element nodes that declare relationships between the presentations of their child sub-trees we can support an extensible and rich repertoire of layout and encourage the use of locality and scoping. Single assignment presentational variables can support tree-based acyclic layout dependencies; global effects can be defined by high-level declarations.
- **Interspersed namespaces.** Document sections and behaviours operating in different phases and with differing intents can be interspersed within document tree representations, using namespaces to preserve separation of concerns.
- **Good XML citizens.** Programs and tools should behave as *good XML citizens* by i) tolerating and allowing for unknown information (elements and attributes) within their XML source material and ii) transferring such unknown information into approximate tree-isomorphic positions in their results. If this is so, ‘higher-level’ operations can transmit information through intermediate processing steps.
- **Approximate tree-isomorphism.** If possible the major evaluation steps should generate similar-topology tree structures in their results matching those found in their input. This is especially true in the resolution of layout declaration to final grounded graphical form. By so doing, preserved ancestry and grouping in results can support ‘higher-level’ operations.
- **Layout intent as attributes.** Layout intent can be added as attributive decoration to a grounded SVG graphical result, leading to *idempotency* in layout. Possibly modified results can be reprocessed, yielding the intended output without recourse to an independent definition.

- **Hybrid action.** Hybrid actions between sections in differing semantic spaces (XSLT and layout) interspersed within the overall XML tree, support active and robust modification of a previously bound document. This can range from alteration of simple properties (e.g. sizes) to substantial topological changes when entire sub-trees are added, deleted or replaced. Modification of processed layout is performed by the XSLT phase adding new components or changing signals in the layout tree, and the subsequent layout phase when evaluating *meta-layout* functions, choosing *not to copy* some components based on those signals.
- **Compiler support.** Using an XSLT-based compiler which generates an executable to implement the semantics of the document is very powerful. It enables extensions to the document model (e.g. tracking external resources) to be supported, but most importantly enables retention of programmatic intent in variable documents, through partial and multiple bindings. Such retention can be indicated by extended declarations in the document sources.
- **Indirect XPath evaluation.** Much document flexibility can be described with the use of embedded XPath expressions to examine, select from and compute over many points in the document: data, logical structure and presentation. Useful and robust models can be built provided that such expressions can be extracted and evaluated at runtime. In XSLT3.0 this will be possible as standard; in XSLT2.0 a fully-featured extension function suffices.

Some of these points are worth more discussion here, namely, universal use of XML, interspersed namespaces, XSLT as the programming language and compiler support.

Universal XML

Having almost everything described in XML syntaxes is probably the most important feature of all in this work. Firstly it means that *everything*, ranging from data through to program and document is provided to programs in a primarily-parsed form, and that a uniform search mechanism, *XPath*, can be employed whenever structures need to be accessed. These facilities are not confined just to data access, but also to examination and modification of program code. Arguments about the size and verbosity of the serialised forms of XML are not relevant for the in-memory structures over which the main computation takes place.

Secondly the tree nature of XML provides a natural mechanism for locality, scoping and grouping that works well with the generally localised and partitioned nature of most documents and presentations. This is best demonstrated by SVG, where deeply grouped sub-trees (**svg:svg**,

svg:g) provide mechanisms for retaining grouping in presentation by acting as a focus for common geometric transforms and styling coupled with inheritance down the tree. But more importantly for this work it also provides attachment points for group-related properties that can be used for compound layouts (such as naming groups) and higher order effects, most notably recording layout intent for later re-satisfaction.

Interspersed namespaces

Combining different semantic models within a single XML tree, by interspersing elements and attributes in differing namespaces has also turned out to be vital for this work. Firstly it means that the *actions* of one namespace can take effect at the correct places in a tree of another namespace. Obviously the use of embedded XSLT fragments within SVG and DDFL means that presentation can alter as a result of variability in exactly the way intended – there is no need for some other system of declaring where a computational result will be placed. (XSLT's ‘push’ mode processing is merely a pattern-directed model – the actual result effect only appears where the **xsl:apply-templates** statement was located.)

Moreover, processes may examine the situation from which they are invoked and allow for, or even modify, structures in other namespaces. The obvious examples are the interpreted resolution of DDFL layout, where ‘foreign’ elements are tolerated and preserved, and embedded XSLT being able to write signals into the DDFL structure within which it is embedded.

XSLT as the programming model

Apart from being written in XML and therefore embeddable within and around other XML structures, XSLT's functional nature is critical. Having a program that is free from side-effects, and with a well-defined scope for action and naming, means that transformations can be performed on program fragments correctly and relatively simply. This is especially important in adding self-propagation to code sections, via compiler-recognised declarations.

Document compilers

Many necessary features for a practical document system are not defined by default in a standard programming language, or fragments thereof that might be buried in variable documents. The clearest examples are i) following data-contained source indirection pointers, ii) tracking data elements that by inference will point to possibly position-relative resources, mostly images,

and iii) higher-order code-propagation semantics.

A document designer could add suitable code to implement these three things, or add calls to functions in some system library. But it is more coherent to extend the model with clear semantics and control, with a declarative syntax, and to use a transforming compiler to generate appropriate executables. In the first example *XInclude* semantics can be requested for data input: the compiler adds templates to the executable for intercepting and processing such requests in the input stream³. In the second case patterns can be declared to match input that names a resource: templates during execution will check for relative paths therein and either make them absolute, or track location via a *context map* mechanism. Writing code that propagates itself – a facility needed within higher-order operation – is aided considerably by a compiler that can correctly transform code against declarative intent.

13.3 Redesign

From the start DDF was considered to be a research tool, exploring possible features and mechanisms for extensible variable document formats and architectures. Some aspects (security, human interactivity with a document) were considered to be possibly orthogonal to the central concerns of structure, variability and layout, and were excluded from the research. What might feature in any new design in the light of the experience described in this thesis?

SVG as presentation and layout

Firstly we might consider describing all presentation in SVG, with layout intent attached as attributes or specialist elements in separated namespaces. That is to replace **ddfl:layout** combinator nodes with **svg:svg** elements, decorated suitably. It should be possible to make the presentation legal SVG by exploiting the **svg:metadata** element as a container for foreign subtrees, provided that some compiling operation can ‘promote’ the consequences of, say, an XSLT fragment, through the metadata wrapper. The **@display** attribute in SVG can control visibility and geometric effect, and has not been used thus far – it might act as a suitable ‘guard’.

One feature of SVG has not been used – **svg:use** for calling re-usable components. These can be any drawable element, though **svg:symbol** is recommended for groups; a guideline is that re-usable components are contained within **svg:defs** elements that are direct children

³Later versions of Saxon supported this as a direct option.

of ancestors of referencing **svg:use** elements. A drawback is that such elements are called using global (IRI) references: these are *not* tree-local. This construct might be suitable to represent the presentational variable system (**ddfl:variable**), though the post-presentational program capability (**ddfl:for-each ...**) may need something different. The model of Thompson, King & Schmitz[93] using **template** and **instance** variants is worth considering.

Proposals are being considered for SVG 2.0 that may be helpful. The most important of these is text flow which should be explored to determine how it can be used in variable and mutable situations. There will definitely need to be some representation for pages – until the SVG Print sub-standard produces a recommendation, those in the SVG 1.2 proposal (**svg:page-Set/svg:page**) and used already, must suffice.

Hold data and structure within presentation

Consider placing equivalents of the **ddf:data** and **ddf:struct** spaces within the presentation (e.g. within **svg:metadata** children). In this way the document is inherently a presentation – program is buried within it to respond to variability. Make the use of a structural level optional, or better still we might permit any number of named ‘layers’ between data and presentation result.

For some documents (especially reports and papers) the logical structural space plays a useful role, providing a canonical layer for reuse. For others, such as the brochure examples, and higher-order documents, such a layer serves little purpose, save providing a ‘space’ for some intermediate computation (e.g. converting maps with labels into labelled images).

Use higher-order XSLT and additional compilation

Exploit all the higher-order features of XSLT3.0, not just ‘indirect’ XPath evaluation, but also functions being treatable as first-class objects, higher-order functions such as fold and map and support for lambda functions in XPath. The use in Saxon of compiling and executing transforms at run-time could also be exceptionally powerful.

The layout processor is a very large XSLT-based interpreter that roams over a document's presentational tree definition while building up the eventual presentation tree from in a bottom-up manner. For layout of large documents, such as this thesis, it may be advantageous to build a compiler that converts the layout definition into an XSLT program (incorporating extens-

ive support libraries for the layout stubs) that will execute significantly faster than the interpreter. As an example, the presentational variables are used extensively, not least for propagating an appropriate text column width to paragraphs. Values of these variables are interpolated by searching by name (with XPath) through a stack frame. Converting their operation to use XSLT variables interspersed through sections of layout resolution could enhance performance greatly, as well as perhaps avoiding some of the complexities of implementing the interpreter. An interesting research problem would be to try partial evaluation of the interpreter (without executing its necessary extension functions) on a layout to generate an intermediate XSLT for final execution.

An additional possibility is that the layout processor starts by invoking XSLT semantics within the document presented to it (if it contains such) – i.e. the layout processor is actually a compiler for the entire document, generating the resulting layout-computing program parts as well as the more usual data processing and structure building code⁴.

Simplify external resource tracking

Tracking relative external resources, even using the *context map* system that was described in section 5.1 has proved to be both very useful and yet problematic, especially in quoting material from the results of other variable documents⁵. When importing part of an example not only does that fragment have to be interpolated, but any relevant entries in context maps that its resources may use must also be collected and added to the set of locations to be tracked, so that, subsequently, the correct resource can be consulted or embedded. Name clashes are inevitable, requiring some renaming mechanism.

In retrospect, if a single evaluation session takes place on a single machine, an easier alternative might be to convert to absolute locations unilaterally on start of processing and, at completion, consider reversing to some relative arrangement, possibly collecting resources using an archival form such as ZIP. However, if processing is scattered all over Νεφελοκοκκυια then some more fundamental distributed reference system will be needed⁶.

⁴Most presentational variables in DDF documents thus far are statically named and thus should be amenable to direct conversion through such a compiler. However the pointers between resorts labels and pins in the maps of the travel brochure (Chapter 7) use presentational variables whose names are generated at run time.

⁵Such as a PhD thesis on *Documents as Functions*

⁶See Aristophanes' *The Birds*.

13.4 Other further work

Apart from a redesign of the architecture already discussed, several interesting research topics arise from this work. The first concerns how a designer can be assisted understanding how a variable document might perform and what sort of feedback can be given to users. The second comes from large-scale document work-flows: what methods should be used to represent and reason about a variable document's *type*? This has two aspects: what is a variable document's *argument signature*, i.e. what does it expect to consume, and how can it interact safely with other documents, to support effective document reuse.

The third is from editing variable documents through *instance-view* graphical user interfaces (section 4.3). It can be arranged that indirections from the presentation can effect edits on the source variable document but resultant changes to the presentation require the entire document instance to be recomputed. For documents of any appreciable size this takes far too long. Can some form of correct *incremental* change to the instance presentation be inferred and computed cheaply?

