

**Automatic Generation of XSLT by Simultaneous Editing**

by

Brian A. Stube

Submitted to the Department of Electrical Engineering and Computer Science  
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2004

© Brian A. Stube, MMIV. All rights reserved.

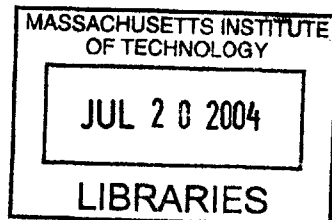
The author hereby grants to MIT permission to reproduce and distribute publicly  
paper and electronic copies of this thesis document in whole or in part.

n

Author .....  
Department of Electrical Engineering and Computer Science  
May 20, 2004

Certified by .....  
Robert C. Miller  
Assistant Professor  
Thesis Supervisor

Accepted by .....  
Arthur C. Smith  
Chairman, Department Committee on Graduate Students



**BARKER**



# Automatic Generation of XSLT by Simultaneous Editing

by

Brian A. Stube

Submitted to the Department of Electrical Engineering and Computer Science  
on May 20, 2004, in partial fulfillment of the  
requirements for the degree of  
Master of Engineering in Electrical Engineering and Computer Science

## Abstract

The proliferation of XML during recent years has been aided by an array of powerful companion tools. One such tool, XSL Transformations (XSLTs), has played an important role in XML's adoption, facilitating interoperability by defining general transformations on XML documents. However, despite XSLT's power and flexibility, the transformations can be hard to define, often requiring recursion to perform simple operations. Further, writing XSLTs requires learning its transformation language, distracting from the primary focuses of the input and output.

This thesis focuses on streamlining the generation of XSLTs through the use of a programming-by-demonstration (PBD) interface. Instead of directly defining an XSLT, a user begins with the input XML document, converting it to the desired output through text editing. Based on the input XML document, the resulting output document, and the sequence of user edits, an XSLT definition will then be automatically generated. To overcome the hindrance of repetitive editing this thesis relies heavily on simultaneous editing as implemented in LAPIS, defined as the editing of text with multiple simultaneous cursors inferred from positive and negative examples. In addition to reducing redundant actions, simultaneous editing provides critical hints into the structure of the desired transformation. Models that have attempted to generate XSLTs based solely on input and output examples have met challenges, but aided with the simultaneous editing information, successful XSLT generation is possible.

Thesis Supervisor: Robert C. Miller  
Title: Assistant Professor



# Contents

<b>1</b>	<b>Introduction</b>	<b>13</b>
1.1	XSLT by Demonstration . . . . .	15
1.2	Simultaneous Editing . . . . .	16
1.3	LAPIS . . . . .	17
1.4	Outline . . . . .	19
<b>2</b>	<b>Related Work</b>	<b>21</b>
2.1	Current XSLT Tools . . . . .	21
2.2	Programming By Demonstration . . . . .	22
<b>3</b>	<b>User Interface</b>	<b>25</b>
3.1	Generating an XSLT by Demonstration . . . . .	25
3.2	The Two-Pane Solution . . . . .	27
3.3	Working with Multiple Cursors . . . . .	29
3.4	Generalizing Selections . . . . .	30
3.5	Generating the XSLT . . . . .	31
<b>4</b>	<b>Implementation</b>	<b>33</b>
4.1	Region Sets . . . . .	33
4.2	Specializing the Inference Engine for XML . . . . .	34
4.3	Inference Backed Windows . . . . .	34
4.4	Document Versions and Coordinate Maps . . . . .	35
4.5	Paste Tree . . . . .	37
4.5.1	Internal Nodes . . . . .	38
4.5.2	Simple Nodes . . . . .	38

4.5.3	Paste Tree Analysis . . . . .	41
4.6	XSLT Engine . . . . .	41
4.6.1	XML Hierarchy . . . . .	41
4.6.2	Traversing the Paste Tree . . . . .	44
4.6.3	Processing the Leaves . . . . .	45
4.6.4	Processing Internal Nodes . . . . .	45
<b>5</b>	<b>Evaluation</b>	<b>49</b>
5.1	Pilot Test . . . . .	49
5.2	Comparison to Human Written XSLTs . . . . .	54
5.2.1	Weather Report to HTML . . . . .	54
5.2.2	Business Cards to HTML . . . . .	54
5.2.3	Address Book to XML Dialect . . . . .	54
5.2.4	People Grouping to Text . . . . .	56
5.2.5	Recipe to HTML . . . . .	56
5.2.6	Summary of Results . . . . .	58
<b>6</b>	<b>Conclusion</b>	<b>61</b>
6.1	Future Work . . . . .	62
6.1.1	Undo . . . . .	62
6.1.2	Error Reporting . . . . .	62
6.1.3	Editable XML Document . . . . .	63
6.1.4	Snapping Selections . . . . .	64
6.1.5	Inferred Pastes . . . . .	64
6.1.6	Counting . . . . .	65
6.1.7	Lookup Table . . . . .	65
6.1.8	Sorting . . . . .	65
6.2	Summary . . . . .	66

# List of Figures

- 1-1 Two XML dialects for an address book. Although they contain the same information, their differing structure makes them incompatible. . . . . 14
- 1-2 An XSLT defining how to transform one dialect in Figure 1-1 into the other. 15
- 1-3 An example of simultaneous editing used to transform an input XML document into an output. Part (1) shows the initial cursor positions. Between Part (1) and Part (2) the user renames the `<person>` tag, inserts a new line, and surrounds `name` with braces to transform it into an element. After Part (2) the `<name>` element is cleaned up with a closing tag and the `</person>` tag is renamed to `</entry>`. Part (3) shows the transformed result. . . . . 17
- 1-4 XSLT by Demonstration built on LAPIS. . . . . 18
  
- 3-1 The beginning of XSLT generation process what the output pane is empty. Each of the `first-name` values have been selected in the input XML document. 26
- 3-2 The state of the application is shown after the first names have been pasted into the output. All seven regions are pasted to different lines, with a cursor placed after each pasted region. . . . . 27
- 3-3 After the last names are pasted into the output, one following each first name, the output document has reached its desired state (in the background). An XSLT represent the transformation has been generated and is displayed in the foreground. It consists of four templates. The first template matches the root element in the source XML, and it calls `apply-templates` on all containing `person` elements. The second template matches each `person` element, and it calls the third and fourth templates to display the first and last names, respectively. The space inserted between the first and last names is represented by the `xsl:text` element in the template matching `first-name`. . . . . 28

4-1	Coordinate Map Ambiguities . . . . .	35
4-2	Default Resolution to Coordinate Map Ambiguities . . . . .	36
4-3	A user clicks just after the name Bob. However, under the default mapping paradigm, the pattern returned, “point just after name,” maps to the end of the inserted text due to ambiguities in conversion process. . . . .	36
4-4	Mapping a cursor to a different document version time, and then mapping it back, is not guaranteed to be an identity operation. . . . .	37
4-5	Shown on the left is a representation of an output document containing three pastes from the original XML. The range of each paste is shaded differently. On the right is the paste tree corresponding to that output document and contains one leaf per paste. . . . .	39
4-6	Each part above shows a snapshot of the output pane along with a representation of its paste-tree. In Part (a), the <code>first-name</code> and <code>surname</code> data were pasted into the output at the same time. This single paste operation created a single node paste tree. To format the paste further, the XML markup was selected and then deleted, splitting the pasted text into two new ReigonSets, seen in Part (b). The contiguous blocks of text for the first names, and those for the surnames, are each managed by separate nodes – the leaves of the tree. A new parallel node has been created, defining the relationship between the new nodes. . . . .	40
4-7	On the left is a simple XML document containing three types of elements and two types of attributes. Shown on the right is the group graph representation for the XML document. Each of the arrows in the graph indicates a parent/child relationship exists between at least one instance from each group. The arrow from group <i>A</i> to group <i>B</i> thus indicates that there is an element of type <i>B</i> contained directly in an element of type <i>A</i> . In this manner, the group graph summarizes the relationships between all of the individual nodes. . . . .	42



4-8	The XSLT generated for the output in figure 4-6 part (a) with input source shown in figure 5-1. The highlighted region identifies the template that was generated from the single paste node for this output document. Each paste region contained two features: the value of the <code>first-name</code> element and the value of the <code>surname</code> element. Each of these has a corresponding <code>value-of</code> element in the XSLT template. Between the <code>value-of</code> elements, the text that separates each feature is output. . . . .	46
4-9	The XSLT generated for the output in figure 4-6 part (b) with input source shown in figure 5-1. The highlighted region identifies the template that was generated from the parallel node in this output's paste tree. This template is called from the root template once for each <code>person</code> element. It applies a template for each of its children and prints a new line. . . . .	47
5-1	Part of the source XML document used in the pilot test. . . . .	50
5-2	The two different outputs that users were asked to generate in the pilot test. Output (a) was performed before output (b). . . . .	51
5-3	Part (a) shows the desired result. Part (b) shows the unexpected results that users new to simultaneous editing encountered. . . . .	52
5-4	Part (a) shows cursor positions following a paste. Part (b) shows the cursor positions users desired, an arrangement that parallels placements they had already seen. Part (c) shows the unexpected result users encountered instead.	53
5-5	The weather reporting example: (a) input XML; (b) output HTML. . . . .	55
5-6	The business card example: (a) input XML; (b) output HTML. . . . .	55
5-7	The address book example: (a) input XML; (b) output XML. . . . .	56
5-8	The people-groups example: (a) input XML; (b) output text. . . . .	57
5-9	Portions of the input XML file for the recipe example. . . . .	57
5-10	Excerpts of the recipe example's output HTML. . . . .	58
5-11	A data XML document containing parents with many children. . . . .	59
5-12	Output structures from the input in figure 5-11 that are unsupported by the current multi-paste operation. Part (a) shows an output that would require pasting 5 child regions into 2 parent regions. Part (b) shows an output that would require distributing 2 parent regions into 5 child regions. . . . .	60

6-1 Part (a) shows an XML document organizing people into groups. Part (b) shows an XSLT output that would require intelligently pasting four regions into two. Part (c) shows an XSLT output that would require intelligently distributing two regions into four. . . . . 65

# List of Tables

4.1	Methods of XSLTNode relevant to editing. . . . .	39
4.2	Methods of XSLTNode relevant to XSLT generation. . . . .	44



# Chapter 1

## Introduction

The rapid adoption of XML in recent years has been aided by powerful companion tools. One such tool, XSL Transformations (XSLTs), has expanded XML's utility by describing a standard way to define transformations on XML documents. The introduction of XSLT extended the versatility of XML, enabling it to be used by applications requiring dynamic solutions.

As the foundation for defining transformations on XML documents, XSLT has found many applications. One of its primary uses has been to remedy incompatibilities between different XML standards, or dialects. These problems derived from the proliferation of XML, as organizations independently defined their own ways to structure XML documents. Two XML documents illustrating this point are shown in Figure 1-1. Each document describes an address book containing two contacts but using different XML structure. Even though each document contains the same content, these differences in representation hinder interoperability. XSLT provides a solution to this problem by enabling transformations to be written that describe how to convert one representation into another in an implementation independent manner. These solutions can then be deployed and reused with the implementation of an organization's choosing. An XSLT defining how to convert one address book example into the other can be seen in Figure 1-2.

XSLTs are also widely used in the generation of HTML documents from XML. In this application an XSLT defines how to convert a data XML document into its corresponding presentation in HTML, thereby enabling a logical separation of the data from its display. Modern web browsers, including Microsoft Internet Explorer and Mozilla FireFox, have

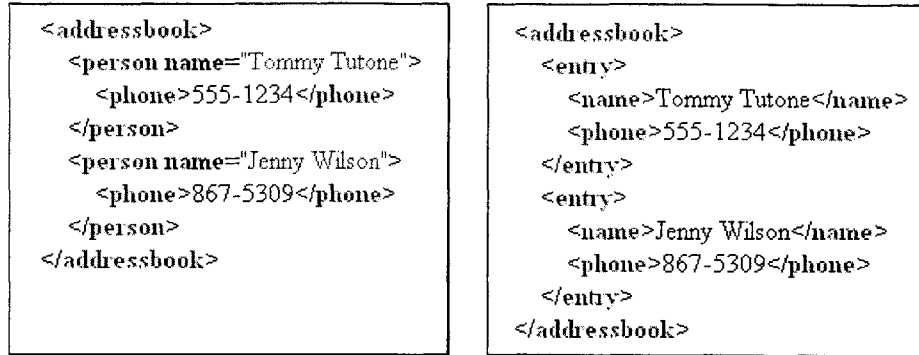


Figure 1-1: Two XML dialects for an address book. Although they contain the same information, their differing structure makes them incompatible.

begun to internally implement XSLT engines, enabling the transformation process to occur on the client's machine. A server can now post an XML document and its associated XSLT file, and then utilize the client's processing power to generate HTML. Many web applications that once required scripting tools and server processing can now use static documents and rely on the client to handle processing.

XSLT is a powerful tool, but some characteristics make it difficult to work with. XSLT defines a transformation language for XSLT documents to be written in. However the transformations are required to be written in well-formed XML, and therefore look very different from traditional programming languages. Also, XSLT variables are read-only and can only be set at declaration time. The reason for this is practical: this property is designed to simplify the implementation of XSLT engines, the systems that actually process a transformation based on an XSLT definition. However, write-once variables can complicate simple tasks. For example, for-loops and while-loops cannot be written in XSLT since variables are immutable. Instead, these loops must be implemented using recursion. As a result of these characteristics, reading and writing transformation definitions can be difficult, even for programmers familiar with other languages. Learning the transformation language is prerequisite, and writing code in well-formed XML is a unique task that takes time to master.

