

Serial Tree Transducers and their Implementation in STX

Working Paper

Tobias Trapp*
AOK Systems GmbH
Frankfurt, Germany

October 6, 2006

Abstract

This paper describes implementations of tree transducers using the Serial Transformations for XML (STX) language. We define fragments of STX that correspond to tree transducers with no lookahead and a limited number of registers. These tree transducers can be used for fast and scalable XML transformations. To overcome the limited expressiveness of tree transducers we extend them by general subtree transformations.

1 Introduction

Without the Extensible Markup Language today's electronic data exchange processes are hardly conceivable. Electronic business processes based on data exchange require standardized information exchange formats. XML Schema languages enable you to formally specify data exchange formats.

Therefore, in data exchange we convert XML documents that are valid according to one schema into a document that is valid according to another one. To perform this task often we use transformation languages. These transformations are the core of modern integration technology like SAP Exchange Infrastructure. Unfortunately most transformation languages have performance problems when dealing with mass data. Therefore we restrict ourselves to serial transformation languages in this paper.

STX (see [Cimprich et al.]) is a serial XML transformation language that is Turing complete and can transform XML documents in a stream like fashion. It is an open question whether this language can be applied in real life data exchange scenarios. In those scenarios XML transducers are generated from a schema mapping. It is not unlikely that there are many schema mappings which allow tree transducers to work in a stream-like fashion. In this paper two STX fragments will be defined that are top-down tree transducers.

This is a list of the main differences between STX and XSLT (see [Clark]):

*Tobias.Trapp@sys.aok.de

- STX is a procedural programming language: we can assign variables many times and can perform loops.
- STX uses STXPath as query language. STXPath is derived from XPath 2.0 but can access only the ancestors of a certain node.
- STX is event-based: STX templates are executed in the specific order of the input stream.
- STX has no named templates but we can define procedures.
- There are no commands like `xsl:for-each` that can change the current node.

STX uses the XSLT 1.0 data model. It supports features of XSLT 2.0 like multiple output documents and text processing. There is a discussion of STX in [Becker]. In this dissertation we find the following example for renaming elements:

```
<stx:transform xmlns:stx="http://stx.sourceforge.net/2002/ns"
  version="1.0" pass-through="all" output-method="xml">

  <stx:template match="frage">
    <question>
      <stx:process-attributes/>
      <stx:process-children/>
    </question>
  </stx:template>
</stx:transform>
```

This program copies the whole input document into the output document using the statement `pass-through="all"`. For `frage`-elements a special transformation rule is defined using `<stx:template match="frage">`. Everytime this element occurs we insert an element `question` and the attributes and children of the element `frage`.

2 Definitions and Notations

Due to its hierarchical structure we can model XML based languages as tree languages. For the reader who is not familiar with this concept let me present the basic facts. Tree languages are a generalization of word languages because they can be seen as word languages in an infinite alphabet of contexts. Many concepts from regular word languages can be taken to regular tree languages. In fact we already know this language class because the set of derivation trees of a context-free language is a regular tree language, but the opposite is not true. For the interested reader I refer to [Comon et al.].

In this document we do not need much help from language theory. We model XML documents as finite ordered trees, the nodes of those trees correspond to the elements of an XML document. We do not consider the rest of the XML

Infoset especially not the attributes and namespaces for two reasons: they can be seen as special nodes, furthermore will we focus on structural conversions of XML documents.

We define \mathcal{T}_Σ as the set of finite ordered trees over the set Σ , in short Σ -tree. We denote ε as the empty tree. If a tree - resp. subtree - t has ordered children $a_1 \cdots a_n$ we write $t(a_1 \cdots a_n)$, if a tree t has subtrees $t_1 \cdots t_n$ we write $t(t_1 \cdots t_n)$.

Let \mathcal{F}_Σ be the set of Σ -forests, that are finite sequences of Σ -trees. For a set Q we define $\mathcal{T}_\Sigma(Q)$ resp. $\mathcal{F}_\Sigma(Q)$ as the set of trees - resp. forests - that have leaves can be labeled with elements of Q . Let $\mathcal{F}_\Sigma^1(Q)$ denote the set of forests in $f \in \mathcal{F}_\Sigma(Q)$ such that f contains at the most one node labeled $p \in Q$.

3 Serial Tree Transducers and their STX implementation

Now we define tree transducers: You can think of them as programs that convert one tree into another. In the following we define a restricted version of the uniform tree transducer defined in [Martens and Neven] that works on ordered trees (so it is not uniform) but restricts the rules such that we cannot double subtrees within an XML document.

At first we will define the serial tree transducer and then the translation which is performed by the transducer. A serial tree transformation works as follows: It walks top-down through the input tree in a depth first way. Everytime a node is encountered, a rule is chosen depending on the current node and state. The rules add a forest for each child-node to the output tree according to the selected rule. In each forest there is at most one subtree with one leaf that corresponds to a state of the transducer and goes on recursively in that state.

Definition 3.1 (Serial Tree Transducer) *A serial tree transducer is a tuple $(Q, \Sigma_S, \Sigma_T, q_0, R)$ where Q is a finite set of states, Σ_S is the input and Σ_T the output alphabet, $q_0 \in Q$ is the initial state, and R is a finite set of rules of the form $(q, a) \rightarrow h$, where $a \in \Sigma_S$, $q \in Q$, $h \in \mathcal{F}_{\Sigma_T}^1(Q)$. When $q = q_0$, h is restricted to $\mathcal{T}_{\Sigma_T}^1(Q) \setminus Q$.*

The restriction that h contains at most one leaf labeled with an element in Q will ensure that at each translation step at the most only one rule will be applied. The second restriction on rules with initial state ensures that the output is always a tree instead of a forest; in fact we expect that property for every tree transducer. If we would allow that the rule for the start symbol would be a tree h containing only one node $p \in Q$ then the rule for this node could produce a forest in the corresponding translation step so that the resulting XML document would contain multiple roots.

The translation defined by a serial tree transducer T on a tree t in state q , denoted by $T^q(t)$, is inductively defined as is defined as in [Martens and Neven]: If $t = \varepsilon$ then $T^q(t) := \varepsilon$; if t is a forest with root a and subtrees $(t_1 \cdots t_n)$ (i.e.

$t = a(t_1 \cdots t_n)$) and there is a rule $(q, a) \rightarrow h$ then $T^q(t)$ is obtained from h by replacing the node u in h labeled with $p \in Q$ by the forest $T^p(t_1) \cdots T^p(t_n)$. If there is no rule $(q, a) \rightarrow h \in R$ then $T^q(t) := \varepsilon$. Finally the transformation of a tree t by T is $T^{q_0}(t)$.

Example 1. Let $T = (Q, \Sigma, \Sigma, q, R)$ where $Q = \{p, q\}$ and R contains the rules:

$$(p, a) \rightarrow d(e)q, (p, b) \rightarrow c(q), (q, a) \rightarrow f(p).$$

Let us consider following input document: $\mathbf{b(aab(ab))}$. It will be translated to $\mathbf{f(d(e)d(e)c(f))}$. In the following I present an implementation in STX.

```
<stx:template match="/">
  <stx:process-children group="q"/>
</stx:template>

<stx:group name="p">
  <stx:template match="a">
    <d>
      <e/>
    </d>
    <stx:process-children group="q"/>
  </stx:template>

  <stx:template match="b">
    <c>
      <stx:process-children group="q"/>
    </c>
  </stx:template>
</stx:group>

<stx:group name="q">
  <stx:template match="a">
    <f>
      <stx:process-children group="p"/>
    </f>
  </stx:template>
</stx:group>
```

Like in XSLT an STX program consists of templates. The transformation is achieved by associating events with templates. A template pattern is matched against events and their context. In STX a context is defined by current node data, the ancestor stack and the position within siblings but in the example above we do not use the context information.

Within the templates we give out literal result elements (these are the elements without namespace) and continue the processing of the current template by processing the children of the current node using the command `stx:process-children`.

To support the state concept of the tree transducer we group the templates using the command `stx:group`. In each command `stx:process-children` we use the `group` attribute to force that only templates within in certain group are applied. The template `stx:template match="/"` defines the start state using `stx:process-children group="p"`. In fact grouping in STX is similar to the concepts of modes in XSLT.

A template must contain a command that continues the processing. In the templates above we used `stx:process-children` that splits up the template. The STX commands thereafter will be executed when the end-tag of the matched elements is processed in the XML input.

This example shows how to implement serial tree transducers in STX:

- Every state is modelled by a group of templates using `stx:group`.
- Every rule corresponds to a template within in a group.
- Within each template we give out literal result elements and continue processing in a certain state.

4 Serial Tree Transducers with a Limited Number of Registers

In STX we can define global variables that can be changed during processing. We will use them as registers and define test expressions that allow us to modify the state of a rule, thereby registers can be either integers or strings. To each rule $(q, a) \rightarrow h$ we assign a set of assignments. Assignments to integer variables can be arithmetical expressions containing integer constants and integer variables. Registers containing strings can be assigned to the text content of a or to constant text.

Now we modify the translation defined by a serial tree transducer T on a tree t in state q , denoted by $T^q(t)$. At each time a rule $(q, a) \rightarrow h$ is executed, value assignments are executed first. Then a new state $q' \in Q$ is evaluated by a set of logical expressions containing registers, constants and the text content of a . Then the execution of the rule is defined as in the section 3.

Example In the following example we do a projection and skip every second element. That means that `a(aaaa)` will be translated to `b(bb)`.

$$(p, a) \rightarrow b(q), (r, a) \rightarrow \varepsilon q.$$

The rule (q, a) is defined as follows:
`var := var + 1.`
`if var modulo 2 = 0 then $\rightarrow a q$ else $\rightarrow a r$`

Here `var` is a register that is first incremented and than evaluated in order to change the transducer state. Now we define an STX program for the transformation above:

```

<stx:variable name="var" select="0"/>

<stx:template match="/">
  <stx:process-children group="p"/>
</stx:template>

<stx:group name="p">
  <stx:template match="a">
    <b>
      <stx:process-children group="q"/>
    </b>
  </stx:template>
</stx:group>

<stx:group name="q">
  <stx:template match="a">
    <stx:assign name="var" select="$var + 1"/>
    <stx:if test="$var mod 2 = 0">
      <b/>
    </stx:if>
    <stx:process-children group="q"/>
  </stx:template>
</stx:group>

```

4.1 Dealing with Text Nodes

The serial tree transducer as defined above can only perform structural mappings because up to now we worked only on empty nodes: `<a/>`. Usually XML elements have a text content `<a>text` that can be modeled by an additional child node that represents the text.

There are several ways to support text nodes: they can be modeled as additional nodes in the input resp. output alphabet. From a practical point of view it would be better to add more flexibility. Therefore we extend for every rule $(q, a) \rightarrow h$ every node of the forest h with labels. These labels mean that during the translation step the text content of each node should be assigned to either literal text or the content of element `a` or the content of a register. In STX this can be done using following commands:

```

<b>
  <c>
    <stx:text>Literal Text</stx:text>
  </c>
  <d>
    <stx:value-of select="text()"/>
  </d>
  <e>
    <stx:value-of select="$var"/>

```

```
</e>
</b>
```

5 Enhanced Serial Tree Transducers

An *enhanced serial tree transducer* is a tree transducer with a limited number of registers that can perform arbitrary XML transformations on a subtree. It allows to create transformations in a ‘divide and conquer’ approