XSLTGen: A System for Automatically Generating XML Transformations via Semantic Mappings*

Stella Waworuntu and James Bailey

NICTA Victoria Laboratory
Department of Computer Science and Software Engineering
The University of Melbourne, Australia
{*stellavw, jbailey*}*@cs.mu.oz.au*

Abstract. XML is rapidly emerging as a dominant standard for representing and exchanging information. The ability to transform and present data in XML is crucial and XSLT is a relatively recent programming language, specially designed to support this activity. Despite its utility, however, XSLT is widely considered a difficult language to learn.
In this paper, we present a novel system called XSLTGen, an automatic XSLT Generator. This system automatically generates an XSLT stylesheet, given a source XML document and a desired output HTML or XML document. It allows users to become familiar with and learn XSLT stylesheets, based solely on their knowledge of XML or HTML. Our method for automatically generating XSLT transformations is based on the use of semantic mappings between the input and output documents. We show how such mappings can be first discovered and then employed to create XSLT stylesheets. The results of our experiments show that XSLTGen works well with a number of different varieties of XML and HTML documents.

1 Introduction

XML (eXtensible Markup Language) [4] is rapidly emerging as the new standard for data representation and exchange on the Web. As the medium for communication between applications, an ability to transform XML to other data representations is essential. This data conversion can be performed by a language called XSLT (eXtensible Stylesheet Language: Transformations) [7]. XSLT plays an important role in transforming XML documents into HTML, text, or other types of XML documents. In this paper, we focus on transformations from XML to HTML (HyperText Markup Language) [26], since we are motivated by publishing applications, but our techniques are also applicable for XML to XML transformations.

Despite its capability of transforming documents having a certain structure, e.g. XML documents, into an HTML representation, XSLT is a relatively new

* Parts of results of this paper appeared in [30].

language, and is widely considered difficult to learn [19]. Rendering to HTML using XSLT requires skills and knowledge of both XSLT programming and Web page styling. Our focus in this paper is on the development of algorithms and tools that can generate XSLT stylesheets automatically, given a user provided source XML document and a user provided target HTML document.

Automatic XSLT generation is an extremely useful facility for students and Web developers in the process of learning XSLT. With such a tool, they are able to see and understand how the XSLT stylesheet should look, in order to transform a particular XML document into a desired HTML document. In addition, this tool can also be useful for aiding the XSLT development process. Programmers may use the automatically generated XSLT stylesheet as a starting point for something more complex.

In this paper, we present *XSLTGen: An Automatic XSLT Generator*, a novel system that automatically generates an XSLT stylesheet, given a source XML document, and a desired output HTML document. The generated XSLT stylesheet contains rules needed to transform the given XML document to the HTML document and can be applied to other XML documents with similar structure, or to the given XML document after updates to it have been applied. The important feature of this system is that users can generate an XSLT stylesheet based solely on their knowledge of XML and HTML, i.e. users only need to create a desired output HTML document based on an input XML document. Moreover, users do not have to know anything about the syntax or programming of XSLT, or be aware of the XSLT rule generation process.

A naive solution to the problem of automatic XSLT generation is to create an XSLT stylesheet consisting of only one template rule, whose pattern matches the XML root element and whose template contains the HTML document markup (in other words create a stylesheet which is very specific to the desired output). This naive approach has a major drawback in terms of reusability. This stylesheet is specific for transforming the given XML document only and could not be used to transform other XML documents having similar structure. In contrast, we are interested in generating a more generic stylesheet, which can then be reused to transform other XML documents with similar structure. A more detailed discussion and illustration of both the naive and more generic solutions is given in Sect. 3.

In this paper, we show how XSLT stylesheets can be automatically generated by first discovering semantic mappings between the input and output. We make the following contributions:

1. We describe the use of text matching and structure matching for finding semantic mappings between an XML document and an HTML document generated from this XML document.
2. We introduce *sequence checking* to the matching context, and show that it enables the system to discover 1-m mappings between the two documents, in addition to its capability of discovering 1-1 mappings.
3. We describe a fully automatic XSLT generation system that generates XSLT rules based on the semantic mappings found.

4. We describe a technique for improving the accuracy of the XSLT stylesheet generated, that examines the differences between the original HTML document and the one produced by applying the generated XSLT stylesheet back to the XML document.
5. We conduct experiments on a number of datasets to validate the matching accuracy and quality of XSLT stylesheets generated by XSLTGen. The results show that XSLTGen works well with different varieties of XML and HTML documents.
6. This is the first paper that we are aware of that describes completely automatic XSLT generation from a source XML and a target HTML document.

This paper is structured as follows. In Sect. 2, we give a brief overview of XSLT and describe important definitions and terminology used in the paper. In Sect. 3, we present the definition of the problem of generating XSLT automatically. In Sect. 4, we present an overview of XSLTGen and then describe the details of each technique involved in the XSLTGen system. In Sect. 5, we conduct experiments to measure the similarity between the original HTML document and the one generated by the resulting XSLT stylesheet. We then evaluate the performance of XSLTGen. In Sect. 6, we survey related work in the field. Finally in Sect. 7, we offer concluding remarks and discuss possible extensions to the current system.

2 Background and Terminology

In this section, we provide a brief overview of XSLT. For a more complete description of XSLT, the reader is directed to [7], and [16]. Following this, we describe important definitions and terminology used in this paper.

2.1 XSLT

XSLT is a "high-level declarative language for transforming XML documents into other XML documents or HTML documents" [7]. The dominant features of XSLT as a declarative language are that it is a rule-based language, where the rules are not arranged in any particular order, and it is side-effect free which enables XSLT rules to be called any number of times and in any order.

XSLT uses XML syntax. The root element is `<xsl:stylesheet>`, which must include a namespace declaration for XSLT. The optional `<xsl:output>` element tells the type of the target document. The root element is then filled with *template rule*s, which describe how to transform elements in the source document, in this case, XML document. Each template rule consists of two parts: a *pattern* and a *template*. The pattern describes which XML element nodes should be processed by the rule. In some cases, patterns are specified using XPath expressions [8]. On the other hand, the template describes the HTML structure that should be generated when nodes that match the pattern are found. In an XSLT stylesheet, a template rule is represented by an `<xsl:template>` element. The

pattern is the value of its `match` attribute, and the template is the element's content.

The template may contain a sequence of text nodes and *literal result elements*[1] to be copied to the output, and instructions to be executed according to the rules of the particular instruction. The complete set of XSLT instructions can be found in Sect. 7 of [7]. The two XSLT instructions that will be used very often in the XSLT stylesheets generated by XSLTGen are `<xsl:value-of>` and `<xsl:apply-templates>`. The `<xsl:value-of>` instruction extracts the data content of an XML element and inserts it into the output. It has a `select` attribute which consists of a pattern. The `<xsl:apply-templates>` instruction finds all nodes that match the `select` attribute pattern, and processes each node in turn by applying the template rule that matches the node.

As described earlier, XSLT is a language specifically designed for transforming the structure of XML documents. The transformation process works as follows. A list of nodes from the source document is processed to create a result tree fragment. The result tree is constructed by processing a list containing just the root node. Within a list of source nodes, each list member is processed in order and the result tree structures are appended. A node is processed by finding all the template rules with patterns that match the node, and choosing the best amongst them; the template of the chosen rule is then instantiated with the node as the current node and with the list of source nodes as the current node list. A template typically contains instructions that select an additional list of source nodes for processing. This process is continued recursively until the list of source nodes is empty.

2.2 Definitions and Terminology

We now present definitions and terminology that will be be useful in describing the semantic mappings and their generation.

Let $m:\ (m.x, m.h)$ denote a mapping between an element in the source XML document to one or more elements in the output (destination) HTML document. The XML component of m is denoted by $m.x$, while the HTML component of m is denoted by $m.h$. Note that $m.h$ can be a sequence (concatenation) of one or more elements, while $m.x$ must be a single element. e.g. (`poem`,`body`) and (`line[1]`,`li++br`) are both mappings. If $m.x$ is an ATTRIBUTE_NODE, $owner(x)$ is defined to be the XML node that owns $m.x$.

Two mappings m_1 and m_2 are *distinct* if both $m_1.x.name \neq m_2.x.name$ and $m_1.h.name \neq m_2.h.name$ (where *name* refers to the tag name of the appropriate element); otherwise they are *coincide*. e.g. (`poem`,`body`) and (`line[1]`,`li++br`) are distinct mappings, whereas (`poem`,`body`) and (`poem`,`li++br`) coincide.

Exact mappings and substring mappings will be used later to describe textual correspondences in the XML source and HTML output. An *exact mapping* is a mapping e such that, $e.x$ is a TEXT_NODE or an ATTRIBUTE_NODE, $e.h$

[1] A literal result element is an element within a template that cannot be interpreted as an XSLT instruction.

is a TEXT_NODE, and $e.x.value \equiv e.h.value$ (where *value* refers to the string content of the appropriate element and $\equiv$ indicates that the two strings being compared have exactly the same characters located at the same positions, after leading and trailing whitespaces have been removed from the strings. e.g. (`"XML is good"`,`"XML is good"`) is an exact mapping.

A *substring mapping* is a mapping s such that, $s.x$ is a TEXT_NODE or an ATTRIBUTE_NODE, $s.h$ is a TEXT_NODE, and $s.x.value$ is a substring of $s.h.value$. In addition to this, the following conditions must be satisfied by a substring mapping in order to rule out meaningless cases, such as a mapping between an XML text "`the`" and an HTML text "`there`" since it is highly unlikely that an HTML text "`there`" is generated from an XML text "`the`":

1. if $s.x.value$ starts with a non-letter and non-digit character, then the character preceding its occurrence in $s.h.value$ (if any) must be either a letter or a digit; otherwise, the preceding character must be both non-letter and non-digit. And,
2. if $s.x.value$ ends with a non-letter and non-digit character, then the character following its occurrence in $s.h.value$ (if any) must be either a letter or a digit; otherwise, the following character must be both non-letter and non-digit.

(`"XML"`,`"XML is good"`) is an example of a substring mapping.

Special HTML elements are the elements `br` and `hr`, which are used to separate text in HTML document.

An *extra node* is a node that does not have any matching node in the other document. *Extra XML nodes* are those XML nodes that are ignored when generating the HTML document, while *extra HTML nodes* are HTML nodes that are added at the time the HTML document was generated and are not constructed from any part of the XML document. Examples of extra nodes are provided later in Table 2.

Let N be an XML or HTML node. We define the *precise node* of N, denoted by $precise(N)$, to be the node that is used to represent a transformation in the XSLT template rule match. For a mapping m, if $m.x$ is a TEXT_NODE, $precise(m.x)$ is the parent of $m.x$, whereas if $m.x$ is an ATTRIBUTE_NODE, $precise(m.x) = m.x$. Finding the precise node for $m.h$ is slightly more complicated since we need to look at the neighbourhood surrounding $m.h$, for the existence of *special* HTML elements. $precise(m.h)$ is discovered as follows:

1. If the next sibling of $m.h$ is a *special* HTML ELEMENT_NODE h_s, then $precise(m.h) = m.h$ `++` h_s.
2. If $m.h$ has no next sibling or the next sibling of $m.h$ is not *special* and $m.h$ has ELEMENT_NODE siblings, then $precise(m.h) = m.h$.
3. Otherwise, $precise(m.h)$ is the highest ancestor of $m.h$ such that each node lying between $precise(m.h)$ and $m.h$ path only has one *non-extra* child.