In addition to its usability challenges, the process of defining an XSLT requires a user to divert focus from his primary interests: the input and output. It becomes clear then that XSLTs are intermediaries, and even though their use may be required, the XSLT is secondary to the input and output. Further, a user is likely to be familiar with the input and

```
<xsl:stylesheet version="1.0">
  <xsl:output method="xml"/>
  <xsl:template match="/">
    <addressbook>
      <xsl:apply-templates
        select="addressbook/person"/>
    </addressbook>
  </xsl:template>
  <xsl:template match="person">
    <entry>
      <name>
        <xsl:value-of select="@name"/>
      </name>
      <phone>
        <xsl:value-of select="phone"/>
      </phone>
    </entry>
  </xsl:template>
</xsl:stylesheet>
```

Figure 1-2: An XSLT defining how to transform one dialect in Figure 1-1 into the other.

output domains, but the process of writing an XSLT may be completely foreign. Therefore, a new method for defining XSLTs is desired that would allow the user to work with the input and output instead of defining the transformation directly.

### 1.1 XSLT by Demonstration

The process of defining an XSLT can be simplified by generating the transformation definition based on a demonstration of the desired effects. A tool accomplishing this would overcome all of the drawbacks highlighted earlier about XSLT creation process. Most notably, the user's work would be performed in the input and output domains, with the XSLT details hidden from view.

Previous work on using Programming By Demonstration (PBD) to generate XSLT has focused on inferring the transformation rules based on input and output examples. But determining the rules proved to be a very difficult problem. Just determining where data from the input mapped to the output proved a significant hurdle on its own. Another

problem with the method was that it assumed input/output examples already existed, when without the transformation definition outputs might not exist or could be hard to create. Further, errors and non-uniformities in human-generated outputs could make valid transformation rules impossible to identify.

This thesis explores an alternate method for demonstrating the transformation. Instead of examining a large set of input/output pairs, this method focuses on the *process* of converting a single XML input to the desired transformation output. Patterns in the editing sequence provide additional information that can be used to help generate an accurate XSLT. Also, this method does not assume a set of input/output pairs already exists – its goal is to assist in output creation from a single input XML document. To explore the feasibility of this approach a proof of concept application, XSLT by Demonstration, has been implemented.

## 1.2 Simultaneous Editing

While the editing process used to transform an XML document can provide important information to the XSLT generating tool, the size of an XML document and its repeated structure pose challenges to efficient editing. To ease this burden XSLT by Demonstration is built upon recent advances in *simultaneous editing* that enable the processing of a document's repetitive structure in parallel. [1] Simultaneous editing allows the user to edit a document with multiple cursors at the same time. The multiple cursors are placed by giving examples, which are generalized into a pattern based on the document's structure.

An example detailing how simultaneous editing can be used to transform an XML document is shown in Figure 1-3. In it the use of multiple cursors allowed both elements to be edited at once, adding efficiency to the editing process. Moreover, the example contained only two `person` elements, but in a real address book there are likely to be many more names, posing a real challenge to traditional editing.

In addition to streamlining the editing process, simultaneous editing provides important clues about the form of the desired transformation. In Figure 1-3 both `person` elements were edited in parallel, suggesting a pattern to appear in the transformation. If the changes had been made using traditional editing the parallelism would not have been apparent; determining the editing pattern would have required detailed text analysis. Even if the



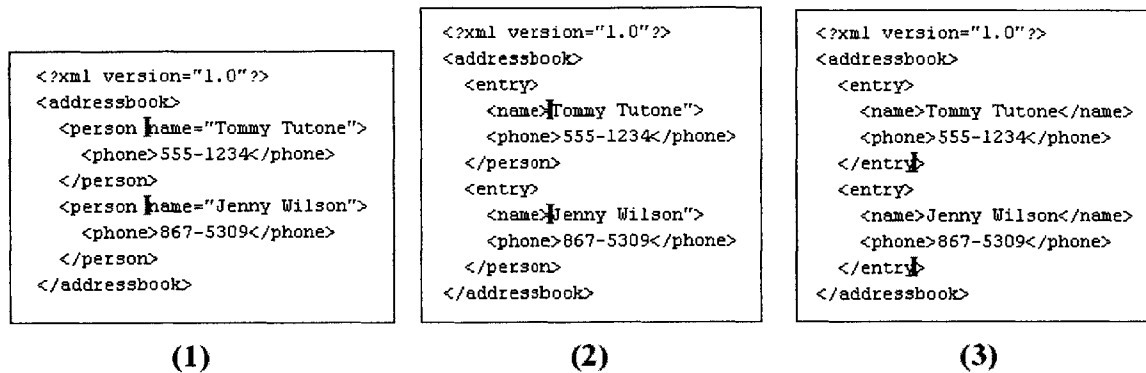


Figure 1-3: An example of simultaneous editing used to transform an input XML document into an output. Part (1) shows the initial cursor positions. Between Part (1) and Part (2) the user renames the `<person>` tag, inserts a new line, and surrounds name with braces to transform it into an element. After Part (2) the `<name>` element is cleaned up with a closing tag and the `</person>` tag is renamed to `</entry>`. Part (3) shows the transformed result.

user attempted to edit each `person` element in the same manner, typos and mistakes could result in a heterogenous output, complicating pattern identification or even preventing representation by any useful XSLT. Simultaneous editing helps prevent these outcomes and at the same time makes it easier to identify the correct transformation.

### 1.3 LAPIS

Much of Miller's simultaneous editing research has been implemented in LAPIS, [2] which features support for text editing with multiple cursors and provides tools to aid in the simultaneous editing process. XSLT by Demonstration has been built on top of LAPIS, applying these features to XSLT generation. A screen shot of the tool is shown in Figure 1-4. The top text pane contains the original XML document and is uneditable. It provides easy access to the source data, which may be pasted into the output. The bottom pane contains the evolving transformation output. It begins as an empty document and evolves through pastes from the XML document and direct simultaneous text editing until it has reached the desired output. When the editing process has completed the user can generate an XSLT definition representing the transformation.

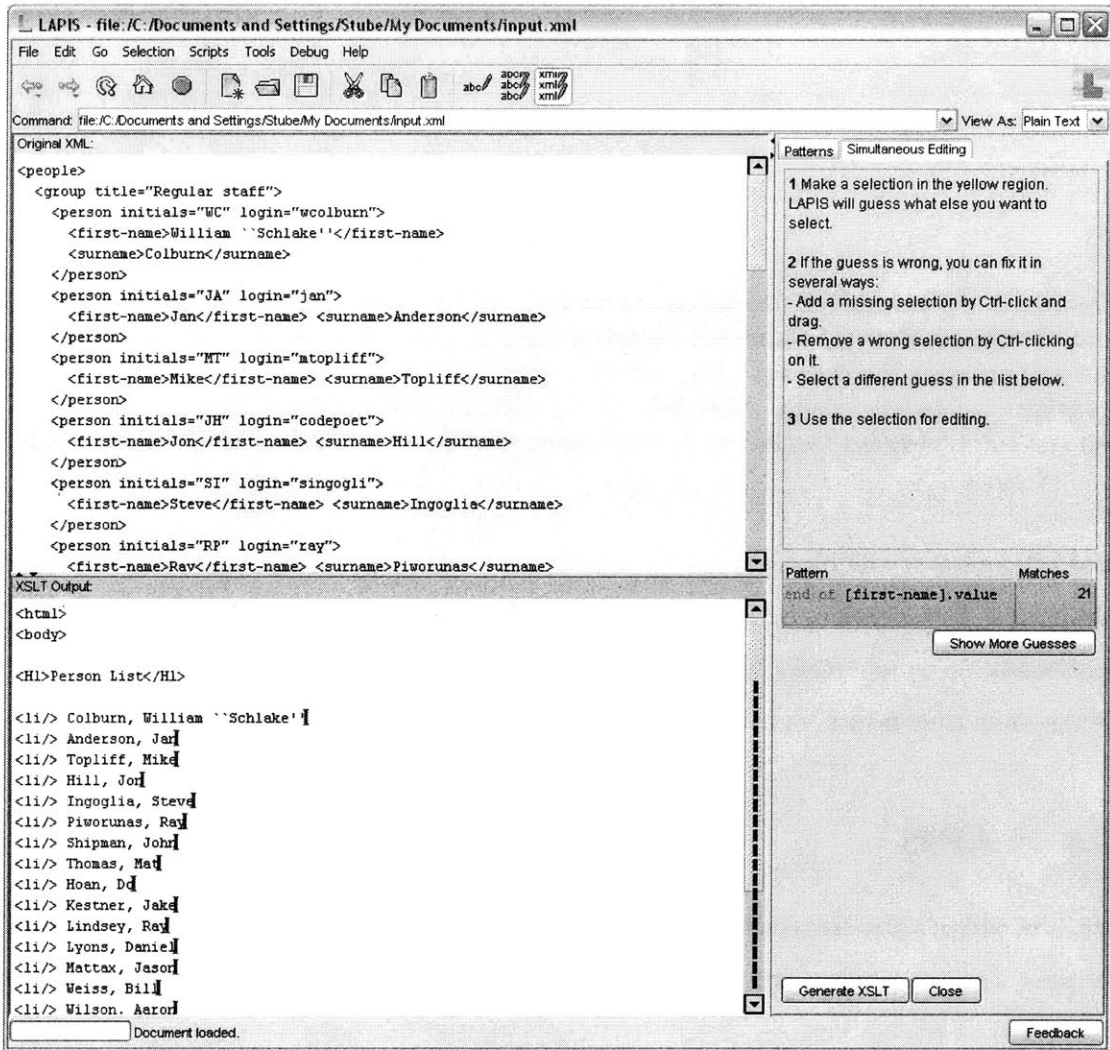


Figure 1-4: XSLT by Demonstration built on LAPIS.

## 1.4 Outline

The rest of this thesis is comprised of five chapters:

**Chapter 2 - Related Work:** Details the current state of XSLT tools and previous work on Programming By Demonstration.

**Chapter 3 - User Interface:** Describes the use of XSLT by Demonstration and the principles behind its design.

**Chapter 4 - Implementation:** Details how user interface tools and XSLT generation were implemented.

**Chapter 5 - Evaluation:** Describes the results of a pilot user study and evaluates the application for generating XSLT.

**Chapter 6 - Conclusion:** Summarizes the results and identifies how the interface and implementation might be improved.



## Chapter 2

# Related Work

### 2.1 Current XSLT Tools

Many tools have been developed to ease the XSLT developing process, with a general focus on helping the user build the XSLT piece by piece. This is often accomplished by examining the structure of the input and desired output, as encoded in their schemas or DTDs, and then constructing a tree representation of the document based on this information. The job of the user is then to link up related input and output components. Some products implementing this type of interface include XSLWiz 2.0 [3] and CapeStudio 3.0. [4] IBM's alphaworks showcases XSLerator, [5] another tool designed under the same paradigm. However, in each of these cases the user is limited to XML at the output, and XSLWiz and CapeStudio require a schema or DTD for both the input and output.

A different approach, implemented in the IBM technology preview XSLbyDemo, [6] streamlines XSLT generation for the specific case of adding style to HTML documents. User operations on a WYSIWYG view of the HTML document are converted to XSLT rules, with minimal knowledge of XSLT required. Its use of demonstrations to generate XSLT make it similar to this thesis, however its primary drawback is its focus on a very specific XSLT application. Further, the implementation of the product was never fully completed. The use of simultaneous editing explored in this thesis may be the usability breakthrough required for such a tool to succeed.

## 2.2 Programming By Demonstration

Programming By Demonstration (PBD) is a means of defining programs while remaining separated from the programming language itself. Instead of writing the code directly, the desired actions of the program are demonstrated, from which the code is then inferred. The goal of PBD is often to simplify the programming process so that knowledge of the detailed programming level is not required. Instead, only knowledge about the program's domain and its desired effects are necessary. Macros in word processors are a well-known example of PBD that also illustrate the technique's goal of making programming accessible to a larger population.

Another advantage of programming by demonstration is that it enables the user to work in the environment where the effects of the program will actually take place. This environment is often more familiar to a user than the programming language itself. Because of this, PBD interfaces can provide a user interface that more closely resembles the user's natural understanding of a problem. [7] This often helps to reduce errors in the programming process.

Previous research in this area includes SMARTedit, [9] a text editor augmented to enable automation of repetitive commands through intelligent macros. After recording the user's input sequence the editor generates predictions of similar actions the user might want to perform next, sorting the predictions by their inferred likelihoods and presenting them one at a time. When the prediction for the next action is shown the user can either accept the action or request the next prediction. This input from the user is then used by the prediction engine to eliminate incompatible hypotheses that have not yet been recommended, increasing the accuracy of future predictions.

PBD research performed by Nix was based in a similar approach, studying examples from the user as a means to automate repetitive tasks during text editing. [8] However the process used more examples from the user to come up with one rule which could then be automated. Instead of making outright suggestions for the next edit, to be accepted or denied as a rule evolved, Nix's implementation displayed a single inferred rule based on the examples it had seen thus far. The user could accept the rule and use it to transform the document, or instead, further examples could be provided until a more satisfactory rule was given. One of the unique characteristics of Nix's approach was that to determine the

inferred rule only the input and output of the examples were considered, while the particular editing sequence used to generate them was ignored. This was data that future work in the field would incorporate into their algorithms.

The XSLT generator being explored by this thesis is instance of PBD categorized as example-based programming. [10] These types of applications seek to enable programming at a natural level of understanding, rather than in detailed and foreign programming language constructs.





## Chapter 3

# User Interface

The primary focus of XSLT by Demonstration's interface is to ease the process of converting an input XML document into the desired transform output. From this conversion process the XSLT definition will be generated.

### 3.1 Generating an XSLT by Demonstration

The generation of a simple XSLT is demonstrated in this section. Beginning with an XML document containing `person` elements distributed into `group` elements, the example transformation's goal is to generate a list of names. The list will contain one person per line, consisting of first name followed by last name.