User feedback

The DDF framework currently presents little feedback to the designer from the document itself. Syntactic errors in XSLT will manifest themselves as errors in execution, but ‘silent’ errors, such as an XPath unexpectedly yielding a null result, are not detected. A system of assertions (attached as attributes, e.g. `@ddf:assert="count(title) le 1"`) could be exploited by the compiler to add run-time checking, which could manifest itself in raising error conditions or providing a run-time report – this can even be added visually to the graphics. These assertions could be created and managed through the visual graphics editor described elsewhere in this thesis.

A similar approach could describe response to partial binding (section 10.1), where hints can be edited just like any other property and sections of code (e.g. the `@special-offer` conditional) can be identified as possible ‘optional’ sections. Alternatively schema attachment or editing could be handled in a similar fashion – if one of the XML-based descriptions (*RelaxNG*, *XML Schema*) were used, they can be manipulated by the same tools.

Users are often unaware of, or have difficulty understanding, the hierarchy inherent in the layout. With the approach to editing used it is distinctly possible to provide multiple co-ordinated views of a document, one of which is a graphical view of the layout tree, so clues can be provided

– selection highlighting is synchronised between all views of the same document. All these areas hold much promise for future research.

Document type

A significant area for future work is in a theory of *type* for such documents. Much work in conventional XML processing involves the use of schema-like descriptions (Document Type Definition[116], XML Schema[124], RelaxNG[112]⁷) of permitted XML syntactic compositions for a given application. These descriptions are primarily used to verify that *input* XML data is acceptable to the application, aiding robustness. In some cases, such as the Saxon XSLT processor, typing of interior XML structures, definable in XSLT via the **as="type"** attribute where *type* can be a schema term, can be used for run-time checking as well.

The DDF document obviously consumes input data in XML form and we could have built a system for supporting schema checking – it would have been of most use in reducing ‘breaking’ type clashes (e.g. encountering a general string where a string representation of an integer was expected.) Such a system would be needed for full commercialisation.

Of more interest is using schema systems to represent type in such a way that document behaviour can be reasoned about, in particular when documents are *combined*, whether as peer-siblings or as imported libraries. A good example would be where one document is used to convert data to some logical structure, and another set of documents generates different presentations from such a structure. If the logical structure was described via a schema (which could be so for a standard such as XHTML) then that schema can be both added as type signature to the appropriate functional parts of the documents and also perhaps checked dynamically or statically against the inherent programmatic semantics.

Editing documents and incremental change

There are some drawbacks in the overall approach to editing described earlier which would warrant further research:

- Editing can only be carried out on causality nodes that *actually appear* in the document instance being viewed. A node in the template whose projection is conditional may not appear in the result for a specific instance and thus lack any means of selection by ‘pointing’.

⁷The last two are written in XML and thus easily parsed and manipulated via XSLT – DTD is not.

- Users have difficulty editing deep layout structures and understanding what is possible. It should be possible to define appropriate macro-forms (e.g. a flow above an alignment) that whilst for the implementation is a multi-level-deep tree, is considered by the editor, and displayed for the user, as a single entity.
- To build a practical system there are a number of engineering issues, such as adding models for more interactive editing (e.g. mouse-based manipulation – drag-n-drop, rubber-band) which will make stringent demands on minimising necessary re-work.
- But most seriously, any change in the function could have repercussions all over the result document, so the safest approach is to regenerate the document instance after each change. For a large document this can be expensive and results in slow response to edits. Support for ‘partial rebuilding’ of documents to decrease editing response time would be exceptionally beneficial. (This was a stimulus for Ollis’ PhD[80].)

All of the document projections discussed so far have involved complete or partial evaluation of a document and its inherent presentation from a ‘null’ start. We have data argument \mathbf{x} and document function \mathbf{f} and we then proceed to evaluate the result $\mathbf{f}(\mathbf{x})$. More complex arrangements make \mathbf{x} a possibly infinite sequence of steps $(\mathbf{x}_0, \mathbf{x}_1, \dots)$ which \mathbf{f} can accommodate, producing a series of outputs: $\mathbf{f}(\mathbf{x}_0), \mathbf{f}(\mathbf{x}_0)(\mathbf{x}_1), \dots$

We have also described finding invariant sections that can be pre-evaluated in a document, so that for long sequences of data bindings, $\mathbf{invariant}(\mathbf{f}) \Rightarrow \mathbf{F}$ followed by $\mathbf{F}(\mathbf{x}_0), \mathbf{F}(\mathbf{x}_1), \dots$ is significantly cheaper than $\mathbf{f}(\mathbf{x}_0), \mathbf{f}(\mathbf{x}_1), \dots$. However, as we have seen in editing, when minor changes are made to a document, the whole document is then re-evaluated. For tasks such as character-by-character interactive editing this is unacceptable.

The interesting technical challenge is document *differentiation*: if we have a (multi-dimensional) data point \mathbf{x} and the result $\mathbf{f}(\mathbf{x})$, what can we say about $\mathbf{f}(\mathbf{x}+\epsilon)$, where the change to the data argument is small, such as an additional word? Or alternatively what can we say about $(\mathbf{f}+\epsilon)(\mathbf{x})$ where the change to the function is small, such as altering the width of a single text-box? Can we build the rough equivalents of Taylor-series expansions? Can we calculate a derivative function (program) so that $\mathbf{f}(\mathbf{x}+\epsilon) = \mathbf{f}(\mathbf{x}) + \delta\mathbf{f}/\delta\mathbf{x}(\mathbf{x})(\epsilon)$ where $\delta\mathbf{f}/\delta\mathbf{x}(\mathbf{x})$ is a significantly cheaper computation than $\mathbf{f}()$?

Obviously this will not be a trivial undertaking – the functional domain and range, as we have

seen, are highly multidimensional and in tree form; the functions can be discontinuous, non-monotonic, non-linear and discrete.

Ollis[80] has explored recalculation of an XSLT computation from a stored intermediate state, but work in this direction will need a more theoretical examination of the document functionality itself. Important issues will include deciding limits of ‘monotonicity’ (where positive change starts to make ‘negative’ alterations elsewhere in the document) and ‘linearity’ – i. e. regions where the behaviour is additive and principles of superposition can be exploited.

But it is clear that such things can be considered and, with careful design, can be added to the implementation machinery. For example when a paragraph has been set there is normally a partially-filled line at the end. So, as part of the return from that operation, the remaining ‘space’ on that last line can be recorded. If the text binding for that paragraph then changes only *by addition of characters or words at the end* then we have a cheaply-computed superpositional solution – using font metrics we can calculate the horizontal extent of the addition. Provided it is less than that remaining, the new text can be superposed without any other alteration to the entire document⁸. After that point other parts of the document may have to be rebuilt, until the ‘spare line’ is available again.

In a similar manner some changes to the function definition itself may be highly localised in effect – change in wrap-width of a paragraph may have little or no ramifications beyond the paragraph until the number of lines changes. These types of operation can build on some of the work of Macdonald[67] on partial evaluation in layout.

13.5 Lessons

Research in document engineering, at this level of both generality and detail, inevitably teaches one many lessons and leads to several conclusions. This section outlines some of those, both positive and negative, and some of the suggestions for those fortunate enough to be following similar paths.

XML and XSLT

The research is entirely based on using XML as the data type for *almost everything* – doc-

⁸Higher-level features such as automated index generation would invalidate this – but such features could be repaired in some final generation.

ument, data, presentation, programming tools and languages, intermediate and ancillary data forms and other descriptions of many forms: work flows, editability, maps for colours in the thesis and so forth. And the XML is accessed by just *one* mechanism – XPath, and created almost exclusively by programs in just *one* programming language – XSLT.

In many other cases as a research software engineer I have employed several languages in a hybrid manner within a project, each focussed on what it was best at – *Perl* for text alteration, *Mathematica* for symbolic manipulation, *make* for building large-scale software suites, etc. But for this research, and this style of document engineering where flexibility is a goal, the wholesale use of XML is appropriate, for the following reasons:

Documents are essentially trees. Most of the relationships that are important within documents are associated with locality, sequence and hierarchy, both in structure and in presentation. (As we have seen there is often strong isomorphism between the two.) These relationships can be represented naturally within trees. Other relationships that are not castable as trees exist of course, but for most regular document forms they are either constraints on similarity (which can be expressed as isomorphisms across a larger tree, such as using **xsl:attribute-set**) or are minor in number and importance and may be represented as ancillary structures describing graph edges placed on top of the main tree. Using a data type that *is a tree* means that these important relationships are represented inherently.

XML is sufficient and expressive enough to define document trees. As a tree-based meta-syntax with a model for namespaces, it can represent different aspects of a document's semantics as interspersed subtrees. Element nodes can have scalar properties (possibly in differing namespaces to reflect differing semantics) through extensible sets of attributes⁹. Textual values can be attached throughout a document tree and have well-defined systems for character encoding. Subtrees (such as XSLT) can be placed where they are expected to act, or within suitable scopes within which they will have effect. Type declaration standards for specific XML syntaxes can be defined. Serialisation standards exist for generic XML trees, as well as abstract APIs for manipulation. In-memory models for XML, used in many tools, can support the processing of XML trees with very large ($>10^7$) numbers of nodes.

A uniform method of access – XPath. The XPath language provides a uniform, declarative and highly expressive means of access and search within all such tree structures. Many

⁹Using LISP would require a strict protocol for representation of properties, text and elements within S-expressions.

of the common requirements within processing documents ('is all the text in this paragraph in the same font and size?') can be expressed as XPath statements (**count(distinct-values (descendant-or-self::*/@font-family)) le 1 and...**). When used in XSLT processing of a document it becomes even more powerful.

Sources and results are XML trees. If processing tool chains behave, as a matter of course, as *good XML citizens* and generally preserve *tree isomorphism* between source and result, higher level processes and operations can be built in a robust manner, often through tools that have no knowledge of these operations. The most extensive example is in the experimental 'editing on a document instance' (section 4.3 and [63]) where large channels of editing data flow from source DDF documents to SVG result elements because of these protocols.

XSLT consumes XML, generates XML and is defined in XML. If a variable data document is to have any possibility of behaving as 'continual function', then such a document must be able to *contain the description of its own variability* and any modifications to such. In any non-trivial case of variable response (i.e. not just simple interpolation of data, but with conditionality, choice, iteration and so forth) then this description must be programmatic and capable of generating sections of XML within a document. XSLT satisfies both conditions: it is perfectly capable of supporting very high complexity (including self-propagation) and can be carried in an XML document without special syntactic treatment.