We also define the *sequence* of element N, denoted by $seq(N)$ to be a DTD[2] of N if N is the root of a document (e.g. In Figure 1, a possible DTD for the root element is `(author,date,title,stanza*)`). Otherwise, let D' be a DTD of the parent of N, then $seq(N)$ is equal to $trim_N(D')$. Since it is possible that D' involves symbols that represent elements other than N, the function $trim$ is employed here. $trim_N(E)$ removes both the largest prefix not containing the element N and the largest suffix not containing the element N from its argument E, which is a regular expression. Here, E is viewed as an ordered conjunction and so the largest prefix of E not containing N is the maximal prefix of conjuncts in E that don't contain N. Similar for the largest suffix. e.g. $trim_p(a, b, (e|f), p*, (p|a), a) = p*, (p|a)$.

3 Problem Formulation

In this section, we further formulate the problem of generating an XSLT stylesheet automatically from a source XML document and a desired output HTML document. We also discuss both the naive and non-naive ways of solving our problem.

Our primary focus in this paper is to automatically construct an XSLT stylesheet that transforms a source XML document to a desired output HTML document. Thus, the problem addressed in this paper can be stated as follows.

Problem statement. Given an XML document drawn from a class of XML documents, and an HTML document to be generated from the XML document, generate an XSLT stylesheet which contains rules needed to transform the given XML document into the HTML document and which also can be applied to other XML documents of the same document class.

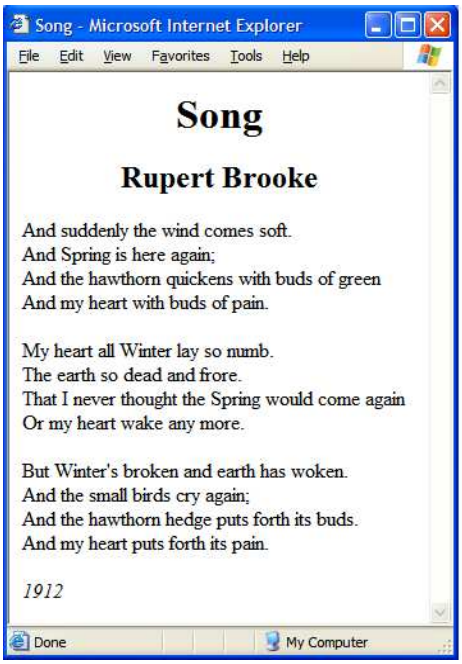
The following example best illustrates the problem described above. Figure 1 shows an XML source for a poem *Song*. We want to create an XSLT stylesheet such that the poem appears in the browser as shown in Fig. 2.

A naive solution to the above problem is to create an XSLT stylesheet consisting of only one template rule whose pattern matches the XML root element `poem` and whose template contains the HTML document markup. Figure 3 shows this naive solution. However, this naive approach has a major drawback in terms of reusability. This solution is very specific for transforming the given XML document only and could not be used to transform other XML documents having similar structure. A particularly important kind of structurally similar document is an updated version of the original (after insertions or deletions). i.e. This naive solution would no longer be applicable if the XML document had another stanza added. This violates the initial purpose of developing XSLTGen, which is to generate a stylesheet based on the supplied source XML and desired output HTML documents, so that the resulting stylesheet can be reused with other XML documents having a similar structure, or with the original document after updates to it have been applied.

[2] A DTD (Document Type Definition) is a grammar for describing the structure of a document. A DTD constrains the structure of an element by specifying a regular expression that its sub-element sequences have to conform to.

```
<poem>
  <author>Rupert Brooke</author>
  <date>1912</date>
  <title>Song</title>
  <stanza>
    <line>And suddenly the wind comes soft.</line>
    <line>And Spring is here again;</line>
    <line>And the hawthorn quickens with buds of green</line>
    <line>And my heart with buds of pain.</line>
  </stanza>
  <stanza>
    <line>My heart all Winter lay so numb.</line>
    <line>The earth so dead and frore.</line>
    <line>That I never thought the Spring would come again</line>
    <line>Or my heart wake any more.</line>
  </stanza>
  <stanza>
    <line>But Winter's broken and earth has woken.</line>
    <line>And the small birds cry again;</line>
    <line>And the hawthorn hedge puts forth its buds.</line>
    <line>And my heart puts forth its pain.</line>
  </stanza>
</poem>
```

Fig. 1. XML source for poem *Song*




```
<html>
  <head><title>Song</title></head>
  <body>
    <div align="center"><h1>Song</h1></div>
    <div align="center">
      <h2>Rupert Brooke</h2>
    </div>
    <p>
      And suddenly the wind comes soft.<br>
      And Spring is here again;<br>
      And the hawthorn quickens with buds of
        green<br>
      And my heart with buds of pain.<br>
    </p>
    <p>
      My heart all Winter lay so numb.<br>
      The earth so dead and frore.<br>
      That I never thought the Spring would
        come again<br>
      Or my heart wake any more.<br>
    </p>
    <p>
      But Winter's broken and earth has
        woken.<br>
      And the small birds cry again;<br>
      And the hawthorn hedge puts forth its
        buds.<br>
      And my heart puts forth its pain.<br>
    </p>
    <p><i>1912</i></p>
  </body>
</html>
```

Fig. 2. Poem *Song* displayed in a browser and its HTML source

In contrast, we are interested in generating a more generic stylesheet, which can then be reused to i) transform structurally similar XML documents and ii) transform the document even after it has been updated. In order to generate this solution, the first thing that we need to do is to discover the semantic mappings between the XML and HTML documents. A complete listing of the mappings found is presented in Table 1. The process of discovering these mappings will be explained in Sect. 4.

Table 1. Mappings for poem *Song*

XML	HTML
poem	body
author	div[2]
date	p[4]
title	div[1]
stanza[i]	p[i], for $1 \leq i \leq 3$
line[i]	text()[i] ++ br, for $1 \leq i \leq 4$

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
<xsl:template match="poem">
  <html>
    <head>
      <title>Song</title>
    </head>
    <body>
      <div align="center"><h1>Song</h1></div>
      <div align="center"><h2>Rupert Brooke</h2></div>
      <p>
        And suddenly the wind comes soft.<br/>
        And Spring is here again;<br/>
        And the hawthorn quickens with buds of green<br/>
        And my heart with buds of pain.<br/>
      </p>
      <p>
        My heart all Winter lay so numb.<br/>
        The earth so dead and frore.<br/>
        That I never thought the Spring would come again<br/>
        Or my heart wake any more.<br/>
      </p>
      <p>
        But Winter's broken and earth has woken.<br/>
        And the small birds cry again;<br/>
        And the hawthorn hedge puts forth its buds.<br/>
        And my heart puts forth its pain.<br/>
      </p>
      <p><i>1912</i></p>
    </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