After choosing the input file, the application begins with two text panes. The top pane contains the source XML and is not editable. The bottom pane begins blank and will be transformed through text editing to become the desired output.

Figure 3-1 shows the application just after selection of the text "Buddy," the contents of the *first first-name* element. Based on the document's structure, LAPIS generalized the selection to the value of *every first-name* element. This hypothesis is described in the pane to the right, indicating that there are 7 *first-name* elements in the XML document. If the first hypothesis was not correct, by clicking the "Show More Guesses" button alternative hypotheses could be viewed. Or instead, more examples could be given by holding down Control and selecting more regions. But since we want to select the value of every first name, we copy the current selection and move to the output pane.

After clicking in the output pane the first names can be pasted. The results from this

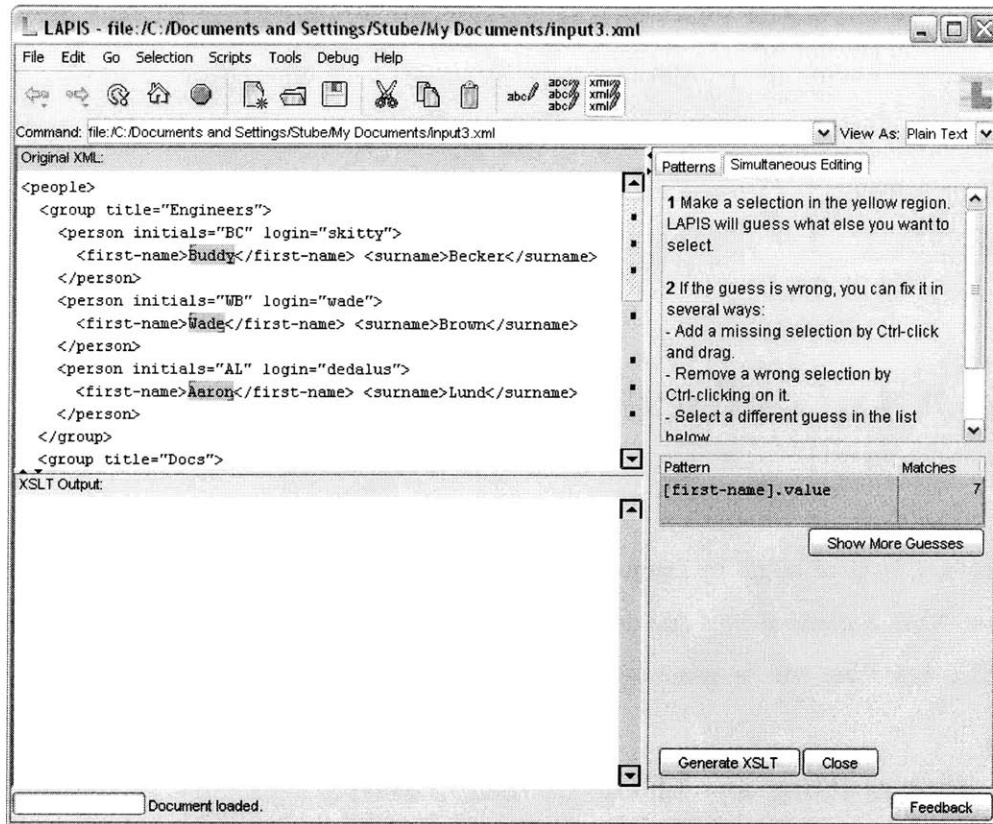


Figure 3-1: The beginning of XSLT generation process what the output pane is empty. Each of the first-name values have been selected in the input XML document.

action are shown in figure 3-2. Each of the names was placed on its own line, followed by a cursor. Now instead of just one cursor there are seven. After each of the first names we intend to paste a last name, but a space is inserted first. One press of the space bar inserts a space at each of the seven cursors.

Next we need to paste the last names following the first names. To do this we go back to the XML source pane and select one of the last names. LAPIS again generalizes the selection to every last names, and after copying them we move back to the output pane.

To position the cursors for the paste, we click after the space following one of the first names. Here the single cursor selection is again generalized to a total of seven cursors, one after each of the inserted spaces. Now that there are seven cursors, and we have seven last names in the clipboard, we paste the regions into the output. With the output document in its final desired state, we can click on the “Generate XSLT” button to create an XSLT that defines our demonstrated transformation. The final state of the output document and

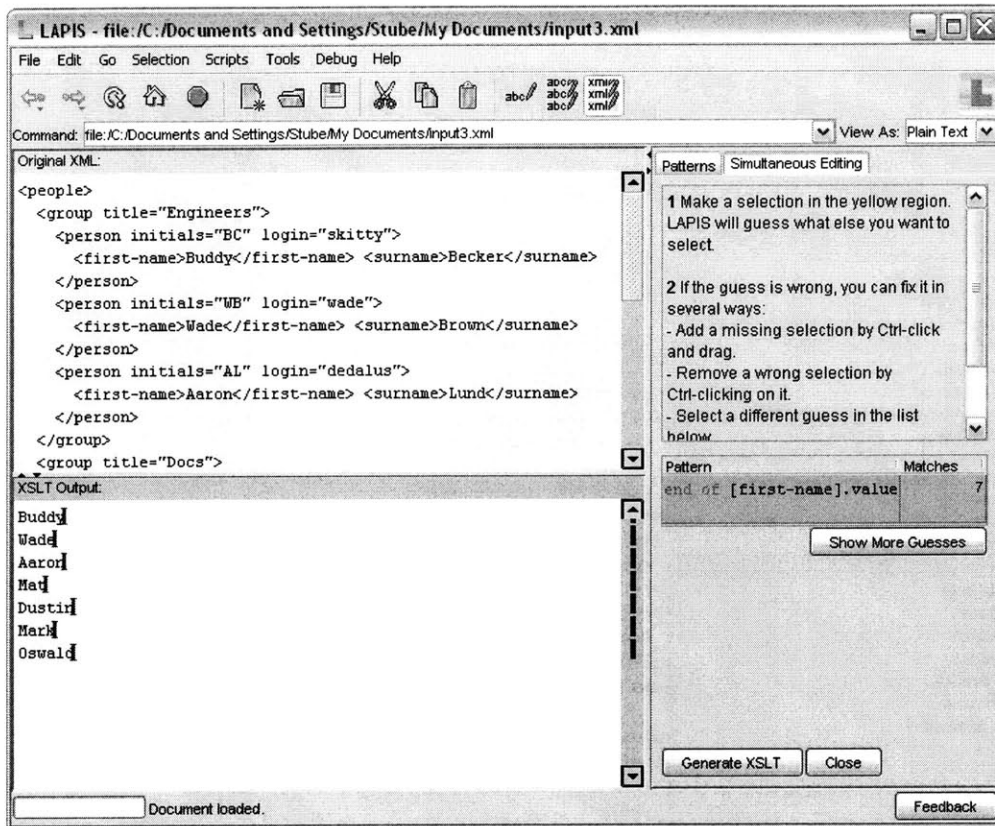


Figure 3-2: The state of the application is shown after the first names have been pasted into the output. All seven regions are pasted to different lines, with a cursor placed after each pasted region.

the generated XSLT can both be seen in figure 3-3.

### 3.2 The Two-Pane Solution

The two-pane setup was chosen as the basis for the user interface because it provided the most flexibility. The outputs of XSLTs vary widely. For that reason it was important to begin with a blank output document, allowing the user to choose what data makes it from the XML source into the output. If only a small selection from the original XML was needed in the output, it would be possible to copy only those parts. If the XML document were to remain mostly unchanged in the output aside from minor alterations, it would also be possible to copy the whole XML to the output and then perform the changes. Beginning with the blank output provided the flexibility to adapt to these different needs.

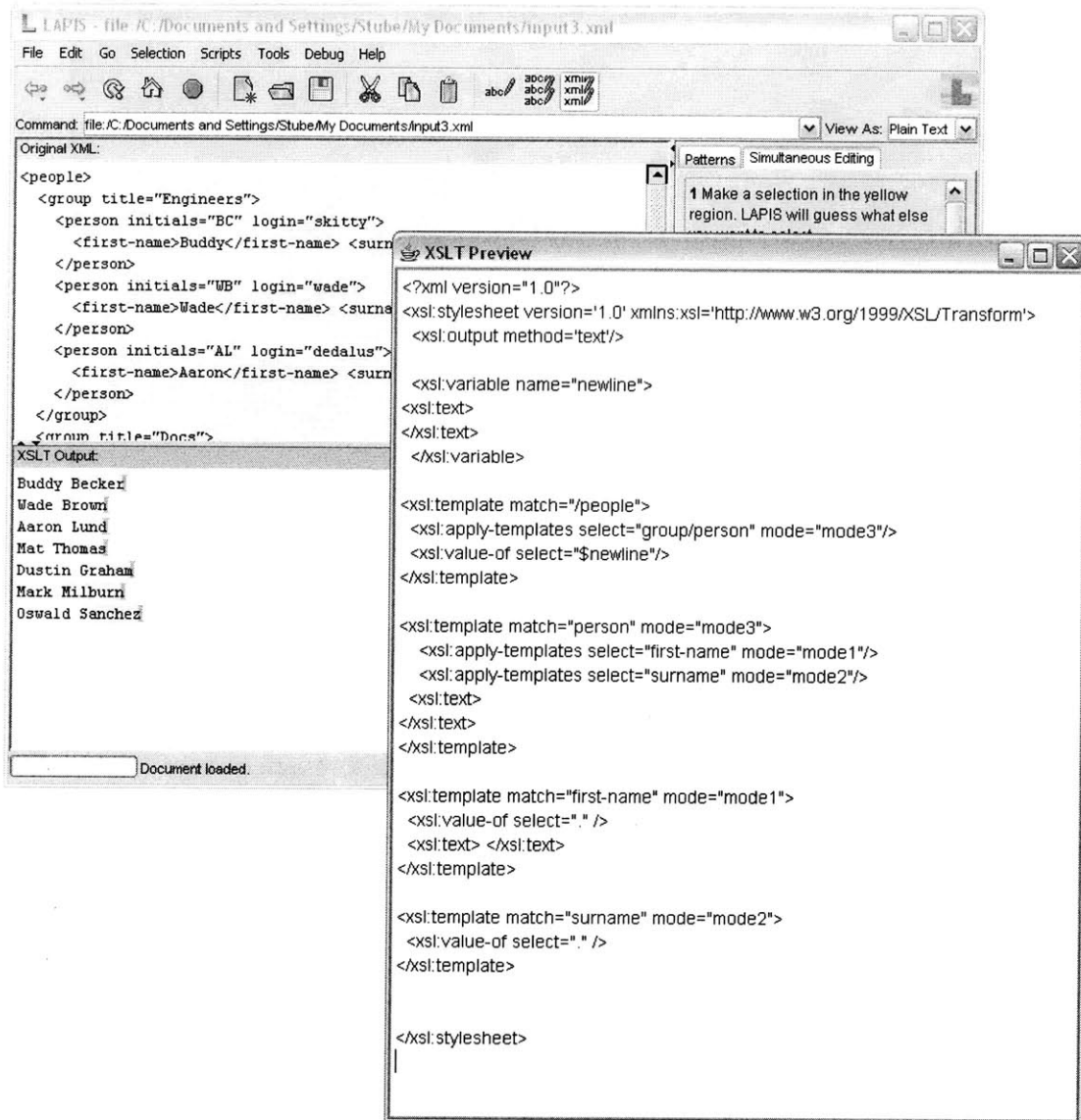


Figure 3-3: After the last names are pasted into the output, one following each first name, the output document has reached its desired state (in the background). An XSLT represent the transformation has been generated and is displayed in the foreground. It consists of four templates. The first template matches the root element in the source XML, and it calls apply-templates on all containing person elements. The second template matches each person element, and it calls the third and fourth templates to display the first and last names, respectively. The space inserted between the first and last names is represented by the xsl:text element in the template matching first-name.

Placing the input in a second pane was a decision that followed logically from the output pane design. If the output began blank then the input document would need to be accessible somewhere. The simplicity of the interface, presenting the input in text form without any special rendering, enabled simultaneous editing functionality to be utilized on both the input for copying, and the output for pasting and editing, and helped to provide a consistent interface. Tools implemented in LAPIS to enhance simultaneous editing (see Section 3.4) were adapted for use in both panes.

The immutable state of the input XML document is not ideal for usability. The input XML pane was initially intended to be a scratch pad, that contained input XML but allowed it to be edited. This would enable the user to manipulate the selections before copying them to the output. Once the copying had occurred, the user could revert the scratch pad contents to the original XML to begin again. However, limitations in the generalizing algorithms, as discussed in Section 6.1.3, caused them to fail when the source XML was mutable.

### 3.3 Working with Multiple Cursors

Most editing operations with multiple cursors work the same way as they did with one cursor, except that the effects occur at each cursor in parallel. This is the case for inserting text, pressing delete, and pressing backspace. If a single region has been copied, then a paste into multiple cursors works in parallel too, with the text being replicated at each cursor.

Copying and pasting with multiple regions must be handled a little more carefully. In traditional single-selection text editing a one-to-one relation between the number of regions copied and the number of cursors is always trivially satisfied. Hence, it is always clear what the result of a paste operation will be. For pastes of multiple regions though, the one-to-one mapping is still generally required, but not always satisfied. When an multiple selection of  $n$  regions is copied the set of selections is remembered. Those  $n$  regions can then only be pasted into one cursor or  $n$  cursors. If the paste is into one cursor, then the  $n$  regions are appended together and inserted at the cursor. If the paste is into  $n$  cursors, then one selection is pasted at each of the cursors. Pastes of  $n$  regions into  $m$  cursors, where  $m \neq 1$  and  $m \neq n$ , are not performed and instead the user is alerted of the error.