This uniform use of XML combined with XSLT has made many parts of the implementations comparatively straightforward, sometimes even trivial. The layout processor has many examples, one of the best of which is changing the size of font in a text block to make it fit a given rectangular container (see section 6.2) which requires only seven actions:

1. Paragraphs that can be treated so are marked with the additional attribute **@mutable-font** (a trivial extension of the syntax);
2. during layout such cases are recognized by a higher-precedence agent matching **fo:block [@mutable-font][@width][@height]** which takes control (XSLT's 'push' processing model and XPath-based patterns);
3. a trial layout with the font-size originally anticipated is performed through **xsl:next-match** ('push' processing again)¹⁰;

4. the result of the trial can be examined with the expression **\$test/@height** (uniform access through XPath);
5. a new font size can be determined using a clear arithmetic XPath expression;
6. the paragraph can be created by copying the original, whilst removing the mutability declaration and writing a new font size property (XSLT's model of copying and modifying trees);
7. the final result is generated by the layout processor evaluating this modified paragraph with **xsl:apply-templates select="\$new"** (XSLT push again)

Such uniform use of XML/XSLT becomes engrained: in the workflow processor (a Java-based suite that examines document dependencies, determine necessary regenerations based on modification times and launches processing threads) it was sometimes easier, and much more robust, to generate a suitable XSLT transform and then compile and run it on an contained XML structure rather than write Java code to modify the tree directly.

Correct choices

Apart from the XML/XSLT choice just described, many of the other high-level choices seem correct from practical experience:

- SVG has been very suitable for defining the resulting graphics: i) the suite of graphical and textual primitives is sufficient, ii) a general paradigm of using **svg:svg** to describe a translatable group with rectangular extent has been exceptionally useful, iii) models for affine geometries, colours and other resources have fitted in smoothly and iv) the similarity of its graphical model with PDF makes generation of final printable documents easy. As it is XML-based, the later addition of declarative layout intent, central to Chapter 9, disturbed none of the existing support tools¹¹.

¹⁰The layout agent has no knowledge of how the **fo:block** is evaluated – only that the result is expected to return as a canonical **svg:svg** structure, whose area is an approximately quadratic function of **@font-size** on the paragraph. If some other effect (e.g. rotation) is defined, that will be intercepted by another agent before or after that dealing with *font-warping*, depending upon precedence, and both effects should still be honoured.

¹¹But documents might still be affected! The author was surprised when the ‘constant-folded’ brochure samples (Figure 77), which previously had appeared as anticipated in the thesis, started to display wrongly. Of course, during the work on Chapter 9, they began to contain sections of SVG that correctly declare their own layout intent (flows, page numbering, interpolation of column width) and were now being re-satisfied during layout of the *thesis*, displaying incorrectly by execution in the wrong environment. The importation needed to be ‘smart’ and remove such decorations, including embedded page-numbering declarations.

- Using the syntax and semantics of the **block** from XSL-FO with a few additional properties, has provided a model for text that has met all major needs thus far.
- The combinatorial model for layout *has* been highly successful, especially in the extensibility of layout as hopefully this thesis has demonstrated¹². One of the best illustrations of this is in Figure 35 where, with the exception of adding a layout function to ‘add a clip path’, all the rest of the effects (trees, execution of the layout ‘program’, links between zoom and source) are defined by document-borne macros using layout primitives that were already present in the default processor.
- Choosing to construct a DDF compiler provided a platform for the manipulation of program. Originally this was intended just for collecting all the programmatic sections of a DDF document together to generate a program that would successfully execute the variable intent of the DDF document. But later this became the point where declarations of higher-order behaviour could be converted into appropriate code, including those of self-propagation.

Difficulty, error and disappointment

Not all designs and implementations were quite straightforward. Some features have worked very well, but needed exceptionally careful implementation. Two in particular feature strongly in this thesis:

- Firstly, whilst defining and designing the model for presentational variables was not difficult (ευεηκα! – the **xsl:variable** model had the necessary semantics), implementing and testing was much more delicate (including modifying interpolating XPath expressions to access such variables). The code supporting the presentational variables is still considered ‘touch only when stone-cold sober’¹³. In addition, detecting and analysing the presence of presentational variables within the XPath-based interpolations was carried out using regular expressions. Really degenerate cases (e.g. strings embedded within such expressions that contain *\$var*) would cause breakdown, though this has been avoided thus far. The best solution would have been to acquire a proper parser for XPath into an XML representation and handle the transformation of the expression properly¹⁴.

¹²On more than one occasion the author thought “If I add a small section of pre-emptive template to *X*, I can achieve *Y*”, only to find that years before, he had written such code, which had lain there unused and forgotten...

¹³A useful feature was to reserve the variable name **ddf:variables**, so that its interpolation returns all the variables currently in scope defined as a sequence – this can be used for certain types of in-graphics debugging.

- The second was the layout for a condensed tree, used widely in illustrating examples both in this thesis and other papers and sometimes as an aid to debugging¹⁵. Naturally its argument is the sequence of XML trees that are its children: there are many variations of what illustration should be used for a given node in the tree – all defined declaratively. But determining the condensed layout for a tree is a doubly recursive implementation, using an XML tree to represent the graphical tree properties of a contained XML tree. Fools rush in

Some areas proved both difficult and not completely successful. Amongst these we can cite very accurate pagination, automated handling of external resource reference and ‘packing layout’:

- Detailed pagination of a report, like this thesis, has several ‘corner cases’ that may need to be recognised and handled specially. Some issues are inherently indeterminate, such as footnotes appearing near a page end migrating to the following page, and are just tolerated. Others do require attention, most notably breaking paragraphs at column end, where the paginator needs to know more about the representation of text paragraphs (as opposed to generic subcomponents) and how wrapped lines are represented (including artifactual hyphens) than we would wish, and it does not perform completely satisfactorily. This is a definite case where an implementation for a very specific flow model, such as XSL-FO, would perform more effectively but less generally.
- Inclusion of content that itself has external reference (e.g. to image resources) has been described several times so far, with the use of context maps as a potential solution. However for a document such as this thesis which itself quotes from sample variable documents that themselves have gathered indirect resources from data, the problem compounds immensely and the author sometimes had to resort to patching to generate the intended effect.

¹⁴Whilst of course an XSLT compiler must contain such a parser, curiously the Saxon implementation does not have an XML output form of such a tree, at least not in 2006.

¹⁵Such as spotting an alien namespace element in a large structure.

- Designing a series of sample documents, one of the team wanted a layout that was essentially ‘pack from the outside in’, with children packing against nominated sides of their parent container and either defining their own sizes or sharing the available free-space with other siblings, all naturally in a recursive manner¹⁶. The design processed these by a hybrid approach of calculating dimensions where possible and propagating such consequences, and converting other cyclic dependencies into a series of constraints that could be solved in the usual manner. This would work for some cases, but the non-linear and discrete nature of unbounded text made the packer too fragile to be used by the uninitiated.

Other areas were explored and either failed to work, or were far too complex:

- It is always tempting to use existing documents (prepared by skilled human designers) to provide samples from which a document's inherent structure and styling can be extracted and reused for other purposes, not least of which is making it *variable*. Some PDF documents were converted successfully into reasonably coherent SVG and thence DDF¹⁷, but, in common with the work of Bagley *et al*, much heuristic consolidation was needed to recover complete text paragraphs and possible styles¹⁸. Further work described the relationships between major components as a constraint network, which made the document's layout geometry (but not topology) variable. By labelling some components as variable, i.e. selecting a paragraph and permitting its text value to be altered, a variable document could be generated. Sometimes it worked, but results were very well short of being robust.
- Text blocks with variable dimensions were supported by using Lin's approximation of a set of linear inequality constraints[57] but it soon became apparent that in general this technique was both too expensive and far too fragile to use to solve most problems with free text.

Perhaps the area of greatest disappointment, but not for technical reasons, was the variable document editor. The implementation was a *tour de force* in the use of XSLT to take a declaration of intent (document elements that were to be editable and how they might be edited) and to generate from them an entire suite of XSLT transforms: to inject editing trace attributes on source documents; to build modified SVG views that highlighted, and responded to, editable sections in graphic results; to construct suitable populated dialogues for those edits

¹⁶Similar to widget packing in GUIs.

¹⁷Using *PDFBox*[99], a Java-based library that is particularly good at traversing the PDF drawing stream.

¹⁸One test document appeared to use more than 10^4 different fonts. More detailed examination revealed that the font varied on a syllable-by-syllable basis by less than 0.001pt – I suspect this was a steganographic watermark.

and to perform the requested alteration at the appropriate point on the correct source document.

The provided workflow was also modified automatically to add all these features during an editing session which was run on an actual instance of a truly variable data document. This was published and demonstrated at ACM DocEng 2008[63] and later some user trials of a Web-deployed version were run. But two issues arose: i) having to rebuild the entire document completely after each editing step made in-text typing interaction impractical (a motivation for Ollis's PhD studentship[78]) and ii) a human editor operating in WYSIWYG mode generally only sees surface level features and the deeper (combinatorial layout) structures which DDF encourages and supports very well, are understood poorly, if at all. This latter point perhaps hints that any deep (layout) structure in a document is destined to remain with those are comfortable with the abstract, e.g. the document engineer, and that variable documents developed by human authors are restricted to shallow models of variability.

13.6 Conclusion

A variable document architecture that is highly extensible, will generate fully professional results, is not restricted to one generic class of document and is capable of operating with some 'higher-order' semantics, can be built using almost exclusively XML technologies and an XML-based functional programming language.

The key enablers for achieving this are i) using a hierarchical final-form graphical representation (SVG) to provide grouping, locality and scoping, ii) interspersing information in other namespaces within such trees, especially generative programs in XSLT, iii) describing layout intent declaratively within such a tree and using a recursive descent set of processing agents to resolve the required layout, iv) having all tools behave as *good XML citizens* and v) using transforming compilers to support 'document management' and 'higher-order' semantics, including code propagation.

FIN

APPENDICES

Appendix A

Detailed Views of Main Examples

These are larger versions of the examples used in Chapters 7 and 12.



Travel Heliskiing

Specially selected for George W. Bush

Val D'Isere



Very little is known about how and where they live. The one certainty is that they are opponents. Nam libero tempore, cum soluta nobis est eligendi opto cumque nihil impedit quo minus id quod maxime placeat facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet ut et voluptates repudiandae sint et molestiae non recusandae.

Nam libero tempore, cum soluta nobis est eligendi opto cumque nihil impedit quo minus id quod maxime placeat facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet ut et voluptates repudiandae sint et molestiae non recusandae.



All-inclusive: £1405

Grindelwald

Very little is known about how and where they live. The one certainty is that they are fearsome opponents. Nam libero tempore, cum soluta nobis est eligendi opto cumque nihil impedit quo minus id quod maxime placeat facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet ut et voluptates repudiandae sint et molestiae non recusandae.



All-inclusive: £2005

Travel Heliskiing Registered office: 19 High Holborn, LONDON W1 4NB
All data and information provided is for informational purposes only. Hovest John makes no representations as to accuracy, completeness, currentness, timeliness, or reliability of the information and will not be liable for any errors, omissions, or delays in this information or any losses, injuries, or damages arising from its display or use.

Page 1



Travel Heliskiing

Whistler



Nam libero tempore, cum soluta nobis est eligendi opto cumque nihil impedit quo minus id quod maxime placeat facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet ut et voluptates repudiandae sint et molestiae non recusandae.