Fig. 3. Naive solution of XSLT stylesheet for poem *Song*

The next phase is to generate an XSLT stylesheet based on the mappings found. The stylesheet should start with the standard header. Following that, a template rule for each XML element specified in Table 1 is created. Finally, we finish off the stylesheet by closing the `<xsl:stylesheet>` element. This non-naive XSLT stylesheet is shown in Fig. 4. This stylesheet can then be reused to transform other XML documents having similar structure as the one in Fig. 1.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
<xsl:template match="poem">
  <html>
    <head>
      <title><xsl:value-of select="title"/></title>
    </head>
    <body>
      <xsl:apply-templates select="title"/>
      <xsl:apply-templates select="author"/>
      <xsl:apply-templates select="stanza"/>
      <xsl:apply-templates select="date"/>
    </body>
  </html>
</xsl:template>
<xsl:template match="title">
  <div align="center"><h1><xsl:value-of select="."/></h1></div>
</xsl:template>
<xsl:template match="author">
  <div align="center"><h2><xsl:value-of select="."/></h2></div>
</xsl:template>
<xsl:template match="date">
  <p><i><xsl:value-of select="."/></i></p>
</xsl:template>
<xsl:template match="stanza">
  <p><xsl:apply-templates select="line"/></p>
</xsl:template>
<xsl:template match="line">
  <xsl:value-of select="."/><br/>
</xsl:template>
</xsl:stylesheet>
```

Fig. 4. Non-naive solution of XSLT stylesheet for poem *Song*

4 XSLTGen System

We now present an overview of the XSLTGen system. We then explain the details of each subsystem in the later subsections. We will use a running example based on a Soccer document.

4.1 System Architecture

The architecture of the XSLTGen system is illustrated in Fig. 5. Two input documents, a source XML document and a desired target HTML document, are given to XSLTGen in order to initiate the stylesheet generation process. The output of XSLTGen is an XSLT stylesheet consisting of rules for transforming the given XML document to the supplied HTML document. As shown in the figure, the system consists of six main components: *DOM Builder*, *Text Matching* subsystem, *Structure Matching* subsystem, *Sequence Checker*, *XSLT Stylesheet Constructor*, and *XSLT Stylesheet Refiner* subsystem.

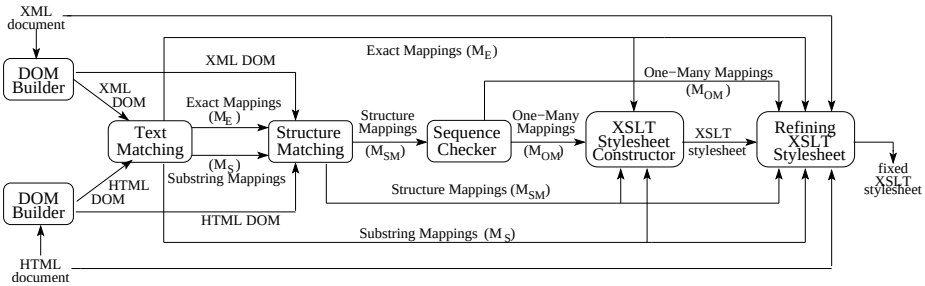


Fig. 5. Architecture of the XSLTGen system

DOM Builder. DOM is "a programming API for documents" [12]. It is based on an object structure that closely resembles the structure of the documents it models. In the DOM, documents have a logical structure which is very much like a tree. For each input document used in XSLTGen, the DOM builder constructs the DOM tree, which represents the structure of the specified document. We adopt the DOM package from W3C (`org.w3c.dom.*`) embedded with in Java to accommodate this task. The input document given to DOM builder has to be strictly well-formed. Figure 6 shows an example of fragments of XML and HTML DOMs for our Soccer example.

Text Matching subsystem. The XML and HTML DOMs built by the DOM builder are input to the text matching subsystem. This subsystem discovers associations between nodes in the XML DOM and those in the HTML DOM, i.e. which nodes in the HTML DOM correspond to which nodes in the XML DOM and what transformations need to be applied to those XML nodes. In this subsystem, the mappings found are instances of text-based element-level

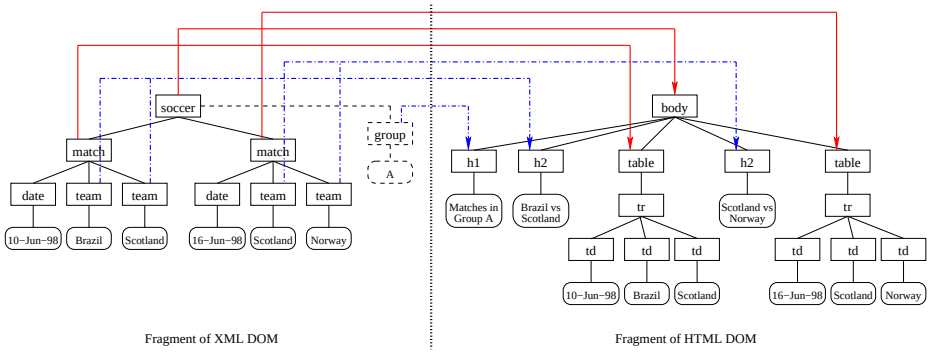


Fig. 6. Soccer example

matching, which can either be exact or substring mappings. Since the HTML document is generated based on the content of the XML document, it is possible to find elements within the HTML DOM that have the same text data or part of text data as the elements in the XML DOM.

Structure Matching subsystem. The structure matching subsystem also takes the XML and HTML DOM trees produced by the DOM builder as its input. Besides these DOM trees, this subsystem takes the list of exact and substring mappings found by the text matching subsystem into account, to discover structure-level associations between XML DOM nodes and HTML DOM nodes. The term structure-level matching refers to matching combinations of elements that appear together in a structure. Loosely speaking, two nodes, an XML and an HTML node, structurally match if their children also match.

Sequence Checker. This subsystem is responsible for verifying each structure mapping found and possibly discovering 1-m mappings (a mapping between an XML element and a concatenation of two or more HTML elements) depending on the result of the verification. In the verification process, the sequence checker checks whether the *sequence* of the XML component conforms to the *sequence* of the HTML component, by comparing an inferred DTD of the XML component and an inferred DTD of the HTML component. The *XTRACT* system [14] is used to infer the DTD of an element. If the verification process fails, the sequence checker generates a new 1-m mapping based on the two extracted DTDs.

XSLT Stylesheet Constructor. This subsystem generates a template rule for each mapping discovered by the text matching, structure matching and sequence checker subsystems. Consecutively, the template rules are put together to construct an XSLT stylesheet.

XSLT Stylesheet Refiner. This subsystem is responsible for repairing the XSLT stylesheet generated by the XSLT stylesheet constructor, in circumstances where there are differences between the original HTML document and the one produced by applying the generated stylesheet back to the given XML document.

4.2 Text Matching

The quality of the XSLT stylesheet generated by the XSLT stylesheet constructor is dependent on the set of mappings input to it. The mappings found in text matching serve as the basis for discovering structure mappings, which in turn serve as the starting point for finding 1-m mappings. Therefore, it is crucial that the text mappings found be complete and accurate. The goal of the text matching subsystem is to achieve this objective by discovering a set of *exact mappings* M_e and *substring mappings* M_s (recall these were defined in section 2.2). As the content of the HTML document is created based on the content of the XML document, it is important to find both exact and substring mappings between the two documents, since there must be HTML elements that have the same string or substring as the XML elements.

The starting point to find a mapping is to compare nodes that have a *value* attribute, i.e. TEXT_NODEs and ATTRIBUTE_NODEs. In this paper, the nodes that we compare are an XML TEXT_NODE with an HTML TEXT_NODE and an XML ATTRIBUTE_NODE with an HTML TEXT_NODE. We do not consider HTML ATTRIBUTE_NODEs, since the attribute value of those nodes is usually specific to the display of the HTML document in the Web browsers and is not generated from the text within the XML document. Example 1 shows examples of both exact and substring mappings.

Example 1. In the Soccer example of Fig. 6,

1. An exact mapping occurs between XML TEXT_NODE "`10-Jun-98`" and HTML TEXT_NODE "`10-Jun-98`", because $X.value \equiv H.value$.
2. A substring mapping occurs between XML TEXT_NODE "`A`" and HTML TEXT_NODE "`Matches in Group A`", since $X.value$ is a substring of $H.value$.

As mentioned earlier, the text matching procedure takes two inputs: an XML DOM x and an HTML DOM h. It discovers as many text mappings as possible between the nodes in x and h. The text matching procedure is called twice, once to first discover all EXACT mappings between the XML nodes in x and the HTML nodes in h and again to discover all SUBSTRING mappings. The output of our text matching algorithm is two lists of mappings: EXACT mappings (M_E) and SUBSTRING mappings (M_S).

Our text matching is implemented using a top-down approach that visits each node in the XML DOM in pre-order and uses the same traversal to explore the HTML DOM, to find a matching node. Note that in order to create an XSLT template rule, the XML node should be an ELEMENT_NODE (not a TEXT_NODE). The discovered exact or substring mapping between two TEXT_NODES cannot represent any transformation in the XSLT template rule. Therefore, we need to determine for each mapping found, the *precise node* of its XML and HTML component that best describes the transformation. This approach allows us to discover more precise mappings between the XML and HTML documents.

Moreover, we set a constraint that every HTML node that has been matched to an XML node during the exact matching process, cannot be considered as a

matching candidate in the substring matching process. In this way, we reduce the number of possible matching elements for substring matching.

Referring back to the Soccer example (Fig. 6), some of the text mappings discovered are: (`<team>Brazil</team>`, `<td>Brazil</td>`) (exact mapping), (`<team>Brazil</team>`, `<h2>Brazil vs Scotland</h2>`) (substring mapping); and
(`group="A"`, `<h1>Matches in Group A</h1>`) (substring mapping).

4.3 Structure Matching

Structure matching discovers all structure mappings between the elements in the XML and HTML DOMs. Finding these mappings is more complicated than finding text matches. We adopt two constraints used in the GLUE system [10] as a guide to determine whether two nodes should be structurally matched:

1. *Neighbourhood Constraint*: "two nodes match if nodes in their neighbourhood also match", where the neighbourhood is defined to be the children.
2. *Union Constraint*: "if every child of node A matches node B, then node A also matches node B" (this constraint is derived from the taxonomy context. It relies on the property that element/concept A is the union of all its children).

Note that there could be a range of possible matching cases, depending on the completeness and precision of the match. In the ideal case, all components of the structures in the two nodes fully match. Alternatively, only some of the components are matched (a partial structural match). Examples of the two cases are shown in Table 2. In the case of partial structure matching between XML node X and HTML node H, there are some children of X that do not match with any children of H; and/or vice versa. However, these XML nodes must be *extra node*s, i.e. they do not have any match in the entire HTML document. Similarly with those children of H that do not match with any children of X, they should not have any match in the entire XML document either. In addition to this, partial structure matching is valid if at least one of the children of X and H matches. If all children of X are extra nodes, and/or all children of H are extra nodes, then this is not a partial structure matching. We allow XSLTGen system to detect partial structure matching because *extra node*s do not give additional information about the mapping and we want to ignore them during the structure matching process, i.e. treat them as if they are not present in the document. Having or not having *extra node*s in the documents should not affect the mappings found.

In order to be able to discover full and partial structure matching, the above constraints need to be modified to construct the definition of structure matching which accommodates both full and partial structure matching.

Definition 1. *A* **structure mapping** $m : (X, H)$ *exists in the following circumstances:*

Table 2. Example of full vs partial structural match

XML elements	HTML elements	
<book> <author>Michael Kay</author> <title>XSLT</title>	<tr> <td>Michael Kay</td> <td>XSLT</td> <td>Wrox</td> <td>34.99</td> </tr>	full structural match of book and tr
<price>34.99</price> <publisher>Wrox</publisher> </book>	<tr> <td>reference</td> <td>Michael Kay</td> <td>XSLT</td> <td>34.99</td> </tr>	partial structural match of book and tr

1. Neighbourhood Constraint*: "X structurally matches H if H is not an* extra HTML node *and every* non-extra *child of H either text matches or structurally matches a* non-extra *descendant of X"; or,*
2. Union Constraint*: "X structurally matches H if every* non-extra *child of H either text matches or structurally matches X".*

As stated in the above definition, we need to examine the children of the two nodes being compared in order to determine if a structure matching exists. Therefore, structure matching is implemented using a bottom-up approach that visits each node in the HTML DOM in post-order and searches for a matching node in the entire XML DOM. This bottom-up approach also enables the system to apply the union constraint to a lower level structure mapping by invalidating and updating it with an upper level structure mapping, if the latter is found to be a better match than the former. If the list of substring mappings M_{SM} is still empty after the structure matching process finishes, we add a mapping from the XML root element to the HTML body element, if it exists, or to the HTML root element, otherwise. The structure matching algorithm is presented in Fig. 7. It takes as input the exact mappings (M_E, the substring mappings (M_S), the XML DOM x and the HTML DOM h. It then returns a list of structure mappings M_{SM}.

In the Soccer example (Fig. 6), some discovered structure mappings are:

1. (`match`, `tr`) (neighbourhood constraint): since every child of `tr` text matches one of the children of `match`, i.e. (`date`, `td[1]`), (`team[1]`, `td[2]`), and (`team[2]`, `td[3]`)
2. mapping (1) will then be invalidated and updated by (`match`, `table`) (union constraint) since the only child of `table`, i.e. `tr`, structurally matches `match`
3. (`soccer`, `body`) (neighbourhood constraint): since every child of `body` either text matches or structurally matches one of the descendant of `soccer`.

```
procedure StructureMatching(M_E, M_S, x, h)
begin
1.   DoStructMatch(M_E, M_S, x, h)
2.   if (M_SM is empty)
3.       if (h has a descendant whose name is body)
4.           add (x, body) to M_SM
5.       else
6.           add (x, h) to M_SM
end

procedure DoStructMatch(M_E, M_S, x, h)
begin
1.   for each child c of h
2.       DoStructMatch(M_E, M_S, x, c)
3.   if h.type = element_node
4.       SearchStructMatch(M_E, M_S, x, h)
end

procedure SearchStructMatch(M_E, M_S, x, h)
begin
1.   E_x := {c_x : c_x ∈ children of x, c_x.type = element_node}
2.   for each node c_x in E_x
3.       SearchStructMatch(M_E, M_S, c_x, h)
4.   if (CheckMatch(M_E, M_S, x, h) = true)
5.       m := (x, h)
6.       if (MappingExist(m) = not_exist)
7.           add m to M_SM
8.       else if (MappingExist(m) = modify)
9.           if (x ≠ x_root)    /* x_root is the root of the XML document */
10.              replace all a.h in {a : a ∈ M_SM, a.x = m.x, a.h.name ≠ body, a.h is a descendant
                 of m.h} with h
end

procedure CheckMatch(M_E, M_S, x, h)
begin
1.   if (h is an extra HTML node)
2.       return false
3.   for each child c_h of h
4.       if (c_h is not an extra HTML node)
5.           let x' be an XML node, such that ((x', c_h) ∈ M_E or (x', c_h) ∈ M_S or (x', c_h) ∈ M_SM)
6.           if (x' ≠ null)
7.               if (x' is not a descendant of x)
8.                   return false
9.           else
10.              return false
11.  return true
end