XML data is not always uniform though, and desired transformations sometimes clashes

with these strict paste requirements. Ideas on how to relax the above conditions for XSLT generation are discussed in Section 6.1.5.

### 3.4 Generalizing Selections

Simultaneous editing with multiple cursors can help the user perform repetitive tasks quickly. But actually placing the cursors prior to editing can still be a problem. To edit 20 different items in parallel would require the tedious placement of 20 different cursors at precise locations. Any placement error of just a single character could have a significant impact. Another challenge for simultaneous editing is dealing with documents that have repeated structure but varied text lengths. To simultaneously delete the text of every `first-name` element the user could start by placing the cursor at the end of every name and then press backspace a number of times. But the length of each name is non-uniform and so this simple deleting process would not be effective.

LAPIS provides the functionality to handle both of these problems in the selection inference engine. When provided a set of selections, the inference engine suggests ways to generalize the selections based on the document's structure. Given a cursor placement at the start of `<person>`, the inference engine might provide a hypothesis list as follows:

1. before every `person` start tag
2. before every `person` tag
3. before every XML tag

By holding down the Control key and generating more cursor selections the user can provide additional examples for the hypotheses to be based on. Also, by holding Control and unselecting suggested selections, the user can identify selections that should be excluded from any hypothesis. Through these positive and negative examples the user can help the system hone in on a desired pattern.

The selection inferences in LAPIS also apply to selections of text. Given that a user has selected a whole block of text, such as the content of a `first-name` element, LAPIS can generalize the selection, providing many possible hypotheses. Some hypotheses that might

be returned for that selection include:

1. the value of every `person` element
2. every XML element value
3. every block of text just after “>” to just before “<”

Once again, by feedback of positive and negative examples the user can hone in on the desired pattern. Once the desired pattern is reached the whole block may be edited in parallel, by deleting every selection, or copying the many selections at once.

### **3.5 Generating the XSLT**

Once the user has reached the desired final state for the output an XSLT performing the desired transformation can be generated. When the “Generate XSLT” button is clicked, the XSLT definition is displayed to the user, along with the results of it being applied to the input XML.

Throughout the editing process the user may require the ability to preview the XSLT. However the output of the preview will depend on the document’s current state, which may or may not be well formed or even sufficiently resemble the final desired output. The ability to view the XSLT at intermediate states is an important sanity check. So even though the generated XSLT may not be valuable at all times, it can still be generated at any time.





## Chapter 4

# Implementation

The implementation of XSLT by Demonstration is divided into two main parts. The first component is the user interface, with its primary design goal being to enable efficient creation of the transformation output through text editing. The specialization of LAPIS's simultaneous editing tools was key to achieving this goal. The second component is the XSLT generating engine that determines the XSLT to produce based on the user's editing history.

### 4.1 Region Sets

XSLT by Demonstration, and LAPIS itself, require a means to represent and manipulate multiple selections. The `RegionSet` object abstracts this functionality.

The basic building blocks of `RegionSets` are regions. A region represents a contiguous block of text and is defined by its starting and ending positions. Regions can represent the entire content of a document, or they can have zero length, representing a cursor position in a document. When a user makes multiple selections in a document, each of these selections is a region, and the regions are all grouped together as one `RegionSet`.

LAPIS provides a variety of ways to manipulate and work with `RegionSets`. XSLT by Demonstration only utilizes a small portion of the provided functionality, most often just intersecting `RegionSets` or testing for containment.

## 4.2 Specializing the Inference Engine for XML

Effective generalization of user selections is critical for the usability of simultaneous editing. The standard inference engine implementation was designed to be general enough to work with any kind of structured document. It readily identified patterns such as dates, sentences, and phone numbers, but when applied to XML documents many of the default patterns were not useful.

To optimize the inferences for use in generating XSLTs the inference engine was specialized to work with XML documents. Many unrelated patterns, such as those that identify sentences and paragraphs, were deactivated. Also, patterns referring to absolute ordering, such as “the first attribute in an element” were also eliminated since XSLT’s often focus on the names of elements and attributes, rather than their sequence. Finally, the hypotheses were ordered by their simplicity and specificity.

The process was also motivated by concern that users might copy selections that could not be represented in an XSLT. However the inference engine was not constrained so far as to only allowed patterns representable by XPath. This is a feature that might be useful to implement in the future.

## 4.3 Inference Backed Windows

Generalizing selections in the output pane required a different solution. The content that most frequently requires parallel processing in the output pane is text that has been pasted from the input XML. Because of this, selection generalization focuses on the pasted regions. When the user clicks in content that has been pasted from the source XML, inferences should still treat the paste regions as part of an XML document, since they still contain many XML features. To implement this, clicks in the paste region are mapped back to the original XML document, generalized there, and then mapped back to the paste region. The object that performs this task is the Inference Backed Window.

An Inference Backed Window is created for every paste from the source XML into the output document. It remembers where the regions were copied from in the XML document, and where the regions were pasted to in the output document. It also computes a mapping between those regions so that selections can be transferred between them. When a user clicks in the output document, the selections are then mapped to the XML document.

There the selections are generalized in the source document, and only the hypotheses that are wholly contained in the copied regions are passed back to the output.

## 4.4 Document Versions and Coordinate Maps

The mapping between source and output documents created in an Inference Backed Window can become invalid as the user continues to make edits. The locations of the pasted text may change as the user inserts or deletes text both before and between the regions. Hence, this tool requires a means to represent past versions of a document and how selections in one version are related to those in another. Then selections in the current document version could be mapped back to the document at paste time, where the Inference Backed Window works.

To solve similar problems in LAPIS the `CoordinateMap` interface was developed. It defines how to map a region in one document version to another document version and provides the means to convert `RegionSets` based on these mappings. But there are ambiguities in the `CoordinateMap` process since no one-to-one mapping exists between index positions in different versions. To see this consider the simple example shown in Figure 4-1 where the original text “sum” has had the text “odi” inserted to transform it into “sodium.” A cursor between the ‘s’ and ‘u’ in “sum” can be mapped to many possible positions in “sodium.” It could follow the ‘s,’ or the ‘u,’ or be mapped to any location in between.

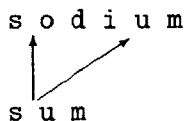


Figure 4-1: Coordinate Map Ambiguities

LAPIS remedies this ambiguity by following a convention to never expand endpoints to include inserted text, but text inserted inside of a region will be included in the mapped region. Examples of this conversion rule can be see in Figure 4-2. When converting from “sum” to “sodium,” the regions “s” and “um” do not expand to include the new text. But the selection of the whole word “sum” does map to include all of “sodium.”

The convention breaks down for regions of zero length though, such as the one shown

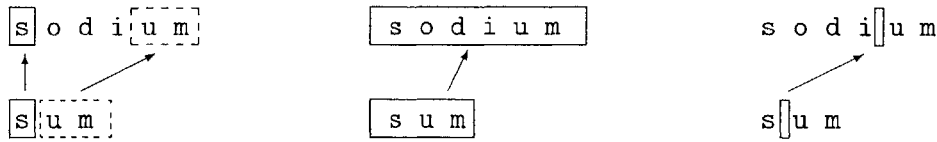


Figure 4-2: Default Resolution to Coordinate Map Ambiguities

in 4-2. Even without expanding to contain nearby text, the cursor has all of the same ambiguities as before. The default resolution to this ambiguity is to always follow the character to the right, in this case the 'u.'

```

John is a name.
Bob is a name.
Clarence is a name.
  
```

Figure 4-3: A user clicks just after the name Bob. However, under the default mapping paradigm, the pattern returned, "point just after name," maps to the end of the inserted text due to ambiguities in conversion process.

Even with the default conversion functionality provided by LAPIS, converting between regions still caused problems. Mapped regions do not contain information about where they came from, only about how they are represented in the current version. Therefore, after converting a region to a different version, mapping it back might result in a region different than when it started. An example of this is shown in figure 4-3 where three first names have been pasted and then text has been inserted after them. After clicking just after the name Bob, the selection is generalized, and using the default conversion functionality, the regions returned are mapped to the end of the line. Clearly this is not the desired hypothesis - it doesn't even contain the original cursor. Figure 4-4 shows the path the cursor followed through the different versions.

To deal with these mapping problems, care had to be taken to identify when the cursor would be mapped outside of the paste region and to fix it when it did. To simplify the solution it was always assumed, and ensured in the implementation, that the pasted regions remained contiguous. Hence, mapping of any points on the inside of the paste region would not be adjacent to inserted text, and could be converted without ambiguity. Only the endpoints, then, had to be worried about. Their conversion was done by totally avoiding

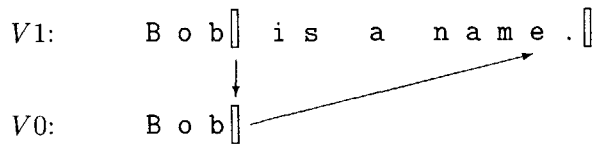


Figure 4-4: Mapping a cursor to a different document version time, and then mapping it back, is not guaranteed to be an identity operation.

the default conversion process, and instead were transferred manually.

## 4.5 Paste Tree

To manage the pasted data a paste-tree structure was designed. The leaves of the tree correspond to single pastes from the original XML document and are responsible for keeping track of the paste source. Meanwhile, the internal nodes represent how the pastes relate to each other, and are responsible for delegating events, such as selections and edits, to the appropriate leaf node.

Each paste-node is derived from the abstract base class `XSLTNode` and implements the methods shown in Table 4.1. The range of a node, as accessed through `getRange` method, indicates the output document regions that the node is responsible for managing. The range for a leaf will at least contain the pasted text it manages and the range of an internal node will at least contain the ranges of its children.

Most the `XSLTNode` event handling methods can be delegated to child nodes and work effectively with the tree structure. A fundamental assumption under this design is that selections will only be contained within one set of paste regions, thereby enabling a single node to fully handle the effects of a user's action. This restriction limits user freedom, but it is assumed that in most situations the user would not need to edit multiple paste regions at the same time. And the simplification it enables in the implementation is worth the trade off.

A representation of text in an output document and the related paste-tree is shown in Figure 4-5. Each paste has a corresponding leaf in the paste tree, with each pair shown in a different color. In the given picture two of the primary paste relationships can be identified: pastes related in series, and pastes related in parallel.

### 4.5.1 Internal Nodes

Given the above restriction on selections, every edit operation can then be handled by a single leaf node. The internal nodes of the paste tree then serve two purposes:

1. They delegate event processing to the appropriate leaf
2. They represent relationships between pasted regions

The children of a series node occupy disjoint regions of the document, ie:

$$child[i].range.end < child[i + 1].range.start$$

The range of a series node is a single region that contains all of its children.

The children of a parallel node have a more complicated relationship. All of the children must have the same number of regions in their range. The arrangement of the regions is like that of a table: each child corresponds to a column, each region corresponds to a cell, and the number of rows equals the number of regions per child. The range of a parallel node has a region for each row of the table.

### 4.5.2 Simple Nodes

When a paste from the XML document into the output pane occurs a simple node is created to manage it. As mentioned in section 4.3, an Inference Backed Window is created for every paste too. The simple node actually contains the newly created Inference Backed Window, giving it a home in the paste tree, and using it to help implement many of the XsltNode methods. Simple nodes manage most of the editing details in the application. These nodes handle almost every call to the paste-tree, and understanding their functionality is key to understanding the implementation as a whole.

The simplest event for a simple node to handle is a uniform edit at either the start or end the node's range. To deal with this event the node extends the range to include the new text, and also adds a feature to the Inference Backed Window so that cursor positions both before and after the edit will be included in any future call to generalize. Through these editing operations the range, which began as the original paste regions, can grow and change over time.

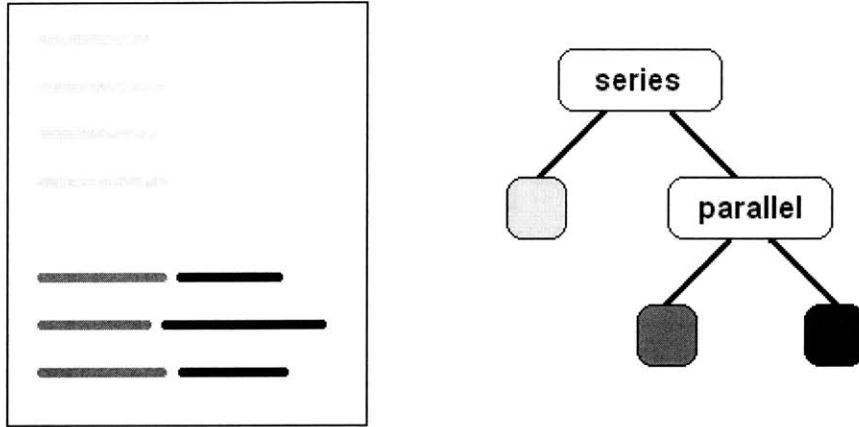


Figure 4-5: Shown on the left is a representation of an output document containing three pastes from the original XML. The range of each paste is shaded differently. On the right is the paste tree corresponding to that output document and contains one leaf per paste.

Table 4.1: Methods of XSLTNode relevant to editing.