All-inclusive: £997

Terrace Special Offer!



Nam libero tempore, cum soluta nobis est eligendi opto cumque nihil impedit quo minus id quod maxime placeat facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet ut et voluptates repudiandae sint et molestiae non recusandae.



All-inclusive: £997

Travel Heliskiing Registered office: 19 High Holborn, LONDON W1 4NB
All data and information provided is for informational purposes only. Hovest John makes no representations as to accuracy, completeness, currentness, timeliness, or reliability of the information and will not be liable for any errors, omissions, or delays in this information or any losses, injuries, or damages arising from its display or use.

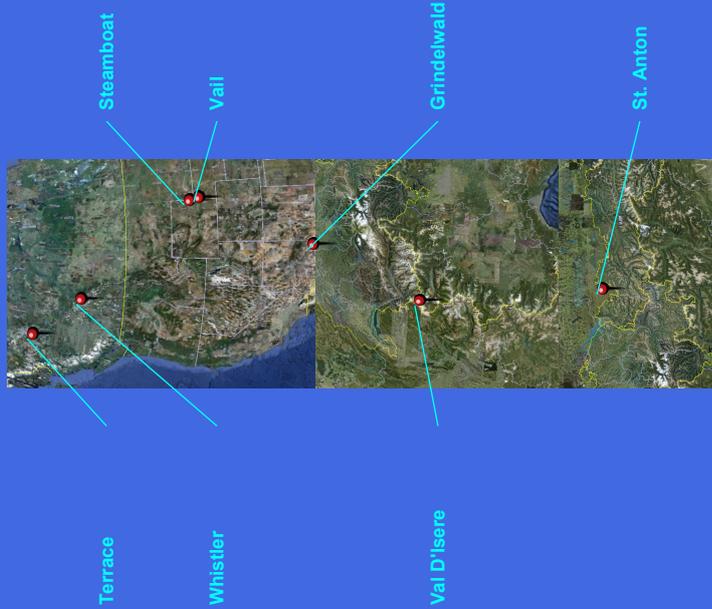
Page 3

Figure 114. Skiing brochure – pages 1 and 3



Travel Heliskiing

Where these bargains are!!



Travel Heliskiing Registered office: 19 High Holborn, LONDON W1 4NB
All data and information provided is for informational purposes only. Hovest John makes no representations as to accuracy, completeness, currentness, or timeliness of the information and will not be liable for any errors, omissions, or delays in this information or any losses, injuries, or damages arising from its display or use.



Travel Heliskiing

For further details, please tick the appropriate box:
Customer: **George W. Bush**
Reference: **4567**

- Val D'Isere **£1405** Grindelwald **£2005**
- St. Anton **£1405** Steamboat **£997**
- Whistler **£997** Terrace **£997**
SPECIAL OFFER!
- Vail **£997**
SPECIAL OFFER!

How to book in a hurry:
Please contact directly via phone or email.

Travel Heliskiing Registered office: 19 High Holborn, LONDON W1 4NB
All data and information provided is for informational purposes only. Hovest John makes no representations as to accuracy, completeness, currentness, or timeliness of the information and will not be liable for any errors, omissions, or delays in this information or any losses, injuries, or damages arising from its display or use.

Figure 115. Skiing brochure – pages 5 and 8

Medical-record for Solomon Grundy

Admitted: Mon 12th September 2011

DoB: 2011-09-11

Next of kin: Tom Thumb

Summary status

- Mon:** 36.9°C 36.4°C The patient appeared generally well and happy and the patient was interviewed by the local constabulary
- Tue:** 36.9°C 37.1°C Signs of sunburn and incipient bruising
- Wed:** 36.8°C 36.1°C The patient took ill, and was unwilling to watch daytime TV
- Thu:** 35.8°C 35.1°C The patient appeared somewhat worse, and worried about the events of the weekend
- Sat:** 38.9°C 38.4°C The patient took a distinct turn for the worse, and appears terminal
- Sun:** Demise of patient 18th

Solomon Grundy - Patient Charts

DDF Healthcare

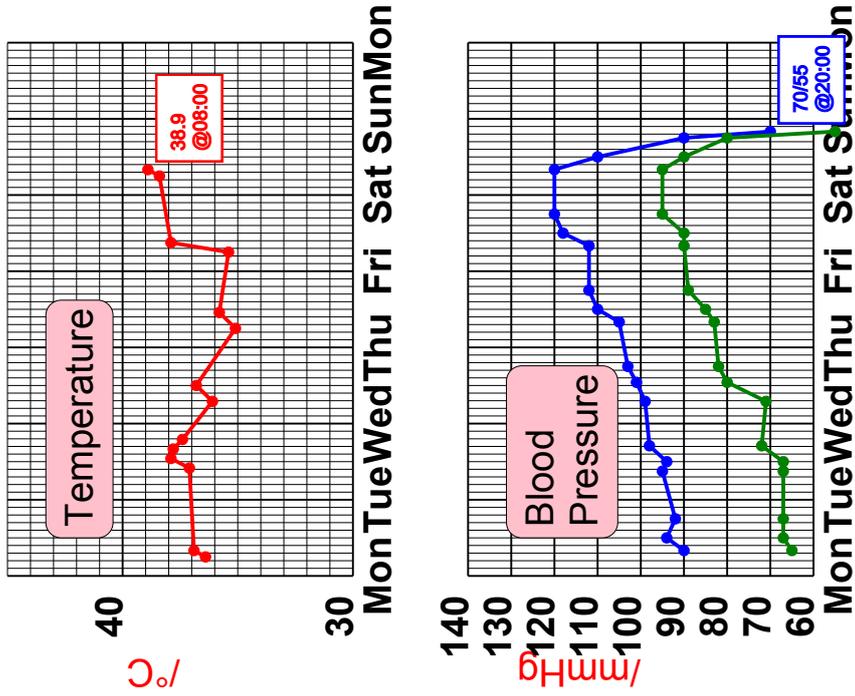


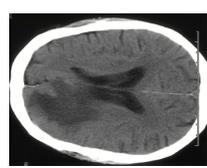
Figure 116. Medical record – pages 1 and 2

Patient notes for Thu 15th

Tests & Reports

- T 35.1 @ 6:00
- T 35.8 @ 11:00
- BP 105/83 @ 8:00
- BP 110/85 @ 12:00
- BP 112/89 @ 18:00
- 5.3 dB dropoff @ 5kHz

Plenty of it



Of his skull

General Notes

The patient took ill, and was unwilling to watch daytime TV

Solomon Grundy - Charges on Account

credit-card:12345678901234

- Mon:** reading temperature @ 6:00 23.45
- 12th test catscan 123.45
- Tue:** reading temperature @ 10:00 10.00
- 13th reading temperature @ 13:00 10.00
- reading temperature @ 16:00 10.00
- reading temperature @ 19:00 10.00
- Thu:** test hearing 10.00
- 15th test blood 37.00
- test catscan 75.00
- Sat:** reading temperature @ 6:00 23.45
- 17th test catscan 123.45
- Sun:** reading ecg 13.45
- 18th report death certificate 23.45

Final Account: 492.70
 Payment from card 12345678901234 **-492.70**
Closing Balance: 0.00

DDF
Healthcare

DDF
Healthcare

Figure 117. Medical record – pages 6 and 10

Appendix B

Advanced Pagination

A basic pagination system was described in Chapter 6, whose heart was a test of whether the next piece to be considered would fit into the remaining space in the container currently being filled. This appendix shows how additional measures can be added to convert this into a practical paginator – the sort that is, for example, capable of organising the main flow for a PhD thesis.

Extending the intrinsic functionality mostly involves adding extra cases to the main test in a suitable order of priority. For example ‘new-page’ is represented by a reserved element (**ddfl:new-page** or **ddfl:new-column** when within a multi-column page) which is detected before the ‘height’ tests, closes the current container and recurses onto the next.

A ‘float’ piece (indicated by the attribute **@ddfl:can-float**) is detected *after* the ‘will fit’ test and if so, the piece is swapped with the next in order and then the whole operation is recalled (if it does fit then there is no need to float, so that test is not encountered; the swap semantics might be indicated by the value of that attribute – indicating whether for example floaters may swap position). Similarly separating ‘space’ between paragraphs that is unnecessary at the head of a column can be indicated by marking blank **svg:rect** separators with **@ddfl:drop-column-start** and dropping them when about to be added to an empty container¹.

An important case for text-based paginations is the paragraph which will not fit in entirely but which can be split into parts. This is supported by testing (again after the ‘will-fit’ test) for the **@ddfl:can-break** property. A piece that will not fit, but has this property, can be broken up into a sequence of pieces (lines are the most common but other breakup schemes could

¹The astute reader will observe that such ‘meta-layout’ pieces could take many forms – pieces marked with the inverse **@ddfl:only-column-start**, which *only* appear at a column head could be used for a ‘... continued’ marker between two components where possible splitting between columns needs to be visually indicated.

be considered) and that sequence tacked on to the beginning of the set of parts and the whole procedure recalled. In this manner a paragraph can be broken across container boundaries.

These are all basically simple operations on a one-dimensional sequence of bins and parts. However text paragraphs flow into two dimensional ‘holes’ provided by the containers, and it is very possible that containers have different widths. It therefore follows that the evaluation of a part *may now depend on the container into which it is targetted*. So how do we achieve that for an effective paginator, whilst still preserving generality? The key is to combine slight modification to the ‘evaluation’ stage for parts in the paginator with the system of acyclic presentational variables. Parts that want to respond to variable width attach to a reserved presentational variable²:

<pre><fo:block font-family="Helvetica" font-size="8pt" text-align="justify"> <ddfl:attribute name="width" select="(?\$text- column-width,30)[1]" />This paragraph wants to flow into an externally defined width, if one is available, else it will use 30 as its default. In this case it is <ddfl:value-of select=" (?\$text-column-width,30)[1]" /> wide</fo:block></pre>	<p>This paragraph wants to flow into an externally defined width, if one is available, else it will use 30 as its default. In this case it is 50 wide</p> <p>This paragraph wants to flow into an externally defined width, if one is available, else it will use 30 as its default. In this case it is 71.25 wide</p>
--	--

Figure 118. A variable width component

Since variable references are interpolated before evaluation, the job of the paginator is to recognise such ‘requests for width’ and arrange a suitable local binding for the variable to the width of the current container. This can be carried out with the evaluation by:

```
<xsl:variable name="piece">
  <xsl:choose>
    <xsl:when test="ddfl:attribute[@name='width']">
      <ddfl:layout function="identity">
        <ddfl:variable name="text-column-width">30</ddfl:variable>
        <xsl:sequence select="$part" />
      </ddfl:layout>
    </xsl:when>
    <xsl:otherwise>
      <xsl:sequence select="$part" />
    </xsl:otherwise>
  </xsl:choose>
</xsl:variable>
<xsl:apply-templates select="$piece" />
```

²Paragraphs are not the only piece that need such treatment – figures that distribute their content horizontally or scale to fit, such as those in this thesis, require a target width. The same mechanism is used.

This technique, basically synthesising small-scale composite computations, either in the documents themselves, or in the implementation, or a combination, has proved to be very powerful. With deeper structures, such as indented lists (**ul/li**) the width can be redefined, within a nested scope:

```
<xsl:template match="ol/li">
  <ddfl:layout function="flow" direction="x" spacing="10">
    <svg:circle cx="4" cy="4" r="4"/>
    <ddfl:variable name="text-column-width" select="$text-column-width - 18"/>
    <xsl:next-match/>
  </ddfl:layout>
</xsl:template>
```

There are three other features worth demonstrating how they can be incorporated fairly easily: content-bearing ‘template’ containers, keep-with-next sections and footnotes.

So far the paginator actually produces a sequence of filled bins – the ‘encapsulate’ action makes a filled component from the pieces selected – what happens to those parts later is unspecified. If we add another level of container ‘argument’ **templates** to the paginator we can accommodate real-life pages³. Templates contain fixed content and to-be-filled containers – when all the containers in the current template are filled, they are inserted into the appropriate position in the template and a finalised page is now emitted. The templates can be considered and marked as a finite sequence (suitable for a fixed-length document) or lazy generators of sequences of pages. Minor modifications to the paginator will accomplish this, and containers are merely parts of a template that are marked for such a role – either by reserved elements (**ddfl:container**) or attribute (**@ddfl:container**) – they are assumed to present a (width and height) extent. These containers will be employed in ‘document order’, so it is certainly possible to have very odd orders of filling and, as discussed above, they need not all be the same width.

But the templates can be extended to include conditionality – declaring circumstances under which they can be employed, by adding guard conditions (as XPath-like statements that operate within the evaluation environment with some reserved ‘variables’, or some specially recognised common forms). For example:

³These are *not* similar to XSLT templates operating in push mode – perhaps an unfortunate choice of nomenclature.

```

layout(paginate)
  template page-range="# eq 1"
    container
      rect x="10" y="10" width="190" height="277"
    template
      container
        rect x="10" y="10" width="90" height="277"
      container
        rect x="110" y="10" width="90" height="277"
  content

```

Figure 119. Conditional page templates

Here there are two templates – the first page has a single column, all remaining pages have two columns. This idea can be extended to even and odd pages or having named pages that are requested by a property on a defined break. What is much more tricky to achieve, and does not have guaranteed deterministic solutions, is content-based template selection, i.e. use template *A* if content *aaaaa* will appear in it.

Keeps (as in **@keep-with-next**) require to be stored up, considered as a group and not emitted until an ‘unkept’ part is encountered. This is done by adding another argument (**keeps**) to the function. (**@keep-with-previous** is best preprocessed to reverse to a canonical use of **@keep-with-next** – this can be achieved simply in a document-borne logical-presentation mapping.)

It is tempting to examine whether entities such as footnotes can be handled by the use of presentational variables, as then it could be a document-borne feature sitting above a pagination. Unfortunately they have a direct and immediate effect on the layout, i.e. they reduce available space in the container, so have to be accounted for. That means the pagination function needs another argument – **footnotes**, to which new notes are added as encountered (such will be marked by attribute **@ddfl:footnote**) and whose total height is accounted for in examining ‘space-remaining’. These are flushed out at the change of container, being added to the base. But the *form* of the footnote is *not* decided by the paginator – they can be *anything* that was created within the presentation description and marked **@ddfl:footnote**.

Appendix C

Compressed Display of XML, XSLT and DDF

The serialization of XML into a human-readable, indented form can result in rather lengthy blocks of text. XSLT, being usually written in such syntax, is often accused of being a verbose language – this may be so, but overlooks the fact that XSLT programs are well-formed XML and therefore consumable and can be created by XSLT. However, the format takes valuable thesis real-estate and can be difficult to read. A common complaint is “I can't see the code for the angle-brackets”.

Hence, where there is no ambiguity, three different types of compressed representation of sections of XML are used, which include XSLT, DDF and layout programs. The first is a purely textual one with the following main features:

- As everything is well-formed XML, depth in the tree is indicated by strict indentation. As such showing closing tags is unnecessary.
- Elements and attributes in some selected namespaces (e.g. XSLT, SVG, DDF) will be shown in a keyed colour without the prefix – the key is consistent throughout the thesis and often displayed where several namespaces are used.
- **xsl:** and **ddfl:** variables and in-element attribute constructors (**xsl:attribute**, **ddl:attribute**) will be shortened to **name** = and **@name** = respectively and coloured appropriately. If the value constructor is an XPath expression (i.e. *@select*) it will be *italicized* – if it is a sequence of children it will be written indented, like any other child-parent relationship.

- The *select* attribute, used on many XSLT elements, the *match* attribute, used on XSLT templates, and the *test* attribute used on choice elements, all being treated as XPath expressions, will be presented as their *value italicized*. When the last element of the XPath is ‘dot’, the shorthand for **current()** (more usually the entire expression is just ‘dot’), it is rendered with the symbol **•** to increase visibility.
- The indirect XPath evaluation **saxon:evaluate()**, which occurs frequently, is abbreviated to **S:E()** within XPath expressions, to save space.
- *Type* declarations (e.g. *as="xs:string"*) are prefixed to the variable or template definition as type casts (e.g. *(xs:string)*).
- Text nodes are displayed in grey.
- Large numbers of attributes can be elided and text nodes shortened. This will be indicated by ellipses.
- **<ddf:layout function="F"/>** constructs are shortened to **layout(F)** and coloured appropriately.

<pre> <xsl:template match="A" mode="m"> <xsl:variable name="var" select="1234"/> <xsl:variable name="parts"> <xsl:apply-templates select="C D" mode="#" #current"/> </xsl:variable> <xsl:for-each select="E"> <xsl:choose> <xsl:when test="count(*) gt 23"> <fo:block > <xsl:value-of select=".,@repeat"/> </fo:block> </xsl:when> <xsl:otherwise> <f/> </xsl:otherwise> </xsl:choose> </xsl:for-each> </xsl:template> </pre>	<pre> match:A mode="m" var=1234 parts= => C D mode="#"current" ∇E: choose when:count(*) gt 23 block val(.,@repeat) otherwise f </pre>
---	--

■ xsl: ■ fo:

Figure 120. Full and compressed XSLT

Figures 121 and 122 show a more extensive example with several namespaces.

```

<ddf:doc >
  <ddf:struct>
    <xsl:template match="/">
      <xsl:sequence select="."/>
    </xsl:template>
  </ddf:struct>
  <ddf:pres>
    <ddfl:layout function="flow" direction="x" spacing="4" overflow="visible">
      <ddfl:variable name="main">
        <svg:svg ddf:layout="flow" encapsulate="background-color:white;stroke:red;shadow:2"
overflow="visible">
          <fo:block font-family="Helvetica" font-size="4" width="50">This is a flow of text and
pieces, some elements of which might be
          variable and hence could change in size</fo:block>
          <svg:ellipse name="ellipse" cx="25" cy="10" rx="25" ry="5" fill="yellow" stroke="black"/>
          <fo:block font-family="Helvetica" font-size="4" width="50">Several pieces are here
and the number could change as a result of
          programmatic selection of variable input data.</fo:block>
          <fo:block name="marginal" font-family="Helvetica" font-size="4" width="50" border-
style="solid" border-width="0.4">But for this piece we want a marginal note.</fo:block>
          <svg:svg ddf:element-type="text-line" font-family="Helvetica" font-size="4" width="
50">And this is some more content that in
          this case follows the targeted block.</svg:svg>
        </svg:svg>
      </ddfl:variable>
      <fo:block font-family="Helvetica" font-size="6" width="50" fill="white" background-color="blue"
border-style="solid" border-width="0.4">
        <ddfl:attribute name="y" select="$main/*[@name='ellipse']/@cy"/>This text should track
the ellipse.</fo:block>
        <ddfl:copy-of select="$main"/>
        <fo:block font-family="Helvetica" font-size="6" font-weight="bold" width="80" background-
color="red" fill="white" border-style="solid" border-width="0.4">
          <ddfl:copy-of select="$main/*[@name='marginal']/@y" as="attribute()"/>This is the
marginal note, which follows the start of the source.</fo:block>
        </ddfl:layout>
      </ddf:pres>
    </ddf:doc>

```

■ ddf: ■ xsl: ■ ddf: ■ svg: ■ fo:

Figure 121. Full DDF



Figure 122. Compressed display of Figure 121

The second and third are graphical tree forms, based on the tree layout and using a sequence of substitutive captions or simple shapes. It is employed primarily to emphasise the structure of the entities being discussed. The tree may be oriented either vertically or horizontally. The colour scheme employed is identical to that used for the compressed textual form (it comes from the same code within a template).

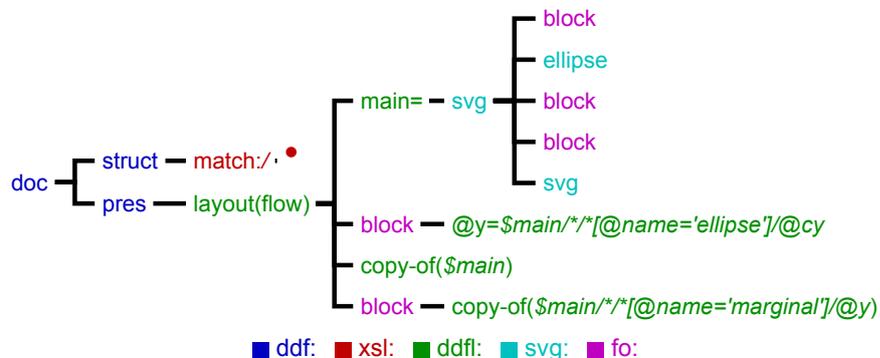


Figure 123. Horizontal tree display of Figure 121

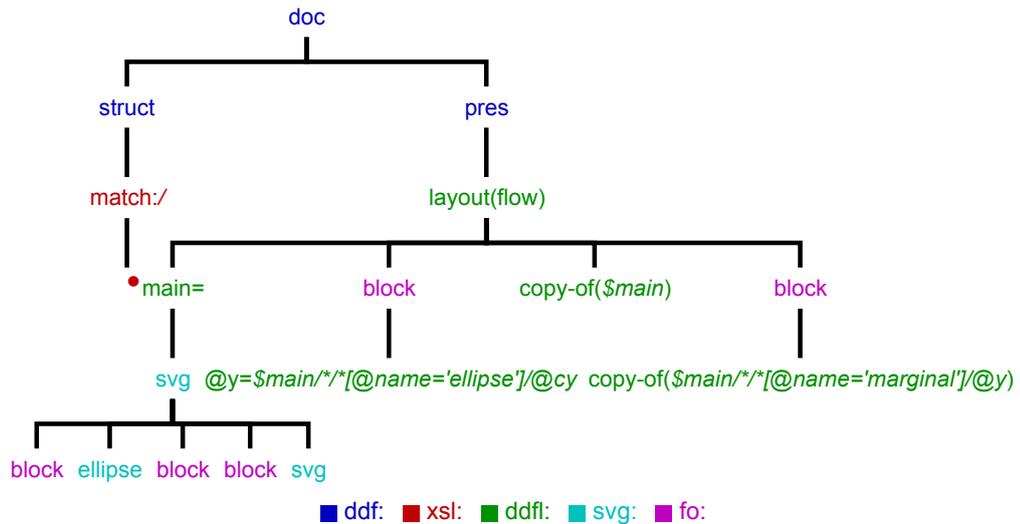


Figure 124. Vertical tree display of Figure 121

In one form many of the same contractions (variables and attribute assignment) are used. In the most compressed form, a simple small coloured shape indicates the presence of an element in a particular namespace – this is used for display of the structure of very large trees. As this is sometimes used to illustrate final presentations, text blocks **svg:svg[@ddfl:element-type='text-block']**, which are subtrees with potentially a very large number of descendants, are collapsed to a single shape.