procedure MappingExist(m)
begin
1.   if (m ∈ M_SM or {a : a ∈ M_SM, a.x is a descendant of m.x, a.h = m.h} is not empty)
2.       return exist
3.   else if ({a : a ∈ M_SM, a.x = m.x, a.h.name ≠ body, a.h is a descendant of m.h} is not empty)
4.       return modify
5.   return not_exist
end
```

Fig. 7. The structure matching algorithm

4.4 Sequence Checking

Up to this point, the mappings generated by the text matching and structure matching subsystems have been limited to 1-1 mappings. In cases where the XML and HTML documents have more complex structure, these mappings may not be accurate and this can affect the quality of the XSLT rules generated from these mappings. Consider the following example:

In Fig. 6, we can see that the sequence of the children of XML node `soccer` is made up of nodes with the same name, `match`; whereas the sequence of the children of the matching HTML node `body` follows a specific pattern: it starts with `h1` and is followed repetitively by `h2` and `table`. Using only the discovered 1-1 mappings, it is not possible to create an XSLT rule for `soccer` that resembles this pattern, since the XML node `match` maps only to the HTML node `table` according to structure matching. In other words, there will be no template that will generate the HTML node `h2`.

Focusing on the structure mapping (`match`, `table`) and the substring mappings {(`team[1]`, `h2`), (`team[2]`, `h2`)}, we can see in the DOM trees that the children of `match`, i.e. `team[1]` and `team[2]`, are not mapped to the descendant of `table`. Instead, they map to the sibling of `table`, i.e. `h2`. Normally, we expect that the descendant of `match` maps only to the descendant of `table`, so that the notion of 1-1 mapping is kept. In this case, there is an intuition that `match` should not only map to `table`, but also to `h2`. In fact, `match` should map to the concatenation of `h2` and `table`, so that the sequence of the children of `body` is preserved when generating the XSLT rule. This is termed as a 1-m mapping, where an XML node maps to the concatenation of two or more HTML nodes.

The 1-m mapping (`match`, `h2 ++ table`) can be found by examining the sub-element sequence of `soccer` and the sub-element sequence of `body` described above. Note that the sub-element sequence of a node can be represented using a regular expression[3]. In this case, the regular expression representing the sub-element sequence of `soccer` is $match*$, whereas the one representing the sub-element sequence of `body` is $h_1(h_2, table)*$. We then check whether the elements in the first sequence conform to the elements in the second sequence, as follows: According to the substring mapping (`group`, `h1`), element h_1 conforms to an attribute of `soccer` and thus, we ignore it and remove h_1 from the second sequence. Comparing $match*$ with $(h_2, table)*$, we can see that element m should conform to elements $(h_2, table)$ since the sequence $match*$ corresponds directly to the sequence $(h_2, table)*$, i.e. they both are in repetitive pattern, denoted by $*$. However, element $match$ conforms only to element $table$, as indicated by the structure matching (`match`, `table`). The verification therefore fails, which indicates that the structure matching (`match`, `table`) is not accurate. Consequently, based on the sequences $match*$ and $(h_2, table)*$, we deduce the more accurate 1-m mapping: (`match`, `h2 ++ table`).

[3] A regular expression is a combination of symbols representing each sub-element and metacharacters: $|, *, +, ?, (,)$. $|$ denotes OR, $*$ denotes zero or more, $+$ denotes one or more, ? denotes zero or one, and ( ) are used for grouping

The main objective of the sequence checking subsystem is to discover 1-m mappings using the technique of comparing two sequences described above. The pseudo-code of the sequence checking procedure is presented in Fig. 8. The task begins with processing the structure mappings in M_{SM} which are provided as input to the procedure. Given a structure mapping m, the issue is to verify that the $seq(m.x)$ conforms to $seq(m.h)$. If $seq(m.x)$ does not conform to $seq(m.h)$, the verification process fails and this situation suggests that there are 1-m mappings that can instead be generated based on these sequences. These discovered mappings 1-m mappings (M_{OM}) are what is finally returned by the procedure.

As we mentioned above, a sequence can be represented using a regular expression. To obtain this regular expression, we adapt the technique for inferring a DTD of an element used by XTRACT [14].

```
procedure CHECKSEQUENCE(M_SM)
begin
1.   check_root = FALSE
2.   for each mapping m in M_SM
3.       if (m.x = x_root and check_root = FALSE)
4.           x_seq = XTRACT(m.x)
5.           h_seq = XTRACT(m.h)
6.       else if (m.x ≠ x_root)
7.           x_dtd = XTRACT(parent of m.x)
8.           x_seq = TRIM_{m.x}(x_dtd)
9.           h_dtd = XTRACT(parent of m.h)
10.          h_seq = TRIM_{m.h}(h_dtd)
11.          if (parent of m.x = x_root)
12.              check_root = TRUE
13.      else
14.          continue
15.      if (x_seq = (s_x)* and h_seq = (x_h)*)
16.          x_seq = s_x
17.          h_seq = s_h
18.      X_SEQS = ∅
19.      if (x_seq = s_1|s_2|...|s_n)
20.          add each s_i to X_SEQS
21.      else
22.          add x_seq to X_SEQS
23.      h_seq = remove symbols representing special HTML elements in h_seq
24.      for each sequence x_i in X_SEQS
25.          if (number of symbols in x_i = 1 and number of symbols in h_seq > 1)
26.              x = XML node represented by symbol x_i
27.              for each symbol h_j in h_seq
28.                  h = h ++ HTML node represented by symbol h_j
29.              m = (x, h)
30.              add m to M_OM
end
```

Fig. 8. The sequence checking algorithm

4.5 XSLT Stylesheet Generation

This subsystem is responsible for constructing a template rule for each of the mappings previously discovered (the exact mappings M_E, structure mappings M_{SM}, and 1-m mappings M_{OM}) and then putting them all together to compose

an XSLT stylesheet. We do not create template rules for the substring mappings in M_S, because in substring mappings, it is possible to have an HTML node whose text *value* is a concatenation of text *value*s of two or more XML nodes. This makes it impossible to create template rules for those XML nodes. Moreover, as the term *substring* implies, there can be some *extra string*s presented in the HTML text *value*. Considering these situations, we implement a procedure that generates a *template* for each distinct HTML node in M_S.

The XSLT stylesheet generation process begins by generating the list of substring rules. We then construct a stylesheet by creating the `<xsl:stylesheet>` root element and subsequently filling it with template rules for the 1-m mappings in M_{OM}, the structure mappings in M_{SM} and the exact mappings in M_E. The template rules for the 1-m mappings have to be constructed first, since within that process, they may invalidate several mappings in M_{SM} and M_E (e.g. rule A and B of Table 6 and thus, the template rules for those omitted mappings do not get used. In each mapping list (M_{OM}, M_{SM}, and M_E), the template rule is constructed for each *distinct* mapping to avoid having conflicting template rules.

Even though the mappings between the nodes in the XML document and the nodes in the HTML documents have been discovered previously, creating a template rule for a mapping m is not an easy task. This is due to the fact that we have to appropriately locate the suitable XSLT instructions, taking into account the structures or the subtrees of the XML and HTML components of m. Therefore, in the next four subsections, we describe the detail of substring rule generation, followed by explanation on how to construct template rules for exact mapping, structure mapping, and 1-m mapping.

Substring Rule Generation. The substring rule generator creates a template from an XML node or a set of XML nodes to each distinct HTML node presented in the substring mapping list M_S. The result of this subsystem is a list of substring rules SUB_RULES, where each element is a tuple (*html_node*, *rule*).

The set of XML nodes that map to the same HTML node could be large, since within that set, it is possible to have two XML nodes with different names but the same string, or two nodes with the same name and string but different locations in the XML DOM. As an example, consider the substring mappings in Fig. 9. In that figure, both XML nodes `artist` (node **I**) and `owner` (node **B**) have the same text value "`Kylie`" and map to the same HTML node `<td>Kylie's by Kylie (2002)</td>` (node **R**). On the other hand, two XML nodes named `date` (nodes **E** and **H**) have the same text value "`2002`" but can either map to HTML node `<td>Aquarium by Aqua (2002)</td>` (node **P**) or `<td>Kylie's by Kylie (2002)</td>` (node **R**).

This situation may cause some ambiguities in the substring rules generated. Consider the mappings where the HTML component is node **R**, i.e. mappings **1**, **6**, **7**, **8**, **9**. There can be four different combinations of the XML components in these mappings that result in two similar substring rules representing the HTML component. Combinations of nodes **E**, **I**, and **J**; or nodes **H**, **I**, and **J** produce the substring rule:

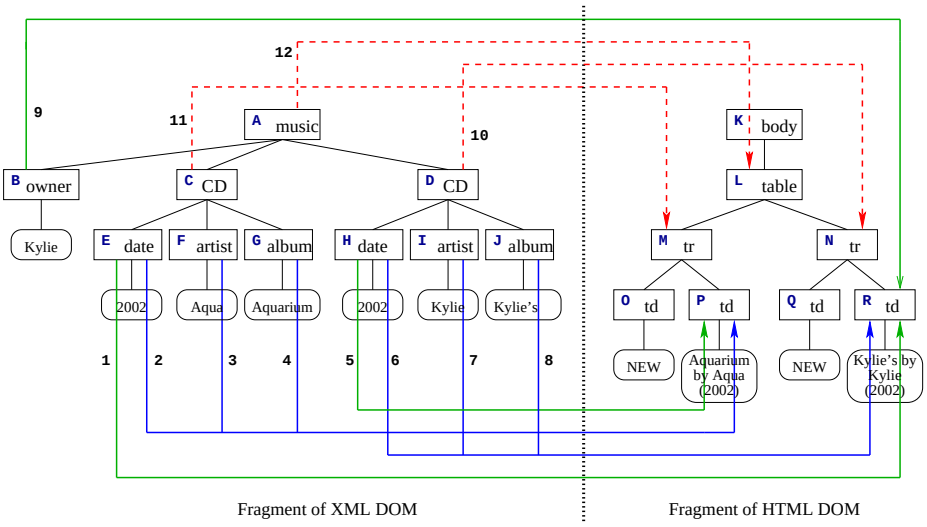


Fig. 9. Music example

```
<tr><xsl:value-of select="album"/> by <xsl:value-of select="artist"/>
    (<xsl:value-of select="date"/>)</tr>.
```

Alternatively, combinations of nodes **B**, **E**, and **J**; or nodes **B**, **H**, and **J** yield the substring rule:

```
<tr><xsl:value-of select="album"/> by <xsl:value-of select="owner"/>
    (<xsl:value-of select="date"/>)</tr>.
```

Clearly, the problem that we face here is choosing the combination of XML nodes that best describe an HTML component. Based on the XML and HTML DOM in Fig. 9, we can see that the combination of XML nodes **H**, **I**, and **J** is the one that best represents the HTML node **R**. This is because the region of these XML nodes is related to the region of HTML node **R**, i.e. the parent of these XML nodes structurally maps to the parent of HTML node **R**, as indicated by the structure mapping **10**. In addition, the mappings **1**, **5**, and **9** should not be considered in the substring rule generation, since the regions between the XML and HTML nodes are not related and hence, they are not accurate.

In the following subsection, we have devised a heuristic strategy for selecting a subset of such HTML component mappings that yields an unambiguous substring rule. We then present a brief description of the algorithm for generating the substring rule itself.

Selecting a Subset of the Mappings to the Same HTML Component The main motivation behind choosing a subset of the mappings that go the same HTML component is that we want to construct a unique and unambiguous substring rule between a set of XML nodes and an HTML node. This purpose is served only if the region of the XML nodes and the region of the HTML node is related. For example, in order to generate a substring rule for HTML node **P** in the

Music example (Fig. 9), there are four substring mappings that can be used: mappings **2**, **3**, **4**, and **5**, since the HTML component of these mappings is the HTML node **P**. However, the subset of these mappings that we select are the mappings **2**, **3**, and **4**, since the XML components of these mappings are located in the region that is related to the region of the HTML node **P**. The two regions that we discuss are the subtree of XML node **C** and the subtree of HTML node **M**, which are structurally mapped as indicated by the structure mapping **11**. Mapping **5** is not selected since the XML node **F** is not located in the subtree of XML node **C**.

In this subsection, we describe how we choose a subset of substring mappings that best describes an HTML node based on the related regions. Note that the set of mappings that we are focusing on, is the one that has the same HTML component for every mapping in that set. This set of mappings is in fact a subset of M_S. Splitting M_S into sets of mappings with the same HTML component is not a complicated task.

Intuitively, given a set of mappings M_H that all map to the same HTML node, a subset G of M_H is a good subset if the combination of the XML components in G produces a single substring rule that represents the HTML component.

Therefore, given a set of substring mappings M_H, which has the same HTML component h for every mapping in it, we first find a mapping s in M_{SM} or in M_{OM}, such that $s.h$, is the closest ancestor of h. Then, a subset G of M_H is a "good" subset for generating a unique substring rule, if for every mapping m in G, $m.x$ is a descendant of $s.x$.

Referring back to the Music example describe in Sect. 4.5, to facilitate the substring rule generation of HTML node **R**, we need to examine the set M_H of substring mappings **1**, **6**, **7**, **8**, and **9**, since these mappings lead to the HTML node **R**. As stated above, the first thing that we need to do is find a structure mapping or a 1-m mapping s, whose HTML component is the closest ancestor of node **R**. In this case, the mapping s is the structure mapping between node **D** and node **N** (mapping **10**). We then select the mappings in M_H whose XML component is a descendant of node **D** and we obtain a good subset G consisting of mappings **6**, **7**, and **8**.

Once we have a "good" subset G, a substring rule R is generated by invoking the algorithm described in the next subsection, and is then added to the list of substring rules SUB_RULES.

Algorithm for Generating a Substring Rule. In this subsection, we show how the substring rule for a set of mappings G can be derived. Recall that all mappings in G have the same HTML component, h. Concisely, the main idea of the substring rule generation is to construct the string of h by combining the strings of the XML components of the mappings in G. Since the mappings in G are found within the text matching subsystem, h only has one string, denoted by *h_string*, which is the *value* of the TEXT_NODE located at the leaf of the subtree of h. For example, the *h_string* of the HTML node `td` (node **R**) in the Music example (Fig. 9) is "`Kylie's by Kylie (2002)`". The same condition applies to the XML component of each mapping in G. If the XML component is an

ELEMENT_NODE, the XML node also has only one string x_string, which is the *value* of the TEXT_NODE located at the leaf of the subtree of the XML node. For instance, the x_string of the XML node `date` (node **E**) in Fig. 9 is "`2002`". In the case where the XML component is an ATTRIBUTE_NODE, the string x_string is simply the *value* of the XML node.

As the term *substring* implies, there must be parts of h_string that do not have any matching x_string in G, i.e. these are *extra string*s. For each substring of h_string that exactly matches an x_string, an XSLT `<xsl:value-of>` statement is generated to extract the x_string from the corresponding XML component. Thus, we can conclude that the generated substring rule is a sequence of *extra string*s and `<xsl:value-of>` statements that together represents h_string.

The substring rule s_rule for the mappings in G is constructed recursively until h_string is empty. At each step, we search for a mapping m' whose x_string of $m'.x$ matches the longest prefix of h_string and then update the substring rule s_rule using the rules specified in Table 3. When h_string is empty, the construction of s_rule of G is finished and a tuple (h, s_rule) is returned.

Table 3. Rules for generating a substring rule

Rule	Condition	Update
A	m' is not found	Extract the first character of h_string and add it to s_rule, since it is suspected that the prefix of the current h_string is part of the *extra strings*
B	m' is found, $m'.x$ is an ELEMENT_NODE	Add `<xsl:value-of select="`$m'.x.name$`"/>` to s_rule and delete the prefix of h_string matching x_string
C	m' is found, $m'.x$ is an ATTRIBUTE_NODE	Add `<xsl:value-of select="`$owner(m'.x).name$`/@`$m'.x.name$`"/>` and delete the prefix of h_string matching x_string

We believe that this algorithm is a reasonable method for constructing substring rules, since it is capable of capturing *extra string*s and matching x_strings accurately. The following example best illustrates the key steps of our algorithm.

Example 2. Consider the set G of substring mappings **6**, **7**, and **8** shown in the Music example of Fig. 9. In this case, the h_string of G is "`Kylie's by Kylie (2002)`", while the set of x_strings is {"`2002`", "`Kylie`", `Kylie's`}. Below, we list how Rules (A) and (B) described in Table 3 are recursively applied to derive the substring rule s_rule.

1. Mapping **8** is chosen since its x_string "`Kylie's`" is the longest string that matches the prefix of h_string "`Kylie's by Kylie (2002)`". According to **Rule (B)**,
   $s\_rule =$ `<xsl:value-of select="album"/>`
   $h\_string =$ " `by Kylie (2002)`".

2. Apply **Rule (A)** recursively for characters in substring " `by` " since it is an *extra string*. At the final execution of **Rule (A)**,
   s_rule = `<xsl:value-of select="album"/> by`
   h_string = "`Kylie (2002)`".
3. Mapping **7** is chosen and we apply **Rule (B)**.
4. Apply **Rule (A)** recursively for characters in *extra string* " `(`".
5. Mapping **6** is chosen and we apply **Rule (B)**.
6. Apply **Rule (A)** for character ")". h_string is now empty and s_rule is
   `<xsl:value-of select="album"/> by <xsl:value-of select="artist"/>`
   `(<xsl:value-of select="date"/>)`

Constructing a Template Rule for an Exact Mapping in M_E. As described in Sect. 2.1, each template rule begins with an XSLT `<xsl:template>` element and ends by closing that element. For a mapping m, the pattern of the corresponding template rule is the name of $m.x$. The difficult task lies in determining the appropriate template, i.e. which XSLT instructions to be used and where they should be placed.

We now describe how we construct a template rule for an exact mapping m. Compared to the procedures for the other two mappings (structure mapping and 1-m mapping), this procedure is the simplest and the most straightforward, since there is only one XSLT instruction used in the template: `<xsl:value-of>`. In this procedure, we only construct a template rule when m is a mapping between an XML ELEMENT_NODE to an HTML node or a concatenation of HTML nodes. The reason that we ignore mappings involving XML ATTRIBUTE_NODEs is that the template for this mapping will be generated directly within the construction of the template rule for structure mapping and 1-m mapping.

Since there could be mappings from an XML node to a concatenation of HTML nodes in text matching, we need to create a template for each HTML node h_i in $m.h$. Given an exact mapping m, the rules for creating a template representing transformation between $m.x$ and h_i are listed in Table 4.

Table 4. Rules for creating the template for exact mappings

Rule	Condition	Template
A	h_i is a *special* HTML element	`<`$h_i.name$`/>`
B	h_i is not a *special* HTML element, $m.x$ is an ELEMENT_NODE	`<`$h_i.name$`><xsl:value-of select="."/>` `</`$h_i.name$`>`

C	h_i is not a *special* HTML element, $m.x$ is an ATTRIBUTE_NODE	`<`$h_i.name$`>`
		`<xsl:value-of select="@`$m.x.name$`."/>`
		`</`$h_i.name$`>`

Example 3. Consider the exact mapping (`line`, `text() ++ br`). The steps performed to construct the template rule are:

1. Create a template rule: `<xsl:template match="line">`