Method Name	Method Description
<code>getRange</code>	Returns the regions in the output document that this node manages
<code>generalize</code>	Returns hypotheses for selections in this node's range
<code>processDelete</code>	Processes a delete that has occurred in this node's range
<code>processEdit</code>	Processes an edit in this node's range
<code>processMultiPaste</code>	Handles a paste that has occurred in this node's range

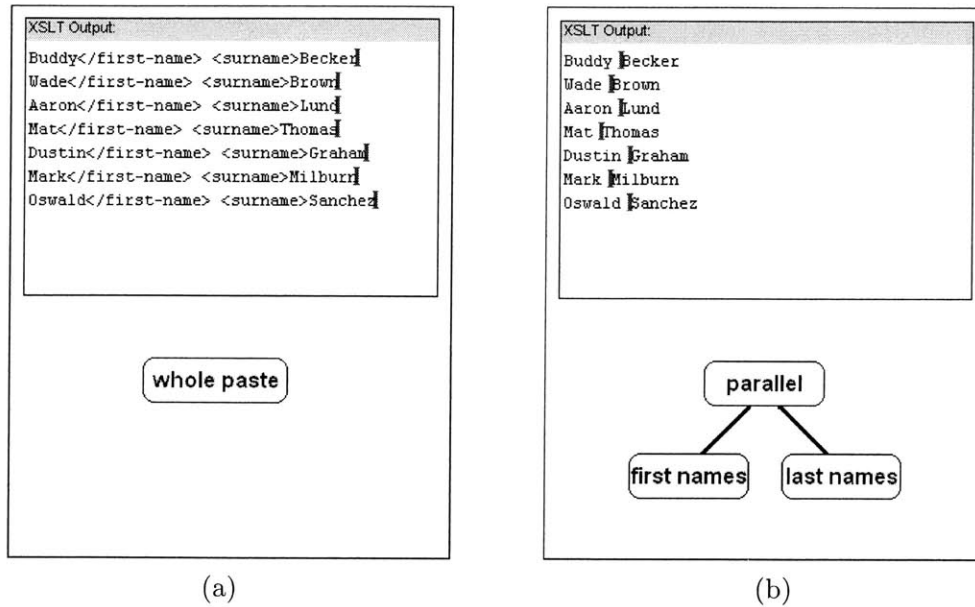


Figure 4-6: Each part above shows a snapshot of the output pane along with a representation of its paste-tree. In Part (a), the `first-name` and `surname` data were pasted into the output at the same time. This single paste operation created a single node paste tree. To format the paste further, the XML markup was selected and then deleted, splitting the pasted text into two new ReigonSets, seen in Part (b). The contiguous blocks of text for the first names, and those for the surnames, are each managed by separate nodes – the leaves of the tree. A new parallel node has been created, defining the relationship between the new nodes.

Some more complicated events can require actions outside of a simple node's intended scope. Simple nodes are designed to have contiguous paste regions, but the regions can be split by an edit operation. To handle these operations the node splits itself into two new simple nodes, and creates a new parallel node to become their parent. Then, the original simple node replaces itself with the parallel node. Since the original node couldn't represent the new structure, it created something that could. An example of a simple node being split is shown in figure 4-6. In the example a delete operation is made in the middle of the paste regions, splitting them into two parallel region sets.

A multi-paste anywhere in a simple node's range also requires similar processing. A simple-node is designed to be a leaf of the tree and thus, cannot add the new node as a child. The paste event that created the parallel node in Figure 4-5 is an example of this event type.



### 4.5.3 Paste Tree Analysis

Some analysis of the various node types yields a few rules. Firstly, it is observed that series nodes can never be children of parallel nodes. However series nodes can contain both parallel nodes and series nodes. It can also be shown that any parallel node possessing a parallel child can be simplified into a single parallel node, and an analogous rule applies for series nodes. Together these rules imply that a paste tree created from these node types will never require a depth greater than three, even though it may be arbitrarily broad. XSLT by Demonstration implements the node merging functionality that enables this property to hold.

## 4.6 XSLT Engine

From a high level of abstraction, XSLT by Demonstration generates XSLT definitions based on three inputs: the original XML document, a sequence of edits and the output document. All of this information is stored in the paste tree, and so to generate the XSLT the paste-tree is traversed and the information in its structure and leaves determines the transformation definition.

### 4.6.1 XML Hierarchy

In the XSLT generation process, XML relationships between various paste sources must be identified and translated to XPath for use within templates. The XML hierarchy component of the XSLT generator assists in this need by providing two main functions:

1. It can determine the most specific element type that contains a set of regions.
2. It can identify the XML features, such as attributes, element contents and other values XSLT can reference, that are contained within a set of regions.

Instead of relying on an XML Schema or a DTD, the XML Hierarchy is constructed directly from the source XML, through a hook into the SAX parser. As the parser identifies start tags, end tags and attributes, it returns information about them, including the starting and ending positions. From this information a tree similar to an XML DOM is constructed. The primary difference is that nodes are supplemented with information detailing their location in the XML document.

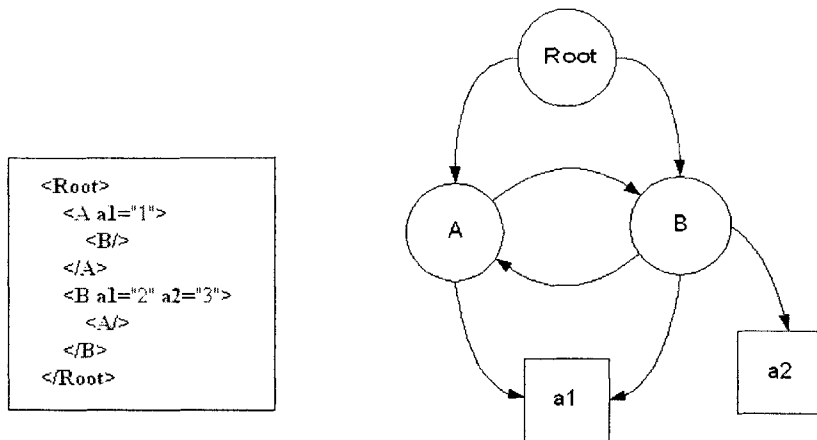


Figure 4-7: On the left is a simple XML document containing three types of elements and two types of attributes. Shown on the right is the group graph representation for the XML document. Each of the arrows in the graph indicates a parent/child relationship exists between at least one instance from each group. The arrow from group *A* to group *B* thus indicates that there is an element of type *B* contained directly in an element of type *A*. In this manner, the group graph summarizes the relationships between all of the individual nodes.

A structure called the group graph is created from the DOM tree to summarize parent-child relationships between node types. One group exists in the group graph for each node type in the DOM tree – one for each element type and each attribute type in the XML document. A parent-child relationship between groups *A* and *B* exists if a node in group *A* is parent of a node in *B*. Additionally, each group contains a master RegionSet, which is the combination of regions from all nodes in that group. Hence, the group graph summarizes an XML document's parent-child structure and also summarizes regions for each node type.

The group graph is what would result if the DOM tree was taken, and one by one nodes of the same type were merged. Therefore, even though the DOM structure from which it is derived is a tree, the group graph may have cycles or may contain groups with multiple parents. An example of a group graph for an XML document is shown in Figure 4-7. Even for such a simple XML file the group graph is surprisingly complex.

The primary purpose of the group graph is to determine XML representations from RegionSet references. One important method implemented by the group graph, is to map a RegionSet to the most specific containing XML element type. This is necessary to determine where a paste came from and how to represent it in an XSLT.

The implementation for finding the most specific containing element must deal well with the flexibility of XML's structure. Because of this, the group graph itself has few structural restrictions. The single group graph property utilized by the algorithm is that if a group has only one parent, then its RegionSet will be a subset of its parent's RegionSet. This means that if the parent doesn't contain the query regions, then neither will the child, allowing a search to be pruned. This property is very powerful for trees, where every non-root node has exactly one parent, but in the group graph there is no requirement that this holds.

To allow for possible use of this performance enhancement when answering queries, the group graph is explored, beginning from the root, in manner similar to depth-first search on trees. The intent of the search order is to process parents before their children. This ordering, however, can be complicated by the presence of cycles in the graph, a condition that renders parents-first ordering meaningless.

The implementation of this ordering maintains a set of unvisited groups. If possible, at each step an unvisited group with no unvisited parents is returned. If this is not possible, and still unvisited nodes remain, then a cycle must be present. In this case, arbitrary groups are selected from the cycle until it is broken, at which point the parents-first processing begins again. The implementation achieves a running time of  $O(n^2)$ , where  $n$  is the number of groups in the graph. Although performance enhancements are possible, since most XML documents have fewer than 50 groups, this running time is not a significant burden. And in the case where the graph reduces to a tree, the algorithm can take advantage of the structure and complete in  $O(h)$  where  $h$  is the height of the group graph.

A different complication with group graph analysis is that attributes with the same name may be shared by different elements. In the example from figure 4-7 the attribute `a1` was shared by element A and element B. Thus, it would be easy for the user to copy every single attribute with that name into the output document, but the "most specific element type" for all `a1` attributes is ambiguous. Both elements A and B are equally as specific. What becomes important then is where the attribute was pasted in the output document and its relation to the previous pastes that were already there. Based on these relations, the best most specific element to use may become clear.

## 4.6.2 Traversing the Paste Tree

It is always trivially possible to generate a transform for a specific output – simply ignore the input and print the output as a constant. But to generate a useful XSLT requires understanding the patterns used to create the output, and generalizing those into a reusable definition. Many indications of these patterns are given through simultaneous editing and then encoded in the paste tree. As a result, the paste tree is responsible for generating the XSLT.

To generate an XSLT, the paste tree is traversed from the root to the leaves. Each parent generates a template that then calls the templates of its children. Children provide each parent a means to call their templates and also define the templates to be called. In this manner, a modular XSLT is created. The implementation of this idea is embodied in three methods that each `XsltNode` must implement, detailed in Table 4.2. The method `toXsltTemplates` defines the templates a node will utilize. These templates may apply children templates or output content. The method `toAnchorPoint` returns a string that the parent node can insert into its template as a means to apply templates from the child.

The final method, `getContext`, aids each node in determining how to display its content. The context of a template in XSLT is similar to the current directory variable when navigating a file system. The context, like a current directory, defines the current environment, thereby enabling relative references. If the context is a `person` element then the XPath expression `@name` references that element's name attribute, the XPath expression `car` reference a child element of the person, and the XPath `car/@make` references the make attribute of that car element. To determine a node's template definitions and how they should be called from within the node's parent, comparisons of the contexts are heavily utilized.

Table 4.2: Methods of `XSLTNode` relevant to XSLT generation.

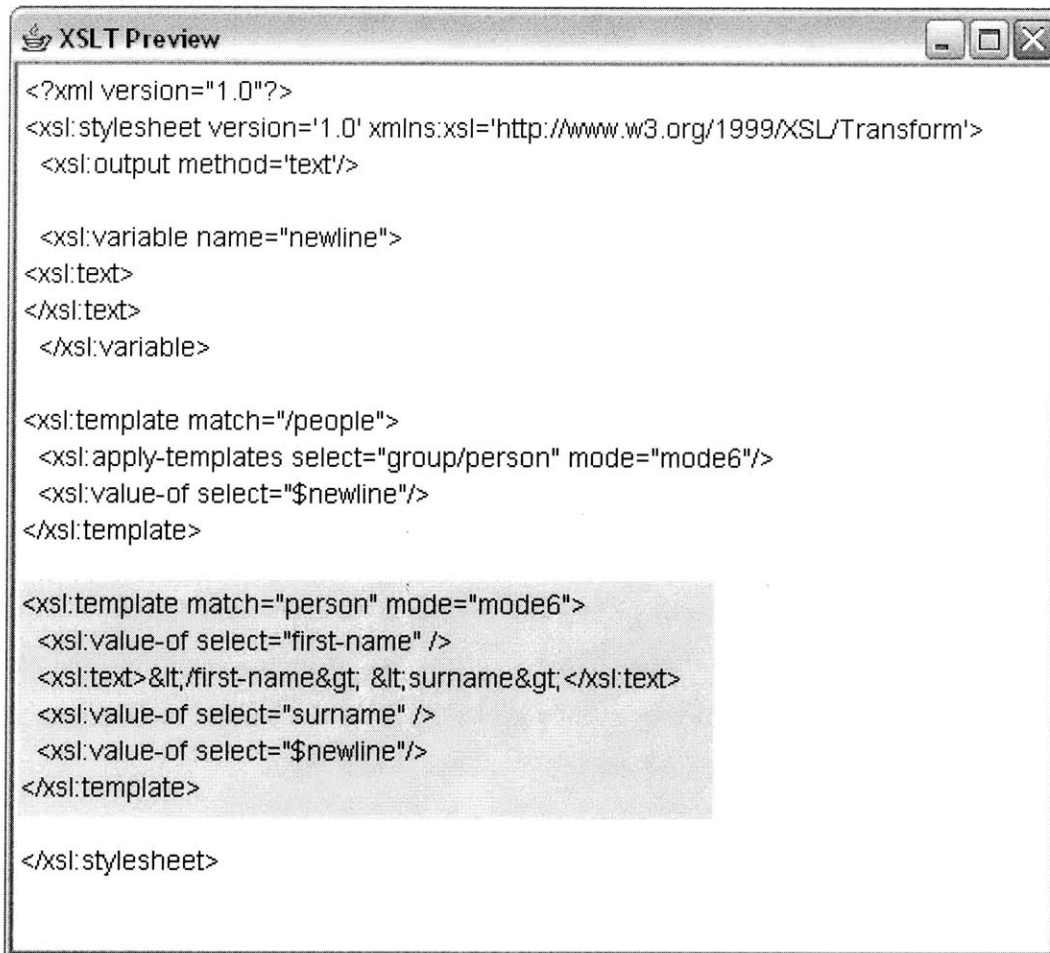
Method Name	Method Description
<code>getContext</code>	Identifies the context of this node
<code>toAnchorPoint</code>	Generates a string to be placed within the parents template
<code>toXsltTemplates</code>	Returns template definitions that may be called from the anchor point

### 4.6.3 Processing the Leaves

During XSLT generation each leaf is responsible for defining a template that will display its contents. Each application of the template will output the text for one of the paste regions. As a prerequisite then, the all of the paste regions must have the same format. Each paste region must contain the same XML features – element values and attributes values – in the same order, and delimited by the same constant text (except for white space). If these required conditions hold, a template is generated that alternates printing delimiting text and feature values, in the order they appear in the paste regions. An example of such a template, derived from the output in figure 4-6 part (a), can be seen in figure 4-8.