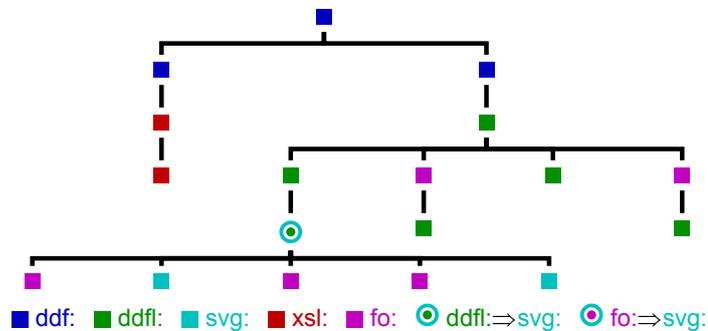


Figure 125. Highly-compressed tree display of Figure 121

There are a few graphical substitutions for common cases to increase understanding. An **svg:svg** element which is decorated with **@ddfl:layout** and an **svg:svg** that was generated from an **fo:block** are both shown as two-color circles.

The full details of the substitutions are shown in the following table:

<ddfl:layout function="layout-type"/>	layout(layout-type)
<ddfl:copy-of select="expression"/>	copy-of(expression)
<ddfl:attribute name="name" select="expression"/>	@name=expression
<ddfl:variable name="name" select="expression"/>	name=expression
<ddfl:for-each select="expression"/>	∀:expression
<ddfl:constraints layout="constraints" parts="parts"/>	constraints:constraints for: parts
<xsl:variable as="type" name="name" select="expression"/>	(type) name=expression
<xsl:param as="type" name="name" select="expression"/>	(type) param:name=expression
<xsl:for-each select="expression"/>	∀:expression :
<xsl:for-each-group select="expression" group-by="discriminant"/>	group:expression by:discriminant
<xsl:for-each-group select="expression" group-adjacent="discriminant"/>	group:expression adjacent: discriminant
<xsl:for-each-group select="expression" starting-with="discriminant"/>	group:expression starting-with: discriminant
<xsl:for-each-group select="expression" ending-with="discriminant"/>	group:expression ending-with: discriminant
<xsl:attribute name="name" select="expression"/>	@name=expression
<xsl:sequence select="expression"/>	expression
<xsl:value-of select="expression"/>	val(expression)
<xsl:copy-of select="expression"/>	copy-of(expression)
<xsl:template match="pattern"/>	match:pattern
<xsl:apply-templates select="expression"/>	⇒ expression
<xsl:with-param name="name" select="expression"/>	param(name)=expression
<xsl:if test="expression"/>	if:expression
<xsl:choose />	choose
<xsl:when test="expression"/>	when:expression
<xsl:otherwise />	otherwise

Figure 126. Substitutions for condensed XML

Appendix D

Employment of Namespaces

Much of the flexibility of DDF comes from the use of several different programmatic models combined into large trees through interspersed namespaces. The major namespaces used in the framework are described here – several are unique to DDF, others indicate the use of syntax and semantics of XML-based standards.

DDF namespaces

cmap: <http://hpl.hp.com/vda/2006/doc>

Context map elements for tracing external references.

ddf: <http://hpl.hp.com/ddf/2005/doc>

Main DDF namespace, describing major document structures and additional (compiler) directives.

ddf: <http://hpl.hp.com/ddf/2005/layout>

DDF Layout descriptions. Combinator nodes and other parameters.

doc: <http://hpl.hp.com/vda/2006/doc>

Documentation elements and attributes.

tree: <http://mytree>

Intermediate data structures in calculating complex tree layouts.

XSL: MyXSL

XSLT code to be ‘quoted’ within other XSLT executable structures.

XSLT: `output.xml`

Compiler quoted XSL – to create basic XSLT (**xsl:**) in the output of the compiler.

XSLT2: output2.xml **XSLT3:** output3.xml

Second and third-stage quoted XSL in the compiler. To build quoted XSLT (**XSLT:**) during successively removed executions. Needed in partial binding and evaluation.

Standard namespaces

fn: <http://www.w3.org/2005/02/xpath-functions>

Basic XPath functions – usually implicit for functions within XPath expressions, i.e. **count()** is equivalent to **fn:count()**.

fo: <http://www.w3.org/1999/XSL/Format>

Formatting Objects. Text block formatting.

math: <http://exslt.org/math>

Mathematical functions.

svg: <http://www.w3.org/2000/svg>

Scalable Vector Graphics. Presentational graphical results.

xi: <http://www.w3.org/2001/XInclude>

Inclusion of XML content from within other XML trees, via URL reference

xlink: <http://www.w3.org/1999/xlink>

Xlink – resource pointers, either local or as URL.

xs: <http://www.w3.org/2001/XMLSchema>

Data structure schemas used to indicate type information (**as="xs:double"**) on variables and constructors. The current implementation only uses this for the base and atomic types (e.g. **xs:string***, **xs:anyURI**).

xsl: <http://www.w3.org/1999/XSL/Transform>

XSL Transforms.

Proprietary namespaces

saxon: <http://saxon.sf.net/>

The Saxon XLST implementation engine. Used to attach to a small number of extension functions: **saxon:serialize()** and **saxon:evaluate(xpath)** (The latter is used extensively to implement presentational variables, selections within embedded inclusions and tree displays, when dynamic XPath expressions have to be evaluated.)

Appendix E

Construction of the Thesis

This thesis is, of course, built from the technology described therein, as a specific binding of some data to a variable data document. In this case the variable document is a ‘thesis template’, that anticipates a source document described principally in XHTML, with some extensions for figures, embedded layouts, citations and so forth. Much of the functionality of this document is supplied by included libraries, such as a generic conversion from XHTML to **ddfl:**, **fo:** and **svg:** elements – this library has been used for all publications for DDF such as those for the ACM.

The thesis source is a set of XHTML **section** chapters and appendices plus an XML-based reference collection. Headings (**h**) within sections can be identified (e.g. **h ddf:id="thesisConstruction"**) and used as targets for cross-references (**<ref>priorArt..**) – these elements are expanded as much as possible during the generation of the document structural layer (section numbering...) and cast as hyper-links using the **<a href="#..** form around the structural element. Page references are resolved in a late layout pass. Similar methods are used for citations (**<cite>Lumley2010..**).

Meta-document elements (contents, list of figures) are built along with and inserted into the document structure and thereafter become indistinguishable from other cross-references. A document outline tree is also built and copied into the eventual layout (as an **svg:desc** element) where its references will be resolved during PDF generation.

Figures within are described as **figure** elements, which usually have caption properties and directives for arrangement of their contents (columns, spacing, size, scaling etc.) – the children can take many forms, such as embedded images, DDF document quotations, PDF pages, displayed XML trees and layout declarations that should be evaluated. Some of these figure

constructs (such as the ‘zoom in’) are supported by macro libraries which exploit presentational variables extensively. The more extensive figures (e.g. the large examples) are graphical embeddings of the resulting SVG or PDF of evaluated test documents¹. If in doubt, the figure was computed during the thesis construction – especially if it is in tree form.

Document layout occurs in a two-pass form, as hinted at in Chapter 6, to enable textual page references to be sized correctly.

The generation of the final PDF format is a balance between processing with XSLT transforms and Java code. The presentational SVG in the DDF is ‘regularized’ with several XSLT passes (ensuring that text elements have complete fonting, converting all dimensions into canonical form, removing foreign elements...) and then passed with additional arguments (a list of required fonts, the outline tree...) to a fast Java-based process that uses the iText library[59]. This not only builds the correct ‘drawing stream’ but includes all necessary external resources (images, PDF pages and fonts, which are embedded) as well as placing appropriate hyper-links (with hovers) on the pages and adding the document outline to the PDF.

The thesis structural XML tree, when fully expanded on 26th May 2012 @ 14:56, has 19382 elements, 42796 attributes and 63735 words. The processing of the entire document takes about 6 minutes on a 3GHz quad-core Windows7 machine. The vast majority of this time (5:46 mins) is taken up with resolving the layout; other approximate times are 2 seconds for document compilation, 10 seconds for XSLT execution and 20 seconds for PDF.

E.1 Code base statistics

It is possible to examine and contrast the implementation code base for the technologies described in this thesis. For XSLT and similar XML-based software, a suitable measure of the complexity is the number and distribution of element and attribute nodes in the program-defining structures². Here are some figures for three important components:

- The DDF compiler.

¹Using PDF obviates the problem of internal image reference, but no longer can XPath searching be used to find appropriate nodes for illustrative purposes.

²Interestingly the number of source lines is roughly similar to the sum of element and attribute numbers – this is explained by a filled element needing two lines (opening and closing tags) and assumptions of ~ 1 attribute per element and very little ‘naked’ text.

- The layout processor. Statistics cover *all* the agents currently implemented, many of which are experimental³.
- The DDF thesis template that is actually used to process this document you are reading. This includes imports from 11 other files.