2. Create a template for each HTML node in (`text() ++ br`): for HTML node `text()`, apply **Rule (B)** since the XML node `line` is an ELEMENT_NODE. The template body is: `<xsl:value-of select="."/>`
   Note that there are no opening and closing tags, since `text()` is a TEXT_NODE.
3. Apply **Rule (A)**, since `br` is a *special* HTML element: `<br/>`
4. Close the template rule: `</xsl:template>`

Constructing a Template Rule for a Structure Mapping in M_{SM}. We next explain how the template rule for a structure mapping m is constructed. Recall from the structure matching subsystem that one of the mappings in M_{SM} must be the mapping that has the root of the XML document as its XML component. Let r denote this special mapping. For the mapping r, the template begins with copying the root of the HTML document and its subtree, excluding the HTML component $r.h$ and its subtree.

The next step in constructing the template for mapping r follows the steps performed for the other mappings in M_{SM}. For any mapping m in M_{SM}, the opening tag for $m.h$ is created. The process continues with creating a template for each child c_i of $m.h$ and finishes by closing the $m.h$ tag.

For each HTML child c_i, we need to determine whether an XSLT instruction is needed; and if so, which XSLT instruction should be used. This task depends fully on the similarities and differences between the structure of the XML component $m.x$ and the structure of the HTML node h_i. Given an XML component $m.x$ and a child c_i of the HTML component $m.h$, the rules for creating a template that represents the transformation from $m.x$ to c_i are listed in
Table 5.

Example 4. Consider the structure mapping (`match`, `table`) and the exact mappings (`date`, `td[1]`), (`team[1]`, `td[2]`), (`team[2]`, `td[3]`) discovered in the Soccer example (Fig. 6) . The steps required to generate the template rule for this structure mapping are:

1. Create a template rule: `<xsl:template match="match">`
2. Start the template body by generating an opening tag for HTML node `table`, since `match` is not the root of the XML document: `<table>`
3. Apply **Rule (G)** since `tr` has no matching XML node in any mapping lists:
   (a) Generate an opening tag for HTML node `tr`: `<tr>`
   (b) For the first child of `tr`, i.e. `td[1]`, apply **Rule (B)** because (`date`, `td[1]`) $\in M_E$ and `date` is a descendant of the XML node `match`:
   `<xsl:apply-templates select="./date"/>`
   (c) Apply **Rule (B)** to the second and third child of `tr`, i.e. `td[2]` and `td[3]`, for the same reason as the first child `td[1]`, and we obtain:
   `<xsl:apply-templates select="./team[1]"/>`
   `<xsl:apply-templates select="./team[2]"/>`
   (d) Generate the closing tag for `tr`: `</tr>`
4. Generate the closing tag for HTML node `table`: `</table>`
5. Close the template rule: `</xsl:template>`

Table 5. Rules for creating the template for structure mappings

Rule	Condition	Template
A	c_i is an *extra node* but not a *special* HTML element	Node c_i complete with its subtree
B	$\exists e \in M_E$, $e.x$ is a descendant of $m.x$, $e.h = c_i$	`<xsl:apply-templates select="`{XPath describing $e.x$ in the context of $m.x$}`"/>`
C	$\exists e \in M_E$, $owner(e.x) = m.x$, $e.x$ is an ATTRIBUTE_NODE, $e.h = c_i$	Apply the algorithm for constructing a template rule for an exact mapping to e
D	$\exists e \in M_E$, $owner(e.x)$ is a descendant of $m.x$, $e.x$ is an ATTRIBUTE_NODE, $e.h = c_i$	`<`$c_i.name$`><xsl:value-of select="`{XPath describing $owner(e.x)$ in the context of $m.x$}`/@`$e.x.name$`"/></`$c_i.name$`>`
E	$\exists s \in M_{SM}$, $s.x$ is a descendant of $m.x$, $s.h = c_i$	`<xsl:apply-templates select="`{XPath describing $s.x$ in the context of $m.x$}`"/>`
F	$\exists sr : (c_i, sr.rule) \in$ SUB_RULES	`<`$c_i.name$`>` $sr.rule$ `</`$c_i.name$`>`
G	Otherwise	`<`$c_i.name$`>` {Create template for $(m.x, c_i)$, i.e. apply **Rules (A) - (G)** to each child of c_i} `</`$c_i.name$`>`

Constructing a Template Rule for a One-Many Mapping in M_{OM}. This subsection describes the process for constructing a template rule for a 1-m mapping m in M_{OM}. Being a 1-m mapping, the HTML component $m.h$ must be a concatenation of several HTML nodes. Thus, each HTML node h_i in $m.h$ is processed sequentially to find out the sequence of XSLT instructions that fill up the template rule. Given a 1-m mapping m, the rule for creating a template representing the transformation from $m.x$ to each h_i is presented in Table 6.

Example 5. Consider 1-m mapping (`match`, `h2 ++ table`) discovered in Sect. 4.4, and structure mapping (`match`, `table`) found in Sect. 4.3 for the Soccer example (Fig. 6). Suppose we have generated substring rule (`h2`, `<xsl:value-of select= "team[1]"/> vs <xsl:value-of select="team[2]"/>`). The steps involved in generating the template rule for the 1-m mapping is:

1. Create a template rule: `<xsl:template match="match">`
2. Apply **Rule (F)** since SUB_RULES contains a substring rule for `h2`:
```
<h2><xsl:value-of select="team[1]"/> vs
    <xsl:value-of select="team[2]"/></h2>
```
3. For HTML node `table`, apply **Rule (A)** since (`match`, `table`) $\in M_{SM}$. This is the same as the template body discovered in Example 4, which is:
```
<table><tr>
  <xsl:apply-templates select="./date"/>
  <xsl:apply-templates select="./team[1]"/>
  <xsl:apply-templates select="./team[2]"/>
</tr></table>
```
4. Close the template rule: `</xsl:template>`

Table 6. Rules for creating the template for 1-m mappings

Rule	Condition	Template
A	$(m.x, h_i) \in M_{SM}$	Apply the algorithm for constructing a template rule for a structure mapping to $(m.x, h_i)$. Then, delete mappings $(m.x, h_i) \in M_{SM}$
B	$(m.x, h_i) \in M_E$	Apply the algorithm for constructing a template rule for an exact mapping to $(m.x, h_i)$. Then, delete mappings $(m.x, h_i) \in M_E$
C	$\exists e \in M_E$, $e.x$ is a descendant of $m.x$, $e.h = h_i$	<$h_i.name$><xsl:apply-templates select="{XPath describing $e.x$ in the context of $m.x$}"/></$h_i.name$>
D	$\exists e \in M_E$, $owner(e.x) = m.x$, $e.x$ is an ATTRIBUTE_NODE, $e.h = h_i$	Apply the algorithm for constructing a template rule for an exact mapping to e
E	$\exists e \in M_E$, $owner(e.x)$ is a descendant of $m.x$, $e.x$ is an ATTRIBUTE_NODE, $e.h = h_i$	<$h_i.name$><xsl:value-of select="{XPath describing $owner(e.x)$ in the context of $m.x$}/@$e.x.name$"/></$h_i.name$>
F	$\exists sr : (h_i, sr.rule) \in$ SUB_RULES	<$h_i.name$>$sr.rule$</$h_i.name$>
G	h_i is an *extra node* but not a *special* HTML element	Node h_i complete with its subtree

4.6 Refining the XSLT Stylesheet

In some cases, the (new) HTML document obtained by applying the generated XSLT stylesheet to the XML document may not be accurate, due to the wrong ordering of instructions within a template. i.e. there exist differences between this (new) HTML document and the original (user-defined) HTML document. By examining such differences, we can improve the accuracy of the stylesheets generated by XSLTGen. This step is applicable in circumstances where we have a set of complete and accurate mappings between the XML and HTML documents, but generated erroneous XSLT code based on these mappings. If the discovered mappings themselves are incorrect or incomplete, then this refinement step will not be effective and it is better to address the problem by improving the matching techniques. An indicator that we have complete and accurate mappings is that each element in the new HTML document corresponds exactly to the element in the original HTML document at the same depth.

Refining can be effective in situations where the generated XSLT stylesheet can be fixed by applying local move operations within the template matches, i.e. the generated stylesheet is inaccurate due to the wrong ordering of XSLT instructions within the template rules. This situation typically occurs when we have two or more XML nodes with the same name and are located at the same depth in the XML DOM, but have different order or sequence of children. In this case, the mappings generated are complete and accurate, however our XSLT stylesheet constructor assumes that these XML nodes have the same order of

children and hence, it follows the order of the first node (amongst these XML nodes) encountered in the pre-order traversal of the XML DOM. Therefore, the main objective of this refining step is to fix the order of XSLT instructions within the template matches of the generated stylesheet, so that the resulting HTML document is closer to or exactly the same as the original HTML document.

A naive approach to the above problem is to use brute force and attempt all possible orderings of instructions within templates until the correct one is found (i.e. until there exist no differences between the new and the original HTML documents). However, this approach is prohibitively costly. Therefore, we adopt a heuristic approach, which begins by examining the differences between the original and the new HTML documents. We employ a change-detection algorithm [6], that produces a sequence of edit operations needed to transform the original HTML document to the new HTML document. The types of edit operations returned are insert, delete, change, and move. Of course, other similar change detection algorithms such as [9, 18] could potentially be used instead.

To carry out the refinement, the edit operation that we focus on is the move operations, since we want to swap around the XSLT instructions in a template match to get the correct order. In order for this to work, we require that there are no missing XSLT instructions for any template match in the XSLT stylesheet. After examining all move operations, this procedure is started over using the fixed XSLT stylesheet. This repetition is stopped when no move operations are found in one iteration; or, the number of move operations found in one iteration is greater than those found in the previous iteration. The second condition is required to prevent the possibility of fixing the stylesheet incorrectly. We want the number of move operations to decrease in each iteration until it reaches zero. A sketch of the refinement algorithm is presented in Fig. 10.