### 4.6.4 Processing Internal Nodes

Generating XSLT templates for series nodes is similar to generating templates for parallel nodes. Both types create a template that applies the template of each child node, between which delimiting text may be output. For series nodes the delimiting text will be the region from where the range of one child ends to the range of the next child starts. For parallel nodes, the only delimiting text will be between regions in the last child to regions in the first child, normally a single new line. The rest of the text is incorporated into the child nodes, who will handle it in their own templates. An example of a parallel node's template, derived from the output in figure 4-6 part (b), can be seen in figure 4-9.



```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>

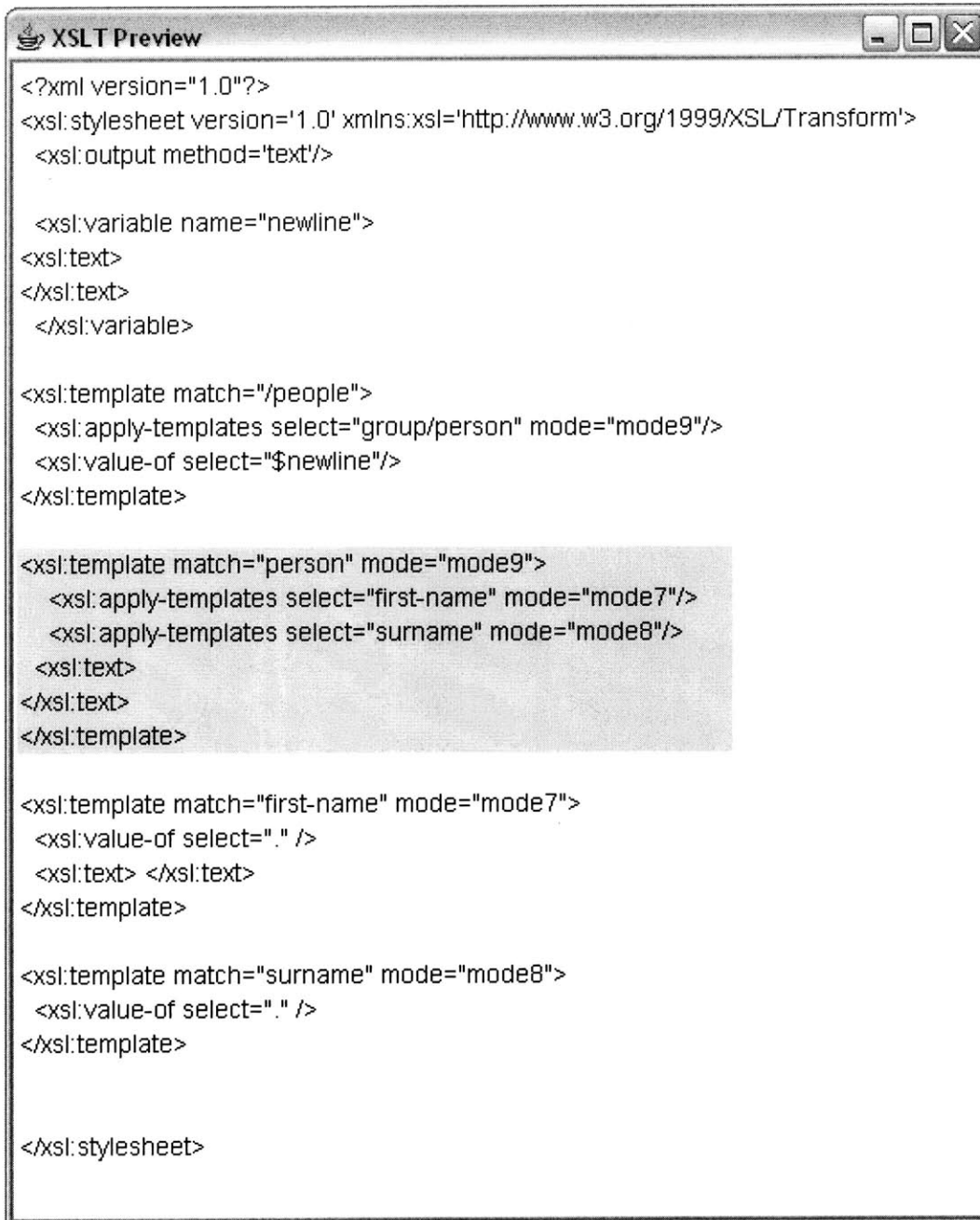
  <xsl:variable name="newline">
<xsl:text>
</xsl:text>
  </xsl:variable>

<xsl:template match="/people">
  <xsl:apply-templates select="group/person" mode="mode6"/>
  <xsl:value-of select="$newline"/>
</xsl:template>

<xsl:template match="person" mode="mode6">
  <xsl:value-of select="first-name" />
  <xsl:text>&lt;/first-name&gt; &lt;/surname&gt;</xsl:text>
  <xsl:value-of select="surname" />
  <xsl:value-of select="$newline"/>
</xsl:template>

</xsl:stylesheet>
```

Figure 4-8: The XSLT generated for the output in figure 4-6 part (a) with input source shown in figure 5-1. The highlighted region identifies the template that was generated from the single paste node for this output document. Each paste region contained two features: the value of the `first-name` element and the value of the `surname` element. Each of these has a corresponding `value-of` element in the XSLT template. Between the `value-of` elements, the text that separates each feature is output.



```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>

  <xsl:variable name="newline">
<xsl:text>
</xsl:text>
  </xsl:variable>

  <xsl:template match="/people">
    <xsl:apply-templates select="group/person" mode="mode9"/>
    <xsl:value-of select="$newline"/>
  </xsl:template>

  <xsl:template match="person" mode="mode9">
    <xsl:apply-templates select="first-name" mode="mode7"/>
    <xsl:apply-templates select="surname" mode="mode8"/>
    <xsl:text>
</xsl:text>
  </xsl:template>

  <xsl:template match="first-name" mode="mode7">
    <xsl:value-of select="." />
    <xsl:text> </xsl:text>
  </xsl:template>

  <xsl:template match="surname" mode="mode8">
    <xsl:value-of select="." />
  </xsl:template>

</xsl:stylesheet>
```

Figure 4-9: The XSLT generated for the output in figure 4-6 part (b) with input source shown in figure 5-1. The highlighted region identifies the template that was generated from the parallel node in this output's paste tree. This template is called from the root template once for each `person` element. It applies a template for each of its children and prints a new line.





## Chapter 5

# Evaluation

### 5.1 Pilot Test

A pilot test was performed to evaluate the usability of XSLT by Demonstration. A total of seven users with varying proficiencies in XML, XSLT and simultaneous editing used the tool to generate two XSLTs each. Three of the users were familiar with simultaneous editing, while the remaining four were not. Only two people had worked with XML before, one of whom was familiar with simultaneous editing, and none of the pilot users had previous experience with XSLT.

The tasks the users were given focused on the generation of an output document given an input XML. The source XML used for all of the tests was a document that organized people into groups, shown in figure 5-1. From this input, the users were asked to create two outputs, as shown in figure 5-2. The first was a document with a list of group names and a separate list of person names. The second was an HTML document containing a table of people's first names, last names, and login IDs.

The same source XML was used for each of the tests so that users could become familiar with it over time. This was meant to generate conditions similar to those where people actually code XSLT directly. In practice the user is normally very familiar with the source XML's structure.

Both groups of users performed the task well and XSLT's were successfully generated for each user output. On average, the first task was completed in around 2 minutes and the second took 4 minutes. Users who were already familiar with simultaneous editing worked faster in general, but new users caught on quickly after figuring out how to copy and paste

```
<people>
  <group title="Engineers">
    <person initials="BC" login="skitty">
      <first-name>Buddy</first-name>
      <surname>Becker</surname>
    </person>
    <person initials="WB" login="wade">
      <first-name>Wade</first-name>
      <surname>Brown</surname>
    </person>
    <person initials="AL" login="dedalus">
      <first-name>Aaron</first-name>
      <surname>Lund</surname>
    </person>
  </group>
  <group title="Docs">
    <person initials="MA" login="mathomas">
      <first-name>Mat</first-name>
      <surname>Thomas</surname>
    </person>
    .
    .
    .
  </group>
</people>
```

Figure 5-1: Part of the source XML document used in the pilot test.

```
XSLT Output:
Group Names:
Engineers
Docs
Maint.

Person Names:
Becker
Brown
Lund
Thomas
Graham
Milburn
Sanchez
```

(a)

```
XSLT Output:
<html><body><table>

<tr><td> Buddy <td> Becker <td> skitty
<tr><td> Wade <td> Brown <td> wade
<tr><td> Aaron <td> Lund <td> dedalus
<tr><td> Mat <td> Thomas <td> mathomas
<tr><td> Dustin <td> Graham <td> dgraham
<tr><td> Mark <td> Milburn <td> aureal
<tr><td> Oswald <td> Sanchez <td> lance

</table></body></html>
```

(b)

Figure 5-2: The two different outputs that users were asked to generate in the pilot test. Output (a) was performed before output (b).

multiple selections. On average, users rated the ease of use at 4 out of 5. Those who were new to simultaneous editing were pleasantly surprised with the efficiency achieved using multiple cursors.

The most common difficulty for users new to simultaneous editing was figuring out how to insert text in parallel. This problem occurred in the second task where the user was asked to create an HTML table. The goal was to put a name on each line, with the text “<tr><td>” before each name, as shown in part (a) of figure 5-3. To accomplish this, all four of the users who were new to simultaneous editing began by typing “<tr><td>” and then pasted the names, giving an the unexpected result shown in figure 5-3 part (b).

The 100 percent error rate hints that the default operation is counter intuitive. To create the list users must perform operations out of the expected order – the names must be pasted first, event though they will follow the text. Also, their intuition from pasting before, where each text region is given its own line, led them to expect a uniform output. One of the users even filled the whole line with “<tr><td> <td> <td>” intending to paste the first names after the first delimitter and the last names after the second. His intuition was that the text would be replicated for each paste item. It might be productive to update the multi-paste action to include the text replication that many of the users expected.

Another problem encountered involved generalizing selections in the output pane. Di-

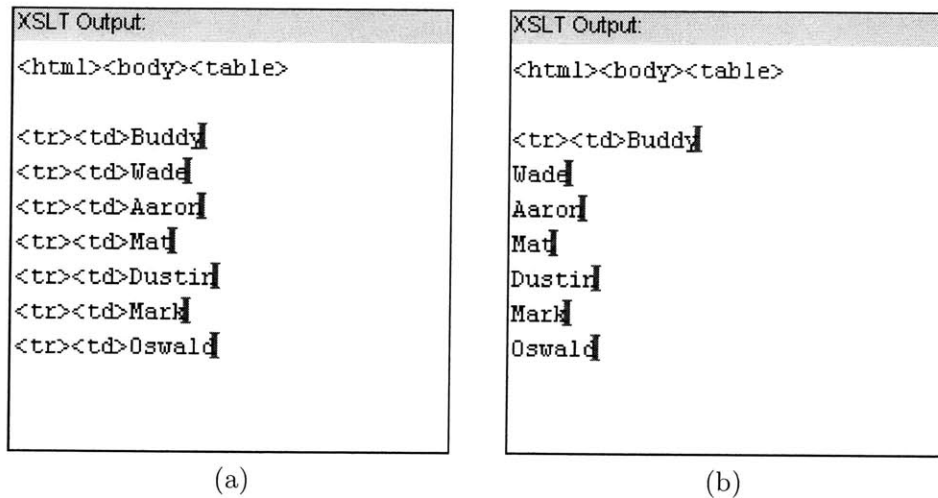


Figure 5-3: Part (a) shows the desired result. Part (b) shows the unexpected results that users new to simultaneous editing encountered.

rectly after a multi-paste into the output pane, one cursor is placed at the end of each paste region in anticipation of parallel editing, as shown in figure 5-4 part (a). If the user then moves the cursor to the beginning of a pasted region, the selection is generalized so that simultaneous editing can occur at the beginning of the pasted region as in figure 5-4 part (b). But if the user moves to the beginning of the very first pasted region, the default hypothesis is no generalization at all, as is shown in figure 5-4 part (c). Without this hypothesis it would be impossible to insert text before the paste without replicating the text in front of every region.

Even though the selection can be generalized to the start of every region by giving more examples, or by showing more hypotheses in the hypothesis pane, the problems it caused for one user highlighted several issues. The user wanted to insert text in front of each of the paste regions, but with the cursor in front of the first region, the selection didn't generalize automatically. The user moved the cursor to the end of the first region, where the selection generalized, and then back to the start, only leading to more confusion since it worked in one case but not the other. Following the test I asked why the user hadn't given more examples, since I had seen him reading the simultaneous editing instructions. He had thought the instructions only applied to the source XML pane, since the hypothesis pane was split horizontally just like the text panes were, matching the instructions to the XML pane. He suggested that the instructions specifically mention that they apply to both edit

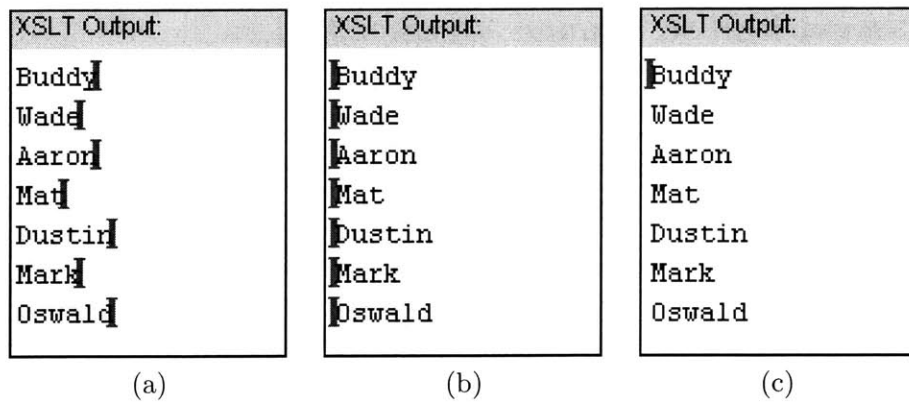


Figure 5-4: Part (a) shows cursor positions following a paste. Part (b) shows the cursor positions users desired, an arrangement that parallels placements they had already seen. Part (c) shows the unexpected result users encountered instead.

panes. It might also have helped him if when the endpoint hypothesis was generated, the active cursor was shown in a different color, the purpose being to focus the user's attention on cursor position within the generalization to prevent assumptions that the generalization is unique.