Measure	DDF Compiler	Layout Processor	Thesis Template
Files	10	36	12
Match Templates	92	366	189
Functions & Named Templates	31	216	25
Elements	902	6535	1378
Attributes	1417	11208	2378

³E.g. the TALL experiment on topologically abstract layout from U. Bologna[15]

REFERENCES

Papers, Books and Journals

- [1] Ager, M., Danvy, O. and Rohde, H. Fast partial evaluation of pattern matching in strings. In *Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 3–9, ACM, 2003.
- [2] André, J., Furuta, R. and Quint, V. *Structured Documents*. Cambridge University Press, 1989.
- [3] Badros, G. et al. A constraint extension to scalable vector graphics. In *Proc. 10th World Wide Web Conference, Hong Kong, 2001*.
- [4] Badros, G., Borning, A. and Stuckey, P. The Cassowary linear arithmetic constraint solving algorithm. In *ACM Transactions on Computer-Human Interaction (TOCHI)*, Vol8 (4), pages 267–306, 2001.
- [5] Bagley, S. COG Extractor. In *Proceedings of the 2006 ACM symposium on Document engineering*, pages 31–31, ACM, 2006.
- [6] Bagley, S. and Brailsford, D. Page composition using PPML as a link-editing script. In *Proceedings of the 2004 ACM symposium on Document engineering*, pages 134–136, ACM, 2004.
- [7] Bagley, S., Brailsford, D. and Hardy, M. Creating reusable well-structured PDF as a sequence of component object graphic (COG) elements. In *DocEng '03: Proceedings of the 2003 ACM symposium on Document engineering*, pages 58–67, ACM, 2003.
- [8] Bagley, S., Brailsford, D. and Ollis, J. Extracting reusable document components for variable data printing. In *Proceedings of the 2007 ACM symposium on Document engineering*, pages 44–52, ACM, 2007.
- [9] Balinsky, H. and Pilu, M. Evaluating interface aesthetics: a measure of symmetry. In *Digital Publishing, Proc. of SPIE-IS&T Electronic Imaging, Vol 6076*, 2006. <http://www.hpl.hp.com/techreports/2006/HPL-2006-29.html>

- [10] Burchett, K., Cooper, G. and Krishnamurthi, S. Lowering: a static optimization technique for transparent functional reactivity. In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 71–80, ACM, 2007.
- [11] Chao, H., Gabbur, P. and Wiley, A. Preserving the aesthetics during non-fixed aspect ratio scaling of the digital border. In *Proceedings of the 2007 ACM symposium on Document engineering*, pages 144–146, ACM, 2007.
- [12] Chao, H., Zhang, X. and Tretter, D. Structured layout for resizable background art. In *Proceedings of the 1st international workshop on Interactive multimedia for consumer electronics*, pages 67–72, ACM, 2009.
- [13] Damera-Venkata, N., Bento, J. and O'Brien-Strain, E. Probabilistic document model for automated document composition. In *Proceedings of the 11th ACM symposium on Document engineering*, pages 3–12, ACM, 2011.
- [14] Dhamdhere, D. E-path_PRE: partial redundancy elimination made easy. In *SIGPLAN Not.*, Vol37, no8, pages 53–65, ACM, 2002.
- [15] Di Iorio, A. et al. Higher Level Layout through Topological Abstraction. In *Proceedings of the 2008 ACM symposium on Document engineering*, 2008.
- [16] Dong, C. and Bailey, J. Static analysis of XSLT programs. In *Proceedings of the 15th Australasian database conference*, Vol27, pages 151–160, 2004.
- [17] Feiner, K. A grid-based approach to automating display layout. In *Proceedings on Graphics interface '88*, pages 192–197, Canadian Information Processing Society, 1988.
- [18] Findler, R. and Flatt, M. Slideshow: functional presentations. In *J. Funct. Program.*, Vol16, pages 583–619, Cambridge University Press, 2006.
- [19] Flesca, S., Furfaro, F. and Masciari, E. On the minimization of XPath queries. In *J. ACM*, Vol55, pages 2:1–2:46, ACM, 2008.
- [20] Furuta, R. Important papers in the history of document preparation systems: basic sources. In *Electronic Publishing*, Vol5, pages 19–44, 1992 .
- [21] Gallesio, E. and Serrano, M. Scribe: a functional authoring language. In *J. Funct. Program.*, Vol15, pages 751–770, Cambridge University Press, 2005.
- [22] Gange, G. et al. Optimal automatic table layout. In *Proceedings of the 11th ACM symposium on Document engineering*, pages 23–32, ACM, 2011.
- [23] Giannetti, F. A Multi-format Variable Data Template Wrapper extending PODi's PPML-T Standard. In *Proceedings of the 2007 ACM symposium on Document engineering*, 2007.
- [24] Giannetti, F. Paginate dynamic and web content. In *Proceedings of the 11th ACM symposium on Document engineering*, pages 143–152, ACM, 2011.

- [25] Giannetti, F. XSL-FO 2.0: automated publishing for graphic documents. In *DocEng '09: Proceedings of the 9th ACM symposium on Document engineering*, pages 245–246, ACM, 2009.
- [26] Giannetti, F. et al. High performance XSL-FO rendering for variable data printing. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 811–817, ACM, 2006.
- [27] Glushko, R. and McGrath, T. *Document Engineering: Analyzing and Designing Documents for Business Informatics and Web Services*. The MIT Press, 2005 .
- [28] Goldenberg, E. *Automatic layout of variable-content print data*. HP Laboratories Technical Report, 2002. <http://www.hpl.hp.com/techreports/2002/HPL-2002-286.pdf>
- [29] Gottlob, G., Koch, C. and Pichler, R. Efficient algorithms for processing XPath queries. In *ACM Trans. Database Syst.*, Vol30, pages 444–491, ACM, 2005.
- [30] Graf, H., Neurohr, S. and Goebel, R. YPPS—A constraint-based tool for the pagination of yellow-page directories. In *Proceedings of the KI-96 Workshop on Declarative Constraint Programming*, pages 87–97, 1996.
- [31] Hansen, B. A function-based formatting model. In *Electronic Publishing*, Vol3 (1), pages 3–28, 1990.
- [32] Harrington, S. et al. Aesthetic measures for automated document layout. In *DocEng '04: Proceedings of the 2004 ACM symposium on Document engineering*, pages 109–111, ACM, 2004.
- [33] Harrington, S. et al. Expression of document structure in automatic layout. In *Digital Publishing, Proc. of SPIE-IS&T Electronic Imaging, Vol 6076*, 2006.
- [34] Heydon, A. and Nelson, G. *The Juno-2 constraint-based drawing editor*. DEC SRC Technical Report 131a, 1994.
- [35] Hughes, J. The Design of a Pretty-printing Library. In *Advanced Functional Programming*, pages 53–96, Springer, 1995.
- [36] Hurst, N. and Marriott, K. Approximating text by its area. In *DocEng '07: Proceedings of the 2007 ACM symposium on Document engineering*, pages 147–150, ACM, 2007.
- [37] Hurst, N. and Marriott, K. Satisficing scrolls: a shortcut to satisfactory layout. In *DocEng '08: Proceeding of the eighth ACM symposium on Document engineering*, pages 131–140, ACM, 2008.
- [38] Hurst, N., Li, W. and Mariott, K. Review of Automatic Document Formatting. In *Proceedings of the 2009 ACM symposium on Document engineering*, 2009.

- [39] Hurst, N., Marriott, K. and Moulder, P. Dynamic approximation of complex graphical constraints by linear constraints. In *Proceedings of the 15th annual ACM symposium on User interface software and technology*, pages 191–200, ACM Press, 2002.
- [40] Hurst, N., Marriott, K. and Moulder, P. Minimum sized text containment shapes. In *DocEng '06: Proceedings of the 2006 ACM symposium on Document engineering*, pages 3–12, ACM, 2006.
- [41] Hurst, N., Marriott, K. and Moulder, P. Toward tighter tables. In *DocEng '05: Proceedings of the 2005 ACM symposium on Document engineering*, pages 74–83, ACM, 2005.
- [42] Jacobs, C. et al. *Adaptive Grid-Based Document Layout*. Vol22, pages 838 – 847, In *ACM Transactions on Graphics*, 2003.
- [43] Jones, N. An introduction to partial evaluation. In *ACM Comput. Surv.*, Vol28, pages 480–503, ACM, 1996.
- [44] Kahl, W. Beyond Pretty-Printing: Galley Concepts in Document Formatting Combinators. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, pages 76–90, Springer-Verlag, 1998.
- [45] Kay, M. *XSLT 2.0 and XPath 2.0, 4th Edition*. Wiley, Indianapolis, IN, 2008.
- [46] Kay, M. XSLT in the Browser. In *Proceedings of XML Prague 2011*, pages 125 – 134, 2011. <http://www.xmlprague.cz/2011/files/xmlprague-2011-proceedings.pdf>
- [47] Kenji, M. and Hiroyuki, S. Static optimization of XSLT stylesheets: template instantiation optimization and lazy XML parsing. In *Proceedings of the 2005 ACM symposium on Document engineering*, pages 55–57, ACM, 2005.
- [48] Kernighan, B. A TROFF Tutorial, Unix Version 7 manual. 1978.
- [49] King, P., Schmitz, P. and Thompson, S. Behavioral reactivity and real time programming in XML: functional programming meets SMIL animation. In *Proceedings of the 2004 ACM symposium on Document engineering*, pages 57–66, ACM, 2004.
- [50] Kingston, J. The design and implementation of the Lout document formatting language. In *Softw. Pract. Exper.*, Vol23, pages 1001–1041, John Wiley & Sons, Inc., 1993.
- [51] Knuth, D. *TeX – the program*. Addison-Wesley Pub. Co., 1986.
- [52] Knuth, D. and Plass, M. Breaking paragraphs into lines. In *Software---Practice and Experience*, 11(11), pages 1119–1184, 1982.
- [53] Kong, J., Zhang, K. and Zeng, X. Spatial graph grammars for graphical user interfaces. In *ACM Trans. Comput.-Hum. Interact.*, Vol13, pages 268–307, ACM, 2006.

- [54] Krakovsky, M. *All the News That's Fit for You*. Vol54, no6, pages 20–21, In *Communications of the ACM*, 2011.
- [55] Lamport, L. *LaTeX: A Document Preparation System* (2nd Edition). Addison-Wesley Professional, 1994.
- [56] Launchbury, J. A Strongly-Typed Self-Applicable Partial Evaluator. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 145–164, Springer-Verlag, 1991.
- [57] Lin, X. Active layout engine: Algorithms and applications in variable data printing. In *Comput. Aided Des.*, Vol38, pages 444–456, Butterworth-Heinemann, 2006.
- [58] Lok, S., Feiner, S. and Ngai, G. Evaluation of visual balance for automated layout. In *Proceedings of the 9th international conference on Intelligent user interfaces*, pages 101–108, ACM, 2004.
- [59] Lowagie, B. *iText in Action*. Manning Publications, 2007.
- [60] Lumley, J. Automated Extensible XML Tree Diagrams. In *Proceedings of the 2009 ACM symposium on Document engineering*, 2009. <http://www.hpl.hp.com/techreports/2009/HPL-2009-137.pdf>
- [61] Lumley, J. Pre-evaluation of Invariant Layout in Functional Variable-Data Documents. In *Proceedings of the 2010 ACM symposium on Document engineering*, 2010.
- [62] Lumley, J., Gimson, R. and Rees, O. A Framework for Structure, Layout & Function in Documents. In *Proceedings of the 2005 ACM symposium on Document engineering*, 2005. <http://www.hpl.hp.com/techreports/2005/HPL-2005-95R1.pdf>
- [63] Lumley, J., Gimson, R. and Rees, O. Configurable Editing of XML-based Variable-Data Documents. In *Proceedings of the 2008 ACM symposium on Document engineering*, 2008. <http://www.hpl.hp.com/techreports/2008/HPL-2008-53.pdf>
- [64] Lumley, J., Gimson, R. and Rees, O. Endless Documents: A Publication as a Continual Function. In *Proceedings of the 2007 ACM symposium on Document engineering*, 2007. <http://www.hpl.hp.com/techreports/2007/HPL-2007-111R1.pdf>
- [65] Lumley, J., Gimson, R. and Rees, O. Extensible Layout in Functional Documents. In *Digital Publishing, Proc. of SPIE-IS&T Electronic Imaging, Vol 6076*, 2006. <http://www.hpl.hp.com/techreports/2005/HPL-2005-223.pdf>
- [66] Lumley, J., Gimson, R. and Rees, O. Resolving Layout Interdependency with Presentational Variables. In *Proceedings of the 2006 ACM symposium on Document engineering*, 2006. <http://www.hpl.hp.com/techreports/2006/HPL-2006-107.pdf>
- [67] Macdonald, A. *Progressive Document Evaluation*. PhD Thesis, University of Nottingham, 2008.