```
procedure ImproveXSLT(XML, HTML, XSLT, M_E, M_SM, M_OM)
begin
1.   M_prev = ∅
2.   Hist = ∅    /* List of XSLT instructions that have been moved */
3.   repeat
4.       Apply XSLT to XML to get HTML_new
5.       Find the set of edit operations, E, between HTML and HTML_new
6.       M = move operations in E
7.       if (|M| = 0 or |M| > |M_prev|)
8.           return
9.       M_prev = M
10.      NoFix = 0    /* Number of modifications made to XSLT in the current iteration */
11.      for each move m ∈ M
12.          Find the template match to be fixed, T, in XSLT
13.          Find the specific XSLT instruction to be moved, I, within T
14.          if I ∉ Hist
15.              Move I into the correct place using the information from M
16.              Add I to Hist
17.              Increment NoFix
18.  until
19.      No new modifications made to XSLT, i.e. NoFix = 0
end
```

Fig. 10. The XSLT stylesheet refining algorithm (sketch)

5 Empirical Evaluation

We have conducted experiments to study and measure the performance of the XSLTGen system. Our goals were to evaluate the matching accuracy of XSLTGen, and verify that XSLTGen generates useful XSLT stylesheets with a variety of XML and HTML documents. The system is written in Java, and employs a library for HTML cleaning, *JTidy*[4].

5.1 Data sets

It is difficult to test this system on a class of documents, due to the lack of other suitable automatic generation systems for comparison. However, to give the reader some idea on how XSLTGen performs, we evaluated XSLTGen on four examples taken from a popular XSLT book[5] and a real-life data taken from MSN Messenger[6] chat log.

These datasets were originally pairs of (XML document, XSLT stylesheet). To get the HTML document associated with each dataset, we apply the original XSLT stylesheet to the XML document using Xalan[7] XSLT processor. We choose the datasets such that they exhibit a wide variety of characteristics. This is useful for benchmarking the performance of the XSLTGen system with different types of data. The characteristics of the five datasets are shown in Table 7. The Books dataset has its HTML document in a tabular format and the structure of the XML is fairly similar. The Itinerary dataset contains no extra nodes in its HTML DOM, unlike Books. Also, its HTML DOM is roughly twice the size of its XML DOM. The Poem dataset is different from the previous two, containing a large number of special HTML elements. The Soccer dataset has many more HTML nodes than XML nodes and also a very large number of extra nodes. The Chat Log dataset is noteworthy in that it has a large maximum number of children per node, 20 in each DOM, which is significantly bigger than any of the other datasets.

5.2 Manual Mappings

For evaluation purposes, we manually determined the correct mappings between the XML and HTML DOMs in each dataset. These mappings include exact mappings, substring mappings, structure mappings, and 1-m mappings. Table 8 shows the number of manual mappings found for each dataset.

5.3 Experiments

For each dataset, we applied XSLTGen to find the mappings between elements in the XML and HTML DOMs, and generate an XSLT stylesheet that transforms the XML document to the HTML document. We then measured two aspects:

[4] http://sourceforge.net/projects/jtidy
[5] http://www.wrox.com/books/0764543814.shtml
[6] http://messenger.ninemsn.com/Default.aspx
[7] http://xml.apache.org/xalan-j/index.html

Table 7. Datasets used in our experiments

Datasets x : xml h : html		# elem node	# non-leaf elem node	depth	# text node	# attr node	# extra HTML node	# special HTML element	max # children / node
Books	x	17	5	2	12	4	-	-	3
	h	28	7	4	21	2	5	0	5
Itinerary	x	10	1	1	9	9	-	-	9
	h	21	3	3	18	0	0	0	18
Poem	x	19	4	2	15	0	-	-	6
	h	20	6	3	15	1	1	12	5
Soccer	x	25	7	2	18	13	-	-	6
	h	99	44	5	55	18	54	0	13
Chat Log	x	121	61	3	60	161	-	-	20
	h	244	45	6	105	40	100	0	20

Table 8. Manual mappings determined in the datasets

Datasets	Exact	Substring	Structure	1-m
Books	22	0	5	0
Itinerary	9	9	1	9
Poem	1	12	5	0
Soccer	48	68	7	6
Chat Log	1440	0	51	0

1. *Matching accuracy*: the percentage of the manually determined mappings that XSLTGen discovered.
2. The quality of the XSLT stylesheet inferred by XSLTGen.

To evaluate the quality of the XSLT stylesheet generated by XSLTGen in each dataset, we applied the generated XSLT stylesheet back to the XML document using Xalan and then compared the resulting HTML with the original HTML document using HTMLDiff[8]. HTMLDiff is a tool for analysing changes made between two revisions of the same file. It is commonly used for analysing HTML and XML documents. The differences may be viewed visually in a browser, or analysed at the source level.

5.4 Matching Accuracy

Table 9 shows the matching accuracy on the different datasets for the subsystems of XSLTGen. As shown in the table, XSLTGen achieves high matching accuracy across all five datasets. Exact mappings reach 100% accuracy in four out of five datasets. In the dataset *Chat Log*, exact mappings reach 86% accuracy which is still deemed significantly accurate. This is because there are undiscovered mappings from XML ATTRIBUTE NODEs to HTML ATTRIBUTE NODEs, which violates our assumption in Sect. 4.2 that the value of an HTML ATTRIBUTE_NODE is usually specific to the display of the HTML document in the Web browsers and is not generated from a text within the XML document. Substring mappings achieve 100% accuracy in the datasets *Itinerary* and *Soccer*. In contrast, substring mappings achieve 0% accuracy in the dataset *Poem*. This poor performance is caused by incorrectly classifying the substring mappings as exact mappings during the text matching process. In the datasets *Books* and *Chat Log*, substring mappings do not exist. Structure mappings achieve perfect accuracy in all datasets except *Poem*. In the dataset *Poem*, the structure mappings achieve 80% accuracy because the XML node `author` is incorrectly matched with the HTML TEXT_NODE "`Rupert Brooke`" in text matching, while it should be matched with the HTML node `div` in structure matching. Following the success of the other mappings, 1-m mappings also achieve 100% accuracy in all applicable datasets, i.e. *Itinerary* and *Soccer*. In the datasets *Books*, *Poem* and *Chat Log*, there are no 1-m mappings.

The results indicate that in most of these cases, the XSLTGen system is capable of discovering complete and accurate mappings.

5.5 Quality of generated XSLT stylesheets

Table 10 shows the result of comparing the original and the new HTML document, i.e. the one produced by applying the generated XSLT stylesheet to the XML document, for different datasets in terms of the percentage of correct nodes.

As shown in the table, the new HTML documents have a high percentage of correct nodes. Using HTMLDiff, the results of comparing the new HTML

[8] http://www.componentsoftware.com/products/HTMLDiff/

Table 9. Matching accuracy of XSLTGen (in %)

Datasets	Exact	Substring	Structure	1-m
Books	100.00		100.00	
Itinerary	100.00	100.00	100.00	100.00
Poem	100.00	0.00	80.00	
Soccer	100.00	100.00	100.00	100.00
Chat Log	86.11		100.00	

Table 10. Percentage of correct nodes in the new HTML document for each dataset

Datasets	Element Nodes	Text Nodes	Attributes Nodes
Books	100.00	85.71	100.00
Itinerary	100.00	100.00	100.00
Poem	100.00	14.29	80.00
Soccer	100.00	100.00	100.00
Chat Log	100.00	100.00	75.00

document with the original HTML document in each dataset is reflected in the table. The accuracy is very high. In the datasets *Itinerary* and *Soccer*, the HTML documents being compared are identical. This is shown by the achievement of 100% in all types of nodes. In the dataset *Poem*, the two HTML documents have exactly the same appearance in Web browsers, but according to HTMLDiff, there are some missing whitespaces in each line within the paragraphs of the new HTML document. That is the reason why the percentage of correct TEXT_NODEs in the *Poem* dataset is very low (14%). The only possible explanation in this case is that in the text matching subsystem, we remove the leading and trailing whitespaces of a string before the matching is done. The improvement stage also does not fix the stylesheet since there are no move operations. In the dataset *Books*, the difference occurs in the first column of the table. In the original HTML document, the first column is a sequence of numbers 1, 2, 3, and 4; whereas in the new HTML document, the first column is a sequence of 1s. This is because the numbers 1, 2, 3, and 4 in the original HTML document are represented using four *extra node*s, and our template rule constructor in the XSLT stylesheet generator assumes that all *extra node*s that are cousins (their parent are siblings and have the same node name) have the same structure and values. Since in this dataset the four *extra node*s have different text values, the percentage of correct TEXT_NODEs in the new HTML document is slightly affected (86%). Lastly, the differences between the original and the new HTML documents in the dataset *Chat Log* are caused by the undiscovered mappings mentioned in the previous subsection. Because of these undiscovered mappings, it is not possible to fix the XSLT stylesheet in the improvement stage. These undiscovered mappings affect some ATTRIBUTE_NODEs in the new HTML document but the percentage of correct ATTRIBUTE_NODEs is still acceptable (75%).

5.6 Discussion and Future Work

We have also tested XSLTGen on many other examples. In general, it seems to perform most effectively in situations where the XSLT stylesheet that needs to be generated follows a 'fill-in-the-blanks' style design pattern [16]. In such situations, the structure of the required stylesheet is rather similar to the desired output, with variable data being retrieved from the XML and inserted at particular points. The stylesheets generated were also tested on extra structurally similar versions of the given examples, that were generated by insertions and deletions of subtrees. The precision was similar to that for the original examples.

However, there are some problems that prevent XSLTGen from obtaining higher matching accuracy. First, in a few cases, XSLTGen is not able to discover some mappings which deal with relationships between XML ATTRIBUTE_NODEs and HTML ATTRIBUTE_NODEs. The reason is that these mappings violate our assumption stated in Sect. 4.2. This problem can be alleviated by adding HTML ATTRIBUTE_NODEs in the matching process. Undiscovered mappings are also caused by incorrectly matching some nodes, which is the second problem faced in the matching process. Incorrect matchings typically occur when an XML or an HTML TEXT_NODE has some ELEMENT_NODE siblings. In some cases, these nodes should be matched during the text matching process, while in other cases they should be matched in structure matching. Here, the challenge will be in developing matching techniques that are able to determine whether a TEXT_NODE should be matched during text matching or structure matching. The third problem concerns with incorrectly classified mappings. This problem only occurs between a substring mapping and an exact mapping, when the compared strings have some leading and trailing whitespaces. Determining whether the whitespaces should be kept or removed is a difficult choice. In many cases, the whitespaces should be removed in order to correctly match some nodes. This is how whitespaces are treated in our text matching. However in some cases, the whitespaces should be kept since they are used to specify the formatting of the HTML document. This problem can be mitigated by adding an option to keep or remove the whitespaces.

Besides this, as the theme of our text matching subsystem is text-based matching (matching two strings), the performance of the matching process decreases if the supplied documents contain mainly numerical data. In this case, the mappings discovered, especially substring mappings, are often inaccurate and conflicting, i.e. more than one HTML nodes match a single XML node. This is also true for cases where very complex restructurings of the data are need to be performed, such as unnesting and normalizing.

The mappings discovered certainly are an important basis for generating a good and accurate XSLT stylesheet. Undiscovered mappings and incorrect matchings cause the generated stylesheet to be erroneous, since the HTML nodes that are supposed to have matching XML nodes are treated as *extra node*s and thus, are directly copied to the corresponding template. Although in some cases the HTML document generated using this kind of stylesheet is identical to the

original HTML document, this behaviour obviously reduces the reusability of the XSLT stylesheet, since it contains information specific to a particular XML document. On the other hand, incorrectly classified mappings do not cause serious problems in the XSLT stylesheet. They may or may not affect the appearance of the HTML document generated using this stylesheet in a browser.

However, having complete and accurate mappings does not guarantee that the generated XSLT stylesheet will be accurate and of high quality. Another critical factor that should be considered is the similarities and differences between the structure of the XML document and the structure of the HTML document.

Occasionally within those complete and accurate mappings, there is more than one mapping discovered for the same XML node. This case only happens when two or more strings in the HTML document are generated from a single string in the XML document. In this case, the generated XSLT stylesheet is inconsistent, since it contains conflicting template rules. Hence, an additional extension to XSLTGen is to make it be aware of such cases, and not generate conflicting template rules, but instead, integrate the template of each of these template rules to the appropriate `<xsl:apply-template>` instruction that calls it, taking into account the structures of the two documents.

We note that the current version of XSLTGen does not support the capability to automatically generate XSLT stylesheets with complex functions (e.g. sorting). This is a very challenging task and an interesting direction for future work. Another direction for future work would be to modify XSLTGen so that it uses a DTD, if it has been provided with the input XML document in the automatic generation process. Of course, if the desired output is HTML, then the standard DTD for the HTML language is unlikely to be useful.

Lastly, we observe that the focus of the XSLTGen has been on producing quality stylesheets, rather than minimising execution time. Improving the runtime efficiency and scalability of the system for very large documents are interesting ways to enhance the system. However, we expect the XSLTGen system to be deployed in a static, rather than dynamic manner (i.e. run once-only for an input-output pair, rather than repeatedly) and so this is not a primary issue.

6 Related Work

Recently, there has been much work in the literature about XML document transformations, in which only a few address the problem of generating XSLT stylesheet automatically. To the best of our knowledge, this work mainly focuses on XML to XML transformations and the techniques involved are specific to the XML to XML transformations, such as element/tag names comparison, which is impossible in our case since XML and HTML have completely different tag names; and the use of XML Schemas. Although it may be possible to generate an XML Schema for an HTML document, it would not be particularly useful in our scenario.

[13] presents a system that captures the semantics of the XML schemas and using these semantics to automatically generate the necessary XSLTs. The sys-

tem firstly defines a rich information model using ontology and then maps the schema's elements, complex types, and simple types to the information model, thereby formally capturing the schemas' semantics. While the creation of information model is partially automated by arbitrarily declaring an ontological concept per schema component; the mapping process requires a human intervention. In the next step, the active semantic hub is used to automatically generate the XSLT based on the element's meanings. The algorithm find elements of the source and target that mapped to the same ontological concepts, or to concepts that can be related to each other with encoded conversion rules.

A semi-automatic XSLT stylesheet generation is also invoked within the IDACT system [23], which is a tool for automating the compilation of heterogeneous scientific datasets. If a suitable transformation is not found in its database, IDACT attempts to create a new transformation. IDACT firstly represents the input and output XML documents as trees, and then determines the relationships between them, by considering the XML element names, or the XML element content formats. By searching a library of XSLT conversions or functions, and a database of element name relationships, IDACT can build an entire XSLT stylesheet from library components. However, if IDACT finds an element that does not have a relationship, the user is prompted to provide one. This new relationship and the new XSLT stylesheet will then be saved and made available for future use.