There were several other helpful user suggestions. Of the users, who had experience using XML, suggested that the input pane be modeled similar to Internet Explorer or FireFox where XML is treated as a tree with the ability to expand and collapse individual elements. This could also be used to make the input XML more readable and reduce the effect of whitespace irregularities. Another user suggestion was to enable double-click for selecting a whole word. This has become somewhat standard functionality, supported by programs including Eclipse and Microsoft Word, and would help the user make selections efficiently.

Following the pilot study, the biggest unaddressed concern is whether users unfamiliar with XSLT will attempt to create unrepresentable transformations. The pilot study assigned users a desired output which they worked to generate. But if given the flexibility, they might exceed XSLT's limitations. The application's arrangement shows no restrictions on the output's structure, but the XSLT that would represent the transformation does have limitations.

## 5.2 Comparison to Human Written XSLTs

To evaluate XSLT by Demonstration five pre-existing XSLTs were chosen, and the replication of their definitions was attempted using the tool. The results are detailed below:

### 5.2.1 Weather Report to HTML

The XSLT in this case was designed to take a data document describing a weather report and display parts of it in an HTML document (figure 5-5 (a)). XSLT by Demonstration was unable to handle the required output structure though. The output was to contain several locations reports for each weather reading (figure 5-5 (b)). But to accomplish this in the output would have required intelligently pasting five regions into two regions, an operation that is currently unsupported. This problem will come up again in other examples and will be discussed in more detail in section 5.2.6.

An XSLT was generated for a similar output instead and compared to the human created XSLT. They each contained similar structure, however the auto-generated XSLT placed all text inside of `<xsl:text>` elements and substituted all angle brackets with their corresponding entities, leading to a more complex looking and less readable XSLT. The tool does this to ensure that the XSLT is well formed, even when the desired output is not. But when the output is XML or XHTML, the XSLT generator should recognize that and simplify the display. Further, when outputting XML or HTML, it is not as important to exactly replicate all whitespace, and so this constraint may be relaxed to provide a more readable output.

### 5.2.2 Business Cards to HTML

This XSLT transforms an XML file containing business card descriptions, creating an output HTML file to display them (figure 5-6). The XSLT generator worked well for this example, generating a successful XSLT. As before, it was somewhat less readable than the human-generated XSLT.

### 5.2.3 Address Book to XML Dialect

In this XSLT an address book data XML is converted to a different representation. Specifically, an attribute is transformed into an element (figure 5-7). The XSLT generator handled

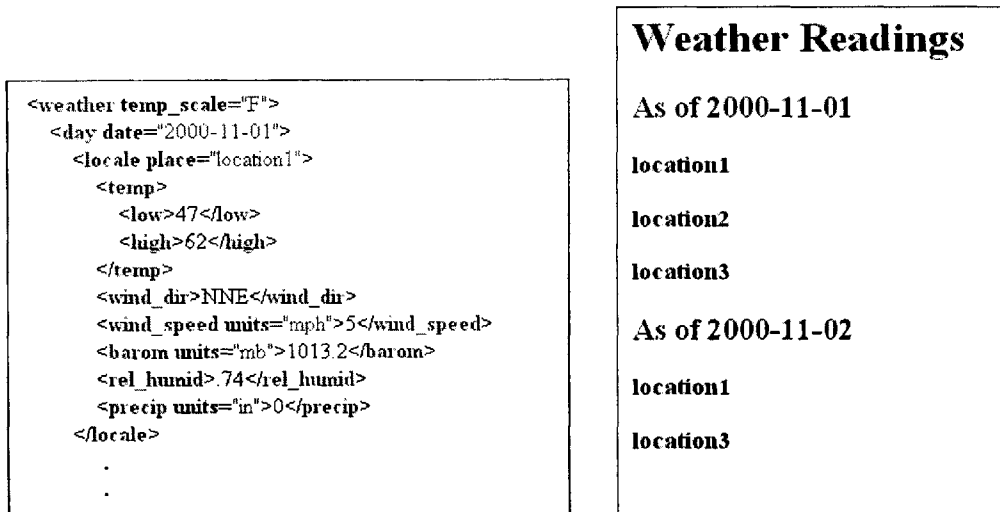


Figure 5-5: The weather reporting example: (a) input XML; (b) output HTML.

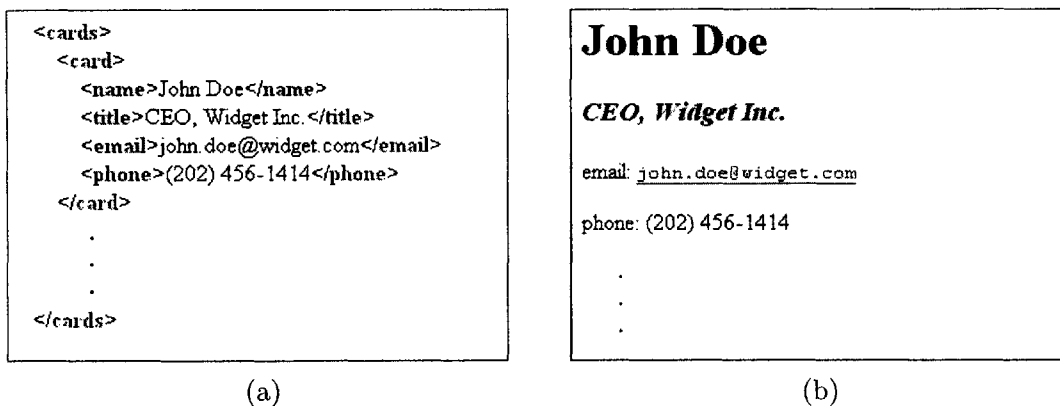


Figure 5-6: The business card example: (a) input XML; (b) output HTML.

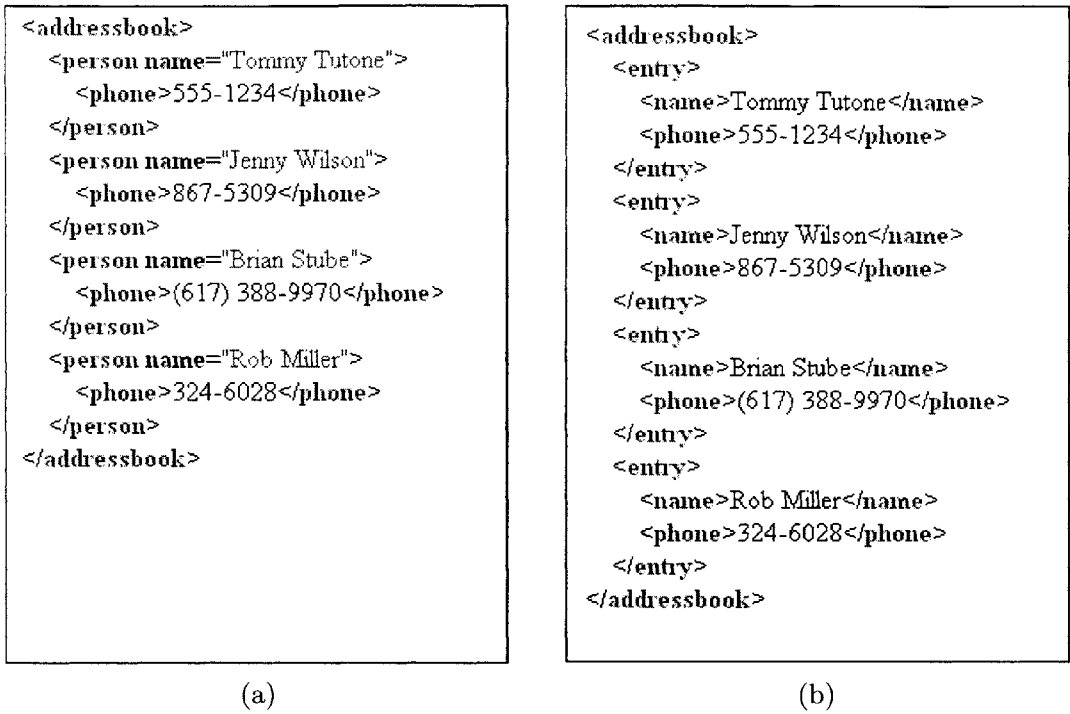


Figure 5-7: The address book example: (a) input XML; (b) output XML.

this case well, generating an XSLT with the same form as the human created one.

### 5.2.4 People Grouping to Text

Here an XML document representing groups of people was to be transformed into a text document listing the people and the group they belong to (figure 5-8). It worked well at generating the people, but did not have the ability to perform a few-to-many intelligent paste of group names into a list of members.

### 5.2.5 Recipe to HTML

The input XML for this case was mark-up for a recipe, with an HTML display as the desired output (figures 5-9 and 5-10). The XSLT by Demonstration tool had trouble dealing with text that was split by mark up. Although pastes could be made with the mark up removed from the text, generating an XSLT failed for this case.



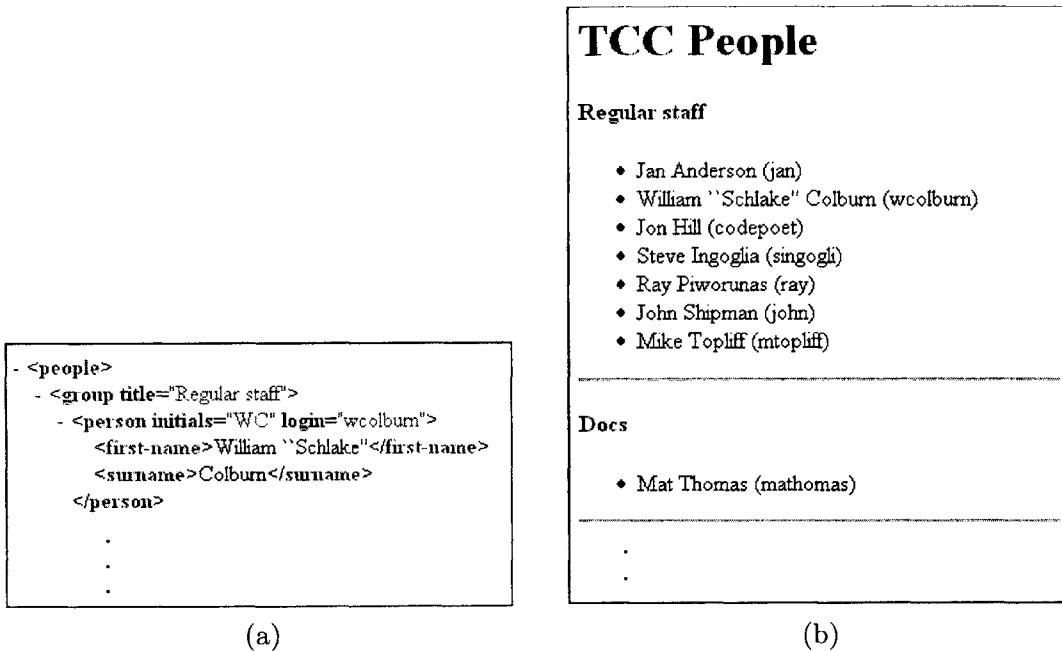


Figure 5-8: The people-groups example: (a) input XML; (b) output text.

```

<recipe>
  <title original="Rogan Josh">Lamb in Fragrant Garlic Cream Sauce</title>
  <author source="Classical Indian Cooking">Julie Sahni</author>
  <subrecipe>
    <para>
      Heat the ghee in a small frying pan over high heat. When it is very hot, add the garlic, and
      stirring rapidly, fry for
      <time>15 seconds</time>
      .
    </para>
    <ingredients>
      <item>
        4 peeled and quartered medium-sized onions (about 1 lb)
      </item>
      <item>2 Tbsp. chopped fresh ginger root</item>
      <item>2 Tbsp. ground coriander</item>
      <item>3/4 tsp. red pepper, or to taste</item>
      <item>2.5 C. plain yogurt</item>
      <item>0.5 C. sour cream</item>
      <item>1 Tbsp. Kosher salt</item>
    </ingredients>
    .
  </subrecipe>

```

Figure 5-9: Portions of the input XML file for the recipe example.

## Lamb in Fragrant Garlic Cream Sauce (*Rogan Josh*)

by Julie Sahni (from *Classical Indian Cooking*)

Cut the meat into 1.5" cubes and place them in large bowl. Pour the marinade and the ghee over it. Mix thoroughly to coat the meat pieces with the marinade. Cover, and let the meat marinate for at least **30 minutes** at room temperature, or **2 hours** in the refrigerator. (Remove from the refrigerator about **30 minutes** before you are ready to cook the meat.)

- 4 peeled and quartered medium-sized onions (about 1 lb)
- 2 Tbsp. chopped fresh ginger root
- 2 Tbsp. ground coriander
- 3/4 tsp. red pepper, or to taste
- 2.5 C. plain yogurt
- 0.5 C. sour cream
- 1 Tbsp. Kosher salt

Figure 5-10: Excerpts of the recipe example's output HTML.

### 5.2.6 Summary of Results

From these examples it can be seen that the XSLT by Demonstration tool works better in cases where XML stores data in attributes and leaf elements, and worse in cases where XML is used to mark up text. In data XML it is easier to identify the selection, making the tool easier to use. Also, the relationships between elements are more straightforward, making the XSLT easier to generate. This last point can be seen in comparing the recipe XML to any of the others. In the recipe XML the content of `para` elements was broken up by `time` elements. This partitioning was difficult for the XSLT generator to model. In contrast, the data XML documents contained content only in leaf elements and in attribute values.

The most serious challenge to generating XSLT's from data XML is inflexibility of the multi-paste operation, as was seen in the people-grouping example and the weather example. There needs to be a way to paste a group of children into their corresponding parents, such as in figure 5-12 part (a), and a way to paste a group of parents into their corresponding children, as in figure 5-12 part (b). These operations could be supported by extending paste to remedy incompatible pastes with intelligent inferences.

Finally, to increase the readability of the generated XSLT, and thereby also promote

```
<people>
  <group title="Engineers">
    <person initials="BC"/>
    <person initials="MA"/>
  </group>
  <group title="Maint.">
    <person initials="DG"/>
    <person initials="MM"/>
    <person initials="OS"/>
  </group>
</people>
```

Figure 5-11: A data XML document containing parents with many children.

confidence in its correctness, the state of the output must be examined before generating the XSLT. If the output happens to be well-formed XML, then it should be possible to generate an XSLT that minimizes its use of the `xs1:text` element. Also, since XML is often viewed and edited in a processed tree representation, the XSLT generator can reduce its focus on exact replication of whitespace in exchange for a more readable transform definition.

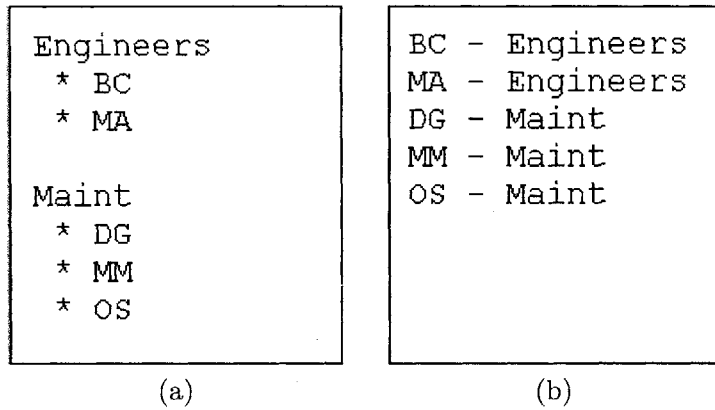


Figure 5-12: Output structures from the input in figure 5-11 that are unsupported by the current multi-paste operation. Part (a) shows an output that would require pasting 5 child regions into 2 parent regions. Part (b) shows an output that would require distributing 2 parent regions into 5 child regions.

## Chapter 6

# Conclusion

The XSLT generator evaluated in this thesis affords several advantages when compared to the standard means for defining XSLTs. By providing a PBD environment for generating XSLT, it was shown that even users new to simultaneous editing could quickly generate XSLT definitions. The low-level programming details, such as escaping special characters, were taken care of automatically, enabling the user to focus on the desired results rather than the process necessary to achieve them.

However, text editing cannot easily generate all possible XSLTs. Consider, for example, an XSLT that, for each node, prints out the node's name and the name of its parent. This is straight-forward to do in XSLT, and it is even easy to order the results by in document order, reverse document order, or based on some other feature, such as alphabetical order. To accomplish this in text editing, one might first paste the name of every element into the output. But then how would the names of the parents be selected and pasted? Further, in this setting it would be difficult to indicate a order based on the tree structure. Other problematic outputs exist too, but most share a focus on processing each node in the same way. By contrast, outputs based more on the XML's structure can be generated with XSLT by Demonstration.

Even when desired outputs are achievable, mark-up XML documents are hard to work with. Having their text broken up into multiple sections makes it difficult both for the user to work with intuitively and for the XSLT generator to represent.

Finally, in the transform creation processes the user is constrained to begin with an input XML that contains all features planned for use in the XSLT. Only features in the

input XML can be part of a transform generated from XSLT by Demonstration. Hence, the user must begin with a sufficiently feature-rich input XML.

## **6.1 Future Work**

What follows is a discussion of factors whose resolution may have a significant impact on the XSLT generating process.

### **6.1.1 Undo**

The undo feature is a requirement for text editing applications. In addition to being a useful tool, its presence also encourages a user to explore an application with confidence, knowing that any unexpected behavior can simply be undone. In the domain of simultaneous editing to produce XSLTs, supporting undo is likewise necessary.

However there are several design hurdles that must be overcome before it can be utilized. The first of these problems is that a node's ranges expand and contract based on the user's sequence of edits, and the manner in which they do so is not modeled by LAPIS's default functionality. Hence, the different versions that each range evolves through must be recorded so that if an undo is executed these prior versions can be revisited.

Another, more difficult problem to overcome is that editing operations can permanently alter the structure of the paste-tree. Edits in the middle of a simple-nodes range will split that node into two and create a new parallel node to contain them. Pastes from the original XML generate new nodes as well. To undo these actions the structure of the paste tree must also be revisited.

A way to describe the progress of a tree over time and to associate the intermediate structures with document versions is required. Perhaps the simple solution of copying the structure at each version and storing it with the undo feature would suffice, despite the inefficient use of space. No matter the implementation though, undo must be supported.

### **6.1.2 Error Reporting**

There are editing sequences that can corrupt the output document so that it becomes impossible to create a useful XSLT. Such an occurrence is possible, for example, when a user pastes two selections in parallel (made possible because they have the same cardinality) but

when no logical relation between the regions exists in the original XML. Although individual regions are paired up between the two multi-pastes, there is no XML structure associated the regions

There are several possible ways for dealing with such erroneous document structures. The first it to ignore it, and instead of reporting the occurrence, represent the two pastes as a constant string and encode that into the XSLT. However this method is undesirable because it gives the user an illusion of success, but rarely the desired behavior.

Other options involve reporting the occurrence to the user. The current implementation of the XSLT generator reports the error in the generated XSLT, thereby allowing the user take advantage of other parts that may have been successfully generated. Another option would be to report the error at paste time, when the problem first arises. Such an alert could even abort the XSLT-incompatible paste and prevent it from occurring. This option would help to identify the problem early, and thus, seems to be the optimal solution. It would require a fundamental change in the tool's interaction with LAPIS though. Its currently acts as a listener to document changes, but this error catching solution would require XSLT by Demonstration to become more closely coupled with the low level document editing process.

### **6.1.3 Editable XML Document**

In the current implementation of XSLT by Demonstration the original XML input pane is non-editable. The content of that document remains constant over time. However a more flexible arrangement would treat the input pane as a scratch pad, allowing the use to edit the XML document prior to copying selections into the output.

The hurdle preventing this more flexible design relates to the inference engine that generalize selections in the XML document. Specifically, the problem is that as a document is edited the inference engine updates as well. But after a paste from the XML document into the output pane occurs, clicks in the paste region are mapped back to use the XML inference engine. The inference engine at the paste time should be used for the generalization, but instead the most recent version is always used. As the XML document evolves so too will this inference, and then clicks into the pasted regions will erroneously reflect these updates.

To effectively deal with this design issue the generalization procedure should be expanded to accept a document version, indicating which version to use in the generalization

procedure.

#### 6.1.4 Snapping Selections

Selections from the original XML document should ignore whitespace, snapping to contained non-whitespace characters. To select items in the current design the user must indicate the precise starting and ending characters, but these are small target zones and Fitt's Law dictates that making these selections will take time and cause errors. Ignoring surrounding whitespace in a selection would expand the selection zone, making it easier for users to make an accurate selection.

However this improvement could not be applied in all cases. If the user wanted to select an attribute value that ended with a space, ignoring that whitespace would be erroneous. Hence, intelligence would be required to determine when white space should be ignored.

For similar purposes, double clicking a word should be made to select it too.

#### 6.1.5 Inferred Pastes

Currently LAPIS supports the ability to paste  $n$  regions into  $n$  regions, 1 region into  $n$  regions, and  $n$  regions into 1 region. But the requirements of many transforms conflict with such stringent constraints. To illustrate this, consider the source XML document and two possible outputs, shown in figure 6-1. Generating those outputs easily through multi-pastes is impossible given the current state of simultaneous editing. There are four **person** elements in the document and two **group** elements in the document, leaving no way to combine them in a paste operation.

Given these examples, and the relative frequency in which they occur, it is desirable to overcome these incompatibilities. A solution that would seamlessly fit into the normal paste functionality is to remedy incompatible pastes automatically using inferences based on the XML document's structure. In the first incompatibility example, when the group names were being appended to the member names, it could have been identified that every **person** element is a child of a **group** element. Hence, it would be logical to paste the appropriate group name following each person name.

Allowing such pastes would require new ways to represent the paste relationship; series and parallel XsltNodes would not be able to handle them. Given the flexibility this feature would provide XSLT by Demonstration, expanding the paste tree representations is justified.



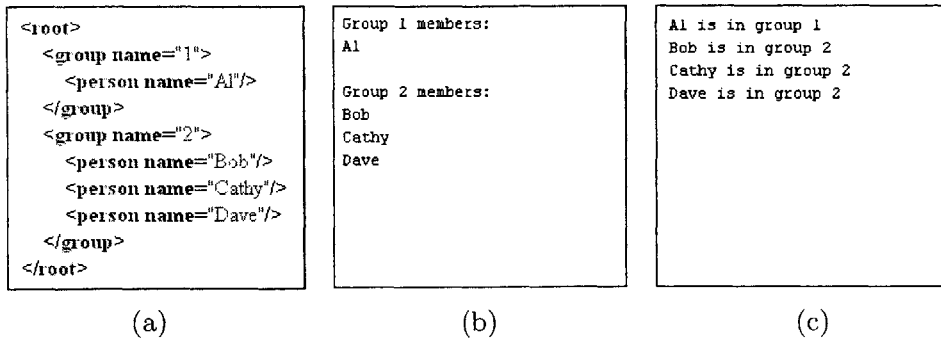


Figure 6-1: Part (a) shows an XML document organizing people into groups. Part (b) shows an XSLT output that would require intelligently pasting four regions into two. Part (c) shows an XSLT output that would require intelligently distributing two regions into four.

### 6.1.6 Counting

XSLT provides functionality to count items, such as the number of children of an element or the total number of attributes in an XML document. This is a relatively simple but also important feature whose incorporation into the XSLT generator would add value.

### 6.1.7 Lookup Table

XML often contains data in a form not meant for display. An example of this would be an XML document that stores US states in abbreviated form, but for the output state names should be spelled out. Lookup tables can solve problems like these by defining a mapping between input values and output values.

In addition to being a useful tool, the implementation of lookup tables in XSLT is complicated and hard to define. Hence, simplifying the use of lookup tables would add value to XSLT by Demonstration by making it more powerful and easier to use. The process involved in defining the lookup table could even be streamlined by using multiple selections in the input XML to define input or output sets.

### 6.1.8 Sorting

Another important function provided by XSLT is the ability to perform sorting in the transformation. But like the lookup table, actually coding it into an XSLT by hand can be a challenge. Hence, supporting this functionality in the XSLT generator would be both

necessary and valuable. LAPIS already provides support for sorting a document's regions, but XSLT generation would require its customization and integration.

If reordering is implemented pasting from the source XML becomes more complex. Care must be taken to ensure that pasted regions are inserted in sorted order, not the original XML document order.

## **6.2 Summary**

XSLT by Demonstration has great potential to simplify the creation of transformation definitions. In the pilot tests, knowledge of XSLT was not required – the problem had been reduced to generating the desired output text – and simultaneous editing, the foundation of this tool, proved intuitive and easy to learn. Although some XSLT definitions are impractical to generate using this method, defining XSLTs based on data XML can be much easier than before. With the addition of an undo-feature and a more flexible multi-paste, XSLT by Demonstration would be a powerful and easy to use alternative for creating XSLTs.

# Bibliography

- [1] Robert C. Miller. *Lightweight Structure in Text*. PhD thesis, Computer Science Department, Carnegie Mellon University, May 2002. Published as CMU Computer Science technical report CMU-CS-02-134 and CMU Human-Computer Interaction Institute technical report CMU-HCII-02-103.
- [2] *LAPIS*, <http://graphics.csail.mit.edu/lapis/>
- [3] *XSLWiz 2.0*, described in [http://www.sys-con.com/xml/wbg/CurrentSearch\\_Detail.cfm?ID=1105](http://www.sys-con.com/xml/wbg/CurrentSearch_Detail.cfm?ID=1105)
- [4] Cape Clear Software Inc, *CapeStudio 3.0*, <http://www.capeclear.com/products/studio/index.shtml>
- [5] IBM Alphaworks, *XSLerator*, <http://alphaworks.ibm.com/tech/xslerator>. March 2001.
- [6] IBM Alphaworks, *XSLbyDemo*, <http://www.alphaworks.ibm.com/tech/xslbydemo>
- [7] Allen Cypher. Eager: Programming repetitive tasks by demonstration. In Allen Cypher, editor, *Watch What I Do: Programming by Demonstration*, pages 205-218. MIT Press, 1993.
- [8] ] Robert Nix. Editing by example. *ACM Transactions on Programming Languages and Systems*, 7(4):600-621, October 1985.
- [9] Tessa Lau, Steven Wolfman, Pedro Domingos, and Daniel S. Weld. Learning repetitive text-editing procedures with SMARTedit. In Henry Lieberman, editor, *Your Wish Is My Command: Giving Users the Power to Instruct Their Software*, pages 209-226. Morgan Kaufmann, 2001.
- [10] Brad A. Myers. *Demonstrational interfaces: A step beyond direct manipulation*. IEEE Computer, pages 61-73, August 1992.