Specifying Transformations for Structured Documents

Xuerong Tang and Frank Wm. Tompa
Department of Computer Science, University of Waterloo
{xtang,fwtompa}@uwaterloo.ca

Abstract

Structured document transformation is important for XML to serve as the universal data interchange format for the Web. W3C's XSLT is a powerful transformation language, but it requires detailed programming. As alternatives, SDT and TT grammars are widely used to specify transformations between programming languages. They have been proposed as descriptive specification techniques for structured document transformation, but they have limited expressive power. We propose a new approach based on syntax tree templates. Our approach is both descriptive and expressive as a specification language. We illustrate how a specification using this approach can be converted to XSLT for execution.

1 Introduction

Repeatedly we find that documents need to be converted from one form to another as they are exchanged among various applications. Such transformations will be especially important in manipulating data that is encoded in XML for widespread interoperability. When structural conversion is needed, the ideal transformation process includes three distinct phases: (1) determine the input and output constraints (e.g. grammars, DTDs, etc.) imposed on the document; (2) specify the transformation in terms of input/output instances (e.g. tree patterns) or input/output constraints that characterize the document; (3) choose an appropriate sequence of operations to carry out the transformation according to the specification.

It is natural to consider specifying a transformation using W3C's XSLT [Cla99b]. An XSLT program (called a stylesheet) is a set of template rules, each of which has two parts: a pattern that is matched against nodes in a source tree and a template that can be instantiated to form part of a result tree. XSLT and similar transformation languages, such as TXL [CP90], are functional programming languages with powerful computational capability. Such languages are well-designed for carrying out a transformation, but their operational nature makes them less desirable candidates for a *specification* language.

In this paper we define a high-level specification language that can be used by non-programmers to specify a structured document transformation descriptively. Our approach is motivated by paired grammars, first introduced in syntax-directed translation [AU72] and subsequently extended by tree transformation grammars [KPPM84] and input/output filters [ST99]. The first two formalisms, however, can express only simple transformations, and the third has incomplete semantics. Our approach instead includes mechanisms to specify constraining conditions and arbitrary mappings between a pair of syntax trees that each represent a forest-regular language [Mur98]. We show that our approach is both descriptive and expressive, and we illustrate how to translate from this high-level specification to an operational description algorithmically.

2 Paired SynTrees

In this section, we introduce paired syntax tree templates for specifying transformations. After their definitions, we illustrate their use through examples and show how they can be translated into XSLT scripts.

2.1 Preliminaries

A *syntax tree* represents the grammar for a document as a hierarchical tree structure. Kuikka et al. defined a *syntax tree template* as a (partial) syntax tree augmented so that a user may specify conditions on the nodes for the purpose of selection [KMRS95]. Salminen and Tompa introduced a promising idea for querying structured documents by means of *paired filters*, each of which is a sequence of *constraining context-free grammars*, but it is only partially defined for describing transformation [ST99].

Our approach extends these ideas, but also builds on forest-regular grammars in place of context-free grammars as the underlying schema language [Mur98]. We use the following definition of a forest regular grammar, in which the term "string-regular expression" denotes a conventional regular expression defined over sequences of characters.

> **Definition**: A *forest-regular grammar* is a 4-tuple $< S, N, P, r_f >$, where S is a finite set of symbols; N is a finite set of non-terminals; P is a finite set of production rules of the form $A \rightarrow a < r >$, where $A \in N$, $a \in S$, and r is a non-empty string-regular expression over $N \cup S$; and r_f is a string-regular expression over N. If r_f is a single non-terminal, the 4-tuple describes a *tree-regular grammar*.

The set of forests derived by applying the production rules in P to the expression r_f constitute the language defined by this grammar. Our simplification of the definition omits the set of variables that Murata uses to represent the external (leaf) nodes in a forest, which are not needed to support the specification of a transformation. It has been proven that each tree produced by a tree-regular grammar corresponds exactly to a parse tree produced by an analogous regular right-part grammar [Lal77] having S as its set of non-terminals. The effect is similar to that achieved by xscheme [Beh00], but forest-regular languages have the advantage of being closed under set union as well as under intersection and difference. Thus forest-regular grammars are a good fit for XML data.

2.2 Describing a transformation

Murata describes text transformations via forest automata, but we adopt instead the higher level approach of specifying transformations via pairs of grammatical representatives. Instead of pairing productions in syntax directed translation schemas or pairing filters consisting of constraining context-free grammars, however, we will pair syntax tree templates described as forest-regular expressions.

We assume that the input data is described by a forest-regular grammar or a DTD, and that the output schema is similarly described. Our specification language contains three components: a pair of "SynTrees" (one for input and the other for output), a set of boolean conditions and a set of mapping rules.

Each *SynTree* is derived by choosing one nonterminal from the appropriate grammar and repeatedly applying production rules to replace nonterminals, except that: (1) after the first application of any rule $A \rightarrow a < r >$, other replacements of that same nonterminal may use $a <>$ instead of $a < r >$, and (2) a nonterminal need not be replaced if its substructure remains unchanged. The first chosen nonterminal gives the context for the specified transformation and need not be the root symbol for the complete grammar. The notation $<>$ is used to indicate that the transformation of the substructure is provided elsewhere in the SynTree.

Each *boolean condition* is a labelled XPath predicate expression [Cla99a], for which the label is placed within square brackets [] in the SynTree. The conditions in the input SynTree are selection criteria, and the ones in the output SynTree serve as construction criteria, in both cases applied to the preceding symbol in the SynTree. We restrict the XPath expression to be within the local context of the node whose symbol is associated with the boolean condition.

Each *mapping rule* uses the following syntax: *operator* [(*parameter*1? *parameter*2? ...)]: $S1 \rightarrow S2$, where $S1$ is a set of nodes from the input SynTree and $S2$ from the output SynTree. The mapping rules specify which nodes from the input are to be transformed to which nodes in the output. Both the operators and parameters are chosen from a predefined closed set, which currently includes: *copy* (with parameters to indicate union vs. intersection, whether duplicates are allowed, and sorting criteria) and *aggregate* (with parameters to indicate grouping criteria, aggregation function, and sorting criteria). Both operators merge elements from the set of input nodes and stream the results to each of the output nodes, subject to the parameters and boolean conditions.

Finally, we define some default semantics that are implicit in the specification language of transformation:

1. If no selection criteria appears on nodes in the input SynTree, all the nodes are selected. Subtrees that match the input SynTree but are not selected by the boolean conditions are deleted.

2. If input and output nodes have the same name and there is no explicit mapping rule to the output node, then there is an implicit mapping rule to copy the input to the corresponding output.

3. Transformation assumes no information loss unless otherwise indicated by deletion or truncation. Thus any selected node that is the source of a mapping must be placed somewhere in the output document.

4. In order to create a document that matches the output grammar, every node that is required to have at least one occurrence in the instance must be the target of some mapping rule. Furthermore, any node that is the target of a mapping must be instantiated by the conversion.

Note that such default behavior does not apply in typical query languages, where data is neither retrieved nor transformed unless explicitly specified.

In summary, a paired SynTree specification attaches contextual conditions and mapping rules to syntax tree structures. The language is designed to make the set of operators compact, their semantics easy to follow, and common simple transformations easy to specify.

2.3 Examples of Paired SynTrees

Consider the transformation problem given by W3C's XML Query Working Group as *Use Case "PARTS"* [CFMR01]: The input is a flat list of *part* elements, each of which has values for *partid* and *name* attributes. Each *part* may or may not be a component of a larger *part*, indicated by the value of the *partof* attribute. The transformation is to convert the flat representation into a explicit hierarchic representation, based on *partof* attributes.

```
Input DTD:                                      Output DTD:
<!DOCTYPE partlist [                            <!DOCTYPE parttree [
   <!ELEMENT partlist   (part*)>                    <!ELEMENT parttree   (part*)>
   <!ELEMENT part        EMPTY>                     <!ELEMENT part       (part*)>
   <!ATTLIST part                                   <!ATTLIST part
      partid  CDATA   #REQUIRED                         partid  CDATA   #REQUIRED
      partof  CDATA   #IMPLIED                          name    CDATA   #REQUIRED> ]>
      name    CDATA   #REQUIRED> ]>
```

The input forest-regular grammar is captured by the following productions:

PartList → partlist< Part* >
Part → part< @partid @partof? @name >

The output forest-regular grammar has the following productions:

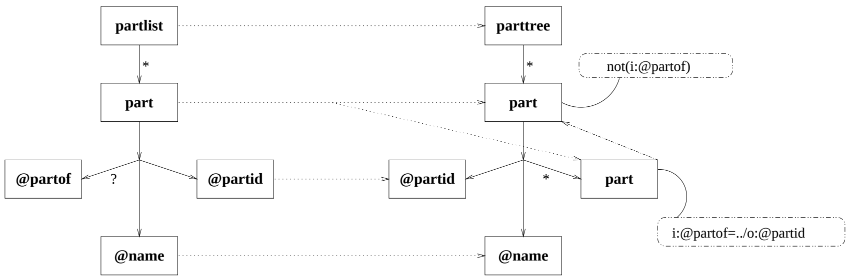
PartTree → parttree< Part* >
Part → part< @partid @name Part* >

For this example, a paired SynTree corresponding to the use case is specified as follows (where prefixes 'i' and 'o' represent input and output namespaces, respectively):

Input SynTree:	i:partlist< i:part*< i:@partid i:@partof? i:@name >>
Output SynTree:	o:parttree< o:part[C1]*< o:@partid o:@partof o:part[C2]*<> >>
Boolean Conditions:	C1: not(i:@partof)
	C2: i:@partof = ../o:@partid
Mapping Rules:	*copy*: i:partlist → o:parttree
	copy: i:part → o:part[1] o:part[2]

In the specification, we use @ to denote attribute names. Note that in the output SynTree, the symbol *o:part* appears twice; the first time its substructure is described as a nested forest, but for the second usage, the notation <> is used. In the mapping rule, *part*[i] refers to the i-th occurrence of the *part* symbol in the corresponding SynTree. The meaning of the first *copy* in the mapping rules is straightforward node copy, and in this case the renaming of a node. The second *copy* takes nodes from the input document and transforms them to one (or more) of the nodes of the output document, depending on the values of the conditions attached to the corresponding symbols in the output SynTree; in this case, the operation results in a partitioning of the input nodes. Copying of *@partid* and *@name* to corresponding nodes in the output is implicit and is therefore omitted in the specification.

A paired SynTree specification can be visualized as a directed graph, which may be useful as a graphical user interface for creating or for understanding transformation specifications (cf. [CCD+99]). The following figure visualizes the paired SynTrees given above.



As a second example, consider a university calendar that includes a list of university professors, organized first by department, and within each department, partitioned by rank. As a further complication, assume that professors who are officers in a department are listed separately after the department name instead of appearing in the standard listings. Assume now that a transformed listing is to be produced to include only members of the Departments of Computer Science and of Electrical and Computing Engineering, but this list is to be strictly partitioned by rank, with the department name appearing as an attribute for each high-tech professor. The number of the professors for each rank is also aggregated and stored in a new attribute *num* for each section in the output.

```
Input DTD:
<!DOCTYPE university [ ...
   <!ELEMENT proflist   (department*)>
   <!ELEMENT department (head, section*)>
   <!ATTLIST department
             title  CDATA  #REQUIRED>
   <!ELEMENT head       (official*)>
   <!ELEMENT official   (professor, rank)>
   <!ATTLIST official
             position  CDATA  #REQUIRED>
   <!ELEMENT section    (rank, professor*)>
   <!ELEMENT rank       #PCDATA>
   <!ELEMENT professor  #PCDATA>  ]>
```

```
Output DTD:
<!DOCTYPE university [ ...
   <!ELEMENT proflist (section*)>
   <!ELEMENT section (hitech_prof*)>
   <!ATTLIST section
             rank CDATA #REQUIRED
             num  CDATA #REQUIRED>
   <!ELEMENT hitech_prof #PCDATA>
   <!ATTLIST hitech_prof
             dept_title CDATA #REQUIRED  ]>
```

The transformation can be specified by the following paired SynTrees:

Input SynTree: i:proflist< i:department[C1]*< i:@title
i:head< i:official*< i:@position i:professor i:rank >>
i:section*< i:rank i:professor* >>>

Output SynTree: o:proflist< o:section*< o:@rank o:@num[C2] o:hitech_prof[C2]*< o:@dept_title >>>

Boolean Conditions: C1: i:@title="ECE" *or* i:@title="CS"
C2: ../i:@rank = ../o:rank

Mapping Rules: *copy*(*distinct*): i:rank[1] i:rank[2] → o:@rank
copy: i:@title → o:@dept_title
copy(*union*): i:professor[1] i:professor[2] → o:hitech_prof
aggregate(*count*): i:professor[1] i:professor[2] → o:@num

Because the roots of the SynTrees correspond to subtrees within the university document, the transformation is applied to those subtrees only, without altering the remainder of the document. Notice that the mapping rules, together with boolean condition C2, specify that the output listing is to include each rank exactly once, and that each professor is to appear under the correct rank name (according to the associated rank in the input document, whether or not that professor was an officer), and the number of professors under the same section is to be counted as the *num* attribute.

2.4 Translation from Paired SynTrees to XSLT

For paired SynTrees to be a useful specification mechanism, we need algorithms to convert from the description of a specification to a sequence of operations that carry out the transformation on document instances.

XSLT provides a convenient and powerful language for expressing the operational behavior of a specified transformation. We observe that in general there are many possible XSLT programs that can carry out a given transformation, and in this section we choose programs that are "output-driven," i.e., where XSLT templates correspond to components that are to be realized in the output document. This contrasts, for example, with an alternative, "input-driven" programming methodology that would produce XSLT templates for each source component that is to be placed in the output.

We illustrate the implementation algorithm by examining the program that can be built to realize the specification for the "PARTS" Use Case, as given in the previous section. Because the input SynTree has no selection criteria, all nodes in the input document are implicitly selected. A top-down tree automaton, called a *tree transducer*, walks down the output SynTree and iteratively translates the specification into XSLT. When the first node that is the target of some mapping rule is reached, the tree transducer will produce the first XSLT template which constructs the root *parttree* of the output tree and then calls the next XSLT template.

The next node encountered in the output SynTree is one of two nested nodes labelled by the same nonterminal *part*, and this self-nesting implies a construction of recursive XSLT templates. The parent *part* node has a construction condition that is to be imposed on instance nodes: only those nodes for which the attribute *partof* is missing in the input document will be transformed to become the *part* node at the highest level in the output instance. Recursively, the children of any *part* node in the output parse tree can be constructed according to the construction criterion of the child *part* node in the output SynTree: the *partof* attributes of the children match the *partid* attribute of the parent node. The mapping between *part* nodes specifies the semantic rule *copy*, which in this example means that the *part* nodes in the input will be divided according to the two given construction criteria.

The corresponding XSLT script for the complete transformation will be produced as follows:

```
<xsl:template match="partlist">
  <partree><xsl:apply-templates mode="part1"/></parttree>
</xsl:template>
```

```
<xsl:template mode="part1" match="part">
 <xsl:if test="not(@partof)">
  <part>
   <xsl:variable name="vPartid"/>
       <xsl:value-of select="@partid"/>
   </xsl:variable>
   <xsl:attribute name="partid">
        <xsl:value-of select="@partid"/>
   </xsl:attribute>
   <xsl:attribute name="name">
        <xsl:value-of select="@name"/>
   </xsl:attribute>
   <xsl:apply-templates mode="part2"
                        select="../part">
      <xsl:with-param name="paramPartid"
                      select="$vPartid"/>
   </xsl:apply-templates>
  </part>
 </xsl:if>
</xsl:template>
```

```
<xsl:template mode="part2" match="part">
 <xsl:param name="paramPartid">1</xsl:param>
  <xsl:if test = "@partof=$paramPartid">
   <part>
     <xsl:variable name="vPartid2"/>
          <xsl:value-of select="@partid"/>
     </xsl:variable>
     <xsl:attribute name="partid">
          <xsl:value-of select="@partid"/>
     </xsl:attribute>
     <xsl:attribute name="name">
          <xsl:value-of select="@name"/>
     </xsl:attribute>
     <xsl:apply-templates mode="part2"
                          select="../part">
        <xsl:with-param name="paramPartid"
                        select="$vPartid2"/>
     </xsl:apply-templates>
   </part>
  </xsl:if>
</xsl:template>
```

3 Conclusion

Prior to developing paired SynTrees, we examined several approaches for specifying transformations for structured documents. None of them seem suitable as a high-level specification language: some are too operational in nature and others can describe local transformations only. We therefore propose a new approach, paired syntax tree templates, which is both descriptive and expressive.

Many XML query languages, such as XQuery [CFR+01] and XML-GL [CCD+99], provide transformation capabilities to varying degrees. Similar to the XSLT solution, the XQuery solution for the "PARTS" Use Case example [CFMR01] also uses a recursive function. Both solutions explicitly specify in which order to carry out the transformation (via recursive calls), and in this sense they are clearly operational rather than descriptive.

Our belief is that a query language will seldom be a good transformation specification language. The primary design of these languages is query-oriented, so they seldom support a complex structure transformation that is to comply to an explicitly defined output DTD. Furthermore, in a query language such as XQuery, a program to perform even a simple change to the structure, such as renaming of a single nonterminal node or moving a small set of nodes, involves recursively rebuilding much of the tree. As a result, such programs are neither efficient nor concise as a specification mechanism for structural transformations.

We believe that a separate, more dedicated transformation language can complement a query language and that paired SynTrees is such a language. Our ongoing research will study algorithms for translating from our specification language to an operational language and to investigate possible optimizations when there are several translations available for one specification.

References

[AU72] A.V. Aho and J.D. Ullman. *The Theory of Parsing, Translation, and Compiling, Vol 1: Parsing*, chapter 3.1 and 3.2. Prentice-Hall, 1972.

[Beh00] Ralf Behrens. A grammar based model for XML schema integration. In *British National Conf. on Database*, pages 172–190, 2000.

[CCD+99] Stefano Ceri, Sara Comai, Ernesto Damiani, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca. XML-GL: a graphical language for querying and restructuring XML. In *Proc. of WWW8, Toronto*, pages 1171–1187, Canada, 1999.

[CFMR01] Don Chamberlin, Peter Fankhauser, Massimo Marchiori, and Jonathan Robie. XML Query Use Cases. W3C Working Draft, February 2001. http://www.w3.org/TR/xmlquery-use-cases.

[CFR+01] Don Chamberlin, Daniela Florescu, Jonathan Robie, Jérôme Siméon, and Mugur Stefanescu. XQuery: A query language for XML. W3C Working Draft, February 2001. http://www.w3.org/TR/xquery.

[Cla99a] James Clark. XML Path language (XPath) version 1.0. W3C Recommendation, November 1999. http://www.w3.org/TR/xpath.

[Cla99b] James Clark. XSL transformations (XSLT) version 1.0. W3C Recommendation, November 1999. http://www.w3.org/TR/xslt.

[CP90] James R. Cordy and Eric Promislow. Specification and automatic prototype implementation of polymorphic objects in TURING using the TXL dialect processor. In *Proc. IEEE International Conference on Computer Languages*, New Orleans, 1990.

[KMRS95] Eila Kuikka, Jouni Mykkänen, Arto Ryynäen, and Airi Salminen. Implementation of two-dimensional filters for sturctured documents in SYNDOC environment. Technical report, University of Joensuu, 1995.

[KPPM84] S.E. Keller, J.A. Perkins, T.F. Payton, and S.P. Mardinly. Tree transformation techniques and experiences. In *Proc. of the ACM SIGPLAN '84 Symposium on Compiler Construction*, 1984.

[Lal77] W.R. Lalonde. Regular right part grammars and their parsers. *Comm. of the ACM*, 20(10):731–741, 1977.

[Mur98] Makoto Murata. Data model for document transformation and assembly (extended abstract). In *Principle on Digital Document Processing*, pages 140–152, 1998.

[ST99] Airi Salminen and Frank W. Tompa. Grammars++ for modelling information in text. *Information Systems*, 24(1):1–24, 1999.