- [68] Macdonald, A. et al. Speculative document evaluation. In *DocEng '07: Proceedings of the 2007 ACM symposium on Document engineering*, pages 56–58, ACM, 2007.
- [69] Macdonald, A., Brailsford, D. and Bagley, S. Encapsulating and manipulating component object graphics (COGs) using SVG. In *Proceedings of the 2005 ACM symposium on Document engineering*, pages 61–63, ACM, 2005.
- [70] Macdonald, A., Brailsford, D. and Lumley, J. Evaluating Invariances in Document Layout Functions. In *Proceedings of the 2006 ACM symposium on Document engineering*, 2006.
- [71] Marlet, R., Thibault, S. and Consel, C. Efficient Implementations of Software Architectures via Partial Evaluation. In *Automated Software Engg.*, Vol6, pages 411–440, Kluwer Academic Publishers, 1999.
- [72] Marriott, K., Moulder, P. and Hurst, N. Automatic float placement in multi-column documents. In *DocEng '07: Proceedings of the 2007 ACM symposium on Document engineering*, pages 125–134, ACM, 2007.
- [73] McCarthy, J. Recursive functions of symbolic expressions and their computation by machine, Part I. In *Commun. ACM*, Vol3, pages 184–195, ACM, 1960.
- [74] McCormack, C., Marriott, K. and Meyer, B. *Adaptive layout using one-way constraints in SVG*. 2004. <http://www.svgopen.org/2004/papers/ConstraintSVG/>
- [75] Nebeling, M. et al. Adaptive layout template for effective web content presentation in large-screen contexts. In *Proceedings of the 11th ACM symposium on Document engineering*, pages 219–228, ACM, 2011.
- [76] Noga, M., Schott, S. and Löwe, W. Lazy XML processing. In *DocEng '02: Proceedings of the 2002 ACM symposium on Document engineering*, pages 88–94, ACM, 2002.
- [77] Novatchev, D. Higher-Order Functional Programming with XSLT 2.0 and FXSL. In *Extreme Markup Languages Conference, Montreal*, , 2006.
- [78] Ollis, J. *Optimised Editing of Variable Data Documents via Partial Re-Evaluation*. PhD Thesis, University of Nottingham, 2011.
- [79] Ollis, J., Bagley, S. and Brailsford, D. Tracking sub-page components in document workflows. In *Proceeding of the eighth ACM symposium on Document engineering*, pages 86–89, ACM, 2008.
- [80] Ollis, J., Brailsford, D. and Bagley, S. Optimized reprocessing of documents using stored processor state. In *Proceedings of the 10th ACM symposium on Document engineering*, pages 135–138, ACM, 2010.
- [81] Peroni, S. and Vitali, F. Annotations with EARMARK for arbitrary, overlapping and out-of order markup. In *Proceedings of the 9th ACM symposium on Document engineering*, pages 171–180, ACM, 2009.

- [82] Piccoli, R. et al. A novel physics-based interaction model for free document layout. In *Proceedings of the 11th ACM symposium on Document engineering*, pages 153–162, ACM, 2011.
- [83] Pinkney, A., Bagley, S. and Brailsford, D. Reflowable Documents Composed from Pre-rendered Atomic Components. In *Proceedings of the 2011 ACM symposium on Document engineering*, pages 163–166, 2011.
- [84] Purvis, L. et al. Document formatting: Creating personalized documents: an optimization approach. In *Proceedings of the 2003 ACM symposium on Document engineering*, 2003.
- [85] Quint, V. and Vatton, I. Editing with Style. In *Proceedings of the 2007 ACM symposium on Document engineering*, 2007.
- [86] Quint, V. and Vatton, I. Techniques for authoring complex XML documents. In *DocEng '04: Proceedings of the 2004 ACM symposium on Document engineering*, pages 115–123, ACM, 2004.
- [87] Reid, B. *Scribe: A Document Specification Language and its Compiler*. PhD Thesis, Carnegie-Mellon University, Pittsburgh PA, 1981.
- [88] Roisin, C. and Vatton, I. Merging logical and physical structures in documents. In *Electronic Publishing*, Vol6 (4), pages 327–337, 1993.
- [89] Rutledge, L. et al. Generating presentation constraints from rhetorical structure. In *Proceedings of the eleventh ACM on Hypertext and hypermedia*, pages 19–28, ACM, 2000.
- [90] Schott, S. and Noga, M. Lazy XSL transformations. In *DocEng '03: Proceedings of the 2003 ACM symposium on Document engineering*, pages 9–18, ACM, 2003.
- [91] Silva, A. et al. Support for arbitrary regions in XSL-FO. In *DocEng '05: Proceedings of the 2005 ACM symposium on Document engineering*, pages 64–73, ACM, 2005.
- [92] Stayton, B. *DocBook XSL: The Complete Guide*. Sagehill Enterprises, 2007.
- [93] Thompson, S., King, P. and Schmitz, P. Declarative extensions of XML languages. In *Proceedings of the 2007 ACM symposium on Document engineering*, pages 89–91, ACM, 2007.
- [94] Villard, L. and Layaida, N. An Incremental XSLT Transformation Processor for XML Document Manipulation. In *Proc. 11th World Wide Web Conference, Honolulu*, 2002.
- [95] Vion-Dury, J. A generic calculus of XML editing deltas. In *Proceedings of the 11th ACM symposium on Document engineering*, pages 113–120, ACM, 2011.
- [96] Wadler, P. A prettier printer. In *The Fun of Programming*, pages 223 – 244, Palgrave Macmillan, 2003.

- [97] Walsh, N. *Literate Programming in XML*. 2002. <http://nwalsh.com/docs/articles/xml2002/lp/paper.html>

Software

- [98] Adobe *Adobe InDesign*. 2011. <http://www.adobe.com/products/indesign.html>
- [99] Apache Software Foundation *Apache PDFBox – Java PDF Library*. 1999. <http://pdfbox.apache.org>
- [100] Apache XML Graphics Project *Apache FOP (Formatting Objects Processor)*. 1999. <http://xmlgraphics.apache.org/fop/>
- [101] Bitstream Inc. *Pageflex*. 2008. <http://www.pageflex.com/>
- [102] CatBase *CatBase database publishing*. 2011. <http://www.catbase.com/>
- [103] HPExstream *Dialogue*. 2010. <http://welcome.hp.com/country/uk/en/prodserv/software/eda/products/dialogue.html>
- [104] HPExstream *DialogueLive*. 2008. <http://welcome.hp.com/country/uk/en/prodserv/software/eda/products/dialogue-live.html>
- [105] Kay, M. *Saxonica: XSLT and XQuery Processing*. 2005. <http://www.saxonica.com/>
- [106] Novatchev, D. *FXSL – the Functional Programming Library for XSLT*. 2006. <http://fxsl.sourceforge.net/>
- [107] Quark *QuarkXPress*. 2011. <http://www.quark.com/Products/QuarkXPress/>

Standards

- [108] Adobe Systems Incorporated *PDF Reference version 1.6, 5th Edition*. Adobe Press, 2004.
- [109] Adobe Systems Incorporated. *PostScript language tutorial and cookbook*. Addison Wesley, 1985.
- [110] ISO, International Standards Organisation *Open Document Architecture*. 1994. <http://www.iso.org/>
- [111] International Digital Publishing Forum *EPUB Content Documents 3.0*. 2008. <http://idpf.org/epub/30/spec/epub30-contentdocs.html>

- [112] OASIS *RELAX NG Specification*. 2001. <http://relaxng.org/spec-20011203.html>
- [113] PODi, Print On Demand Initiative *Personalized Print Markup Language (PPML) Version 2.0*. 2002. <http://www.podi.org>
- [114] W3C, World Wide Web Consortium *CSS Template Layout Module*. 1999. <http://www.w3.org/TR/css3-layout>
- [115] W3C, World Wide Web Consortium *Cascading Style Sheets, Level 1*. 1999. <http://www.w3.org/TR/CSS1>
- [116] W3C, World Wide Web Consortium *Document Type Definition*. 2008. <http://www.w3.org/TR/REC-xml/#dt-doctype>
- [117] W3C, World Wide Web Consortium *Extensible Stylesheet Language (XSL)*. 2001. <http://www.w3.org/TR/xsl/>
- [118] W3C, World Wide Web Consortium *HTML 4.01 Specification*. 1999. <http://www.w3.org/TR/1999/REC-html401-19991224/>
- [119] W3C, World Wide Web Consortium *Scalable Vector Graphics (SVG) 1.1 Specification*. 2003. <http://www.w3.org/TR/SVG/>
- [120] W3C, World Wide Web Consortium *The Secret Origin of SVG*. 2003. http://www.w3.org/Graphics/SVG/WG/wiki/Secret_Origin_of_SVG
- [121] W3C, World Wide Web Consortium *XHTML™ 2.0*. 2006. <http://www.w3.org/TR/xhtml2/>
- [122] W3C, World Wide Web Consortium *XML Inclusions (XInclude) Version 1.0 (Second Edition)*. 2006. <http://www.w3.org/TR/xhtml2/>
- [123] W3C, World Wide Web Consortium *XML Path Language (XPath) 2.0*. 2007. <http://www.w3.org/TR/xpath20/>
- [124] W3C, World Wide Web Consortium *XML Schema Part 0: Primer Second Edition*. 2004. <http://www.w3.org/TR/xmlschema-0/>
- [125] W3C, World Wide Web Consortium *XProc: An XML Pipeline Language*. 2010. <http://www.w3.org/TR/xproc/>
- [126] W3C, World Wide Web Consortium *XQuery 1.0: An XML Query Language (Second Edition)*. 2010. <http://www.w3.org/TR/xquery/>
- [127] W3C, World Wide Web Consortium *XQuery 3.0: An XML Query Language*. 2011. <http://www.w3.org/TR/xquery-30/>
- [128] W3C, World Wide Web Consortium *XSL Transformations (XSLT) Version 2.0*. 2007. <http://www.w3.org/TR/xslt20/>

- [129] W3C, World Wide Web Consortium *XSL Transformations (XSLT) Version 2.1*. 2010.
<http://www.w3.org/TR/xslt-21/>
- [130] Walsh, N. and Muellner, L. *DocBook: The Definitive Guide*. O'Reilly & Associates, 1999.