[11] describes an approach to an automatic XML to XML transformation generator that is based on a theory of information-preserving and approximating XML operations and their associated DTD transformations. The system strictly requires the presence of DTDs of both the source and target documents. The process starts with identifying an algebra of information-preserving and - approxi- mating XML transformations. This is done by defining corresponding relations on XML trees, which are induced by operations on XML values. The operations considered as information-preserving are *renaming of tags*, *regrouping*, and *congruence*; while the one considered as information-approximating is *deletion*. The *renaming of tags* is comparable to our exact mapping. However, its procedure involves some measures of similarity between tag names, which is not applicable to XSLTGen since XML and HTML have completely different set of tags. The *congruence* operation is similar to our structure matching. The next step is to construct a search space of DTDs by applying algebra operations and find a path from a source DTD element to the required target DTD element. The path represents the sequence of operations that realise the sought transformation. Based on the presentation in the paper, the approach seems to be a theoretical one and does not appear to have been implemented in an actual system.

In [3], a conceptual modelling based approach is used for performing semantic matching. It introduces a new layered model for XML schemas, called LIMXS, which offers a semantic view for XML schemas through the specification of concepts and semantic relationships among them. This model initiates a dynamic and incremental algorithm for finding semantic mappings. The system initially

performs semantic matching between a source and target semantic views. Once semantic mapping is generated and validated by user, the result is given to the logical layer, which performs logical matching. Finally, an XSLT code is automatically produced. In a way, XSLTGen also uses the dynamic and incremental approach with our text matching, structure matching, and sequence checking, but without the need for user interaction.

The only prior work about XML to HTML transformations of which we are aware of is XSLbyDemo [24], a system that generates an XSLT stylesheet by example. The process begins with transforming the XML document to an HTML page using an XSLT stylesheet that was manually created taking into account the DTD of the XML document. This HTML page is referred to as *initial HTML page*. The user then modifies the initial HTML page using a WYSIWYG editor and their actions are recorded in an operation history. Based on the user's operation history, an XSLT stylesheet is generated. Obviously, this system is not automatic, since the user directly involves at some stages of the XSLT generation process. Hence, it is not comparable to our fully automatic XSLTGen system. Specifically, our approach differs from XSLbyDemo in three key ways:

1. Our algorithm produces an XSLT stylesheet consisting of transformations from an XML document to an HTML document, while XSLbyDemo generates transformations from an initial HTML to its modified HTML document.
2. Following the first argument, our generated XSLT can be applied directly to other XML documents from the same document class, whereas using XSLbyDemo, the other XML documents have to be converted to their initial HTML pages before the generated XSLT can be applied.
3. Finally, our users do not have to be familiar with a WYSIWYG editor and the need of providing structural information through the editing actions. The only thing that they need to possess is knowledge of a basic HTML tool.

In the process of generating XSL Transformations, XSLTGen involves a step of matching the supplied XML and HTML documents, i.e. finding semantic mappings between the XML and HTML tags. The rest of this section focuses on the related work in the area of tree matching and semantic mapping.

6.1 Tree Matching

There are a number of algorithms available for tree matching. Work done in [1, 28, 29, 32] on the tree distance problem or tree-to-tree correction problem and work done in [6, 20] known as the change-detection algorithm, compare and discover the sequence of edit operations needed to transform the source tree into the result tree given. These algorithms are mainly based on structure matching, and their input comprises of two labelled trees of the same type, i.e. two HTML trees or two XML trees. The text matching involved is very simple and limited since it compares only the labels of the trees. More recent algorithms on tree matching and main memory change detection for XML include the XyDiff system [9] and work in [18], which leverages relational database technology.

6.2 Semantic Mapping

In the field of semantic mapping, a significant amount of work has focused on schema matching (refer to [27] for survey). Schema matching is similar to our matching problem in the sense that two different schemas, with different sets of element names and data instances, are compared. However, the two schemas being compared are mostly from the same domain and therefore, their element names are different but comparable. Besides using structure matching, most of the schema mapping systems rely on element name matchers to match schemas. The TransSCM system [22] matches schema based on the structure and names of the SGML tags extracted from DTD files by using concept of labelled graphs. The Artemis system [2, 5] measures similarity of element names, data types and structure to match schemas. *In XSLTGen, it is impossible to compare the element names since XML and HTML have completely different tag names.*

XMapper [17] is a system built for finding semantic mappings between structured documents within a given domain, particularly XML sources. This system uses an inductive machine learning approach to improve accuracy of mappings for XML data sources, whose data types are either identical or very similar, and the tag names between these data sources are significantly different. In essence, this system is suitable for our matching process since the tag names of XML and HTML documents are absolutely different. However, this system is not automatic since it requires the user to select one matching tag between two documents.

The Clio system [15] is an interactive, semi-automated tool for computing schema matchings. It was introduced for the relational model in [21] and was based on *value correspondences* provided by the user, in order to create the corresponding data transformation/query. In [31], the system has been extended by using instances to refine schema matchings. Refinements are obtained by inferring schema matchings from operations applied to example data, which is done by the user who manipulates the data interactively. User interaction is also needed in [25] where a two-phase approach for schema matching is proposed. The second phase, called *semantic translation*, generates transformations that preserve given constraints on the schema. However, if few or even no constraints are available, the approach does not work well. It is clear that the algorithm for finding schema matching used in the Clio system is not suitable for our work, since user interaction is required along the phases of finding schema matchings. In addition to this, the Clio system is applicable only to structured and semi-structured data that can be described by a schema (a relational schema, a nested XML Schema, or DTD). It is not applicable to the exchange of documents or unstructured data (e.g. HTML documents, multimedia, and unstructured text) [25].

Recent work in the area of ontology matching also focuses on the problem of finding semantic mappings between two ontologies. GLUE system [10] employs machine learning techniques to semi-automatically create such semantic mappings. Given two ontologies: for each node in one ontology, the purpose is to find the most similar node in the other ontology using the notions of *Similarity Measures* and *Relaxation Labelling*. Similar to our matching process, the basis used in the *similarity measure* and *relaxation labelling* are data values and the

structure of the ontologies, respectively. However, GLUE is only capable of finding 1-1 mappings whereas our matching process is able to discover not only 1-1 mappings but also 1-m mappings and m-1 mappings (in substring mappings).

The main difference between mapping in XSLTGen and other mapping systems, is that in XSLTGen we believe that mappings exist between elements in the XML and HTML documents, since the HTML document is derived from the XML document by the user; whereas in other systems, the mappings may not exist. Moreover, the mappings generated by the matching process in XSLTGen are used to generate code (an XSLT stylesheet) and that is why the mappings found have to be accurate and complete, while in schema matching and ontology matching, the purpose is only to find the most similar nodes between the two sources, without further processing of the results. To accommodate the XSLT stylesheet generation, XSLTGen is capable of finding 1-1 mappings and 1-m mappings; whereas the other mapping systems focus exclusively on discovering 1-1 mappings. Besides this, the matching subsystem in XSLTGen has the advantage of having very similar and related data sources, since the HTML data is derived from the XML data. Hence, they can be used as the primary basis to find the mappings. In other systems, the data instances in the two sources are completely different, the only association that they have is that the sources come from the same domain. Following this argument, XSLTGen discovers the mappings between two different types of document, i.e. an XML and an HTML document, whereas the other systems compare two documents of the same type. Finally, another important aspect which differs XSLTGen from several other systems, is that the process of discovering the mappings which will then be used to generate XSLT stylesheet is completely automatic.

7 Conclusion

With the massive upsurge in the data exchange and publishing on the Web, simple conversion of data from its stored representation (XML) to its publishing format (HTML) is becoming increasingly important. XSLT plays a prominent role in transforming XML documents into HTML documents. However, XSLT is difficult for users to learn.

We have devised the XSLTGen system, a system for automatically generating an XSLT stylesheet, given a source XML document and a desired output HTML document. This is useful for helping users to learn XSLT. The main strong characteristics of the generated XSLT stylesheets are accuracy and reusability. We have described how the notions of text matching, structure matching and sequence checking, enable XSLTGen to discover not only 1-1 semantic mappings between the elements in the XML and HTML documents, but also 1-m mappings between the two documents. We have described a fully automatic XSLT generation system that generates XSLT rules based on the mappings found. Our experiments showed that XSLTGen can achieve high matching accuracy and produce high quality XSLT stylesheets.

References

1. D.T. Barnard, N. Duncan, and G. Clarke. Tree-to-tree Correction for Document Trees. Technical Report 95–372, Department of Computing and Information Science, Queen's University, Kingston, 1995.
2. S. Bergamaschi, S. Castano, S.D.C.D. Vimeracati, and M. Vincini. An Intelligent Approach to Information Integration. In *Proceedings of the 1st International Conference on Formal Ontology in Information Systems*, pages 253–267, Trento, Italy, June 1998.
3. A. Boukottaya, C. Vanoirbeek, F. Paganelli, and O.A. Khaled. Automating XML Documents Transformations: A Conceptual Modelling Based Approach. In *Proceedings of the 1st Asia-Pacific Conference on Conceptual Modelling*, pages 81–90, Dunedin, New Zealand, January 2004.
4. T. Bray, J. Paoli, C.M. Sperberg-McQueen, and E. Maler. *Extensible Markup Language (XML) 1.0 (Second Edition).* W3C Recommendation, October 2000. *http://www.w3.org/TR/REC − xml.*
5. S. Castano and V.D. Antonellis. A Schema Analysis and Reconciliation Tool Environment for Heterogeneous Databases. In *Proceedings of the 1999 International Database Engineering and Applications Symposium*, pages 53–62, Montreal, Canada, August 1999.
6. S.S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change Detection in Hierarchically Structured Information. In *Proceedings of the 1996 International Conference on Management of Data*, pages 493–504, Montreal, Canada, June 1996.
7. J. Clark. *XSL Transformation (XSLT) Version 1.0.* W3C Recommendation, November 1999. *http://www.w3.org/TR/xslt.*
8. J. Clark and S. DeRose. *XML Path Language (XPath) Version 1.0.* W3C Recommendation, November 1999. *http://www.w3.org/TR/xpath.*
9. G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in xml documents. In *ICDE*, pages 41–52, 2002.
10. A. Doan, J. Madhavan, P. Domingos, and A. Halevy. Learning to Map between Ontologies on the Semantic Web. In *Proceedings of the 11th International Conference on World Wide Web*, pages 662–673, Honolulu, USA, May 2002.
11. M. Erwig. Toward the Automatic Derivation of XML Transformations. In *Proceedings of the 1st International Workshop on XML Schema and Data Management*, pages 342–354, Chicago, USA, October 2003.
12. A.L. Hors et.al. *Document Object Model (DOM) Level 2 Core Specification Version 1.0.* W3C Recommendation, November 2000. *http://www.w3.org/TR/DOM-Level-2-Core.*
13. J. Fox. Generating XSLT with a Semantic Hub. In *Proceedings of the 2002 XML Conference*, Baltimore, USA, December 2002.
14. M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: Learning Document Type Descriptors from XML Document Collections. *Data Mining and Knowledge Discovery*, 7(1):23–56, January 2003.
15. L.M. Haas, R.J. Miller, B. Niswonger, M.T. Roth, P.M. Schwarx, and E.L. Wimmers. Transforming Heterogeneous Data with Database Middleware: Beyong Integration. *Bulleting of the IEEE Computer Society Technical Committee on Data Engineering*, 22(1):31–36, March 1999.
16. M. Kay. *XSLT Programmer's Reference.* Wrox Press Ltd., 2000.
17. L. Kurgan, W. Swiercz, and K.J. Cios. Semantic Mapping of XML Tags using Inductive Machine Learning. In *Proceedings of the 2002 International Conference on Machine Learning and Applications*, pages 99–109, Las Vegas, USA, June 2002.

18. E. Leonardi, S. Bhowmick, T. Dharma, and Madria S. Detecting content changes on ordered xml documents using relational databases. In *DEXA*, pages 580–590, 2004.
19. M. Leventhal. XSL Considered Harmful. *http://www.xml.com/pub/a/*1999/05/*xsl/xslconsidered*_1.*html*, 1999.
20. S. Lim and Y. Ng. An Automated Change-Detection Algorithm for HTML Documents Based on Semantic Hierarchies. In *Proceedings of the 17th International Conference on Data Engineering*, pages 303–312, Heidelberg, Germany, April 2001.
21. R.J. Miller, L.M. Haas, and M.A. Hernández. Schema Mapping as Query Discovery. In *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 77–88, Cairo, Egypt, September 2000.
22. T. Milo and S. Zohar. Using Schema Matching to Simplify Heterogeneous Data Translation. In *Proceedings of 24th International Conference on Very Large Data Bases*, pages 122–133, New York, USA, August 1998.
23. K.L. Nance and B. Hay. IDACT: Automating Data Discovery and Compilation. In *Proceedings of the 2004 Nasa's Earth Science Technology Conference*, Palo Alto, USA, June 2003.
24. K. Ono, T. Koyanagi, M. Abe, and M. Hori. XSLT Stylesheet Generation by Example with WYSIWYG Editing. In *Proceedings of the 2002 International Symposium on Applications and the Internet*, Nara, Japan, March 2002.
25. L. Popa, Y. Velegrakis, R.J. Miller, M.A. Hernández, and R. Fagin. Translating Web Data. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 598–609, Hong Kong, China, August 2002.
26. D. Raggett, A.L. Hors, and I. Jacobs. *Hypertext Markup Language (HTML) 4.01.* W3C Recommendation, December 1999. *http://www.w*3.*org/TR/html*4.
27. E. Rahm and P.A. Bernstein. A Survey of Approaches to Automatic Schema Matching. *VLDB Journal*, 10(4):334–350, December 2001.
28. S.M. Selkow. The Tree-to-Tree Editing Problem. *Information Processing Letters*, 6(6):184–186, December 1977.
29. K.C. Tai. The Tree-to-Tree Correction Problem. *Journal of the ACM*, 26(3):422–433, July 1979.
30. S. Waworuntu and J. Bailey. XSLTGen: A system for automatically generating XML transformations via semantic mappings. In *Proceedings of the 23rd International Conference on Conceptual Modeling (ER2004)*, volume LNCS 3288, pages 479–492, November, 2004.
31. L.L. Yan, R.J. Miller, L.M. Haas, and R. Fagin. Data-Driven Understanding and Refinement of Schema Mappings. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, Santa Barbara, USA, May 2001.
32. K. Zhang and D. Shasha. Simple Fast Algorithms for the Editing Distance between Trees and Related Problems. *SIAM Journal of Computing*, 18(6):1245–1262, December 1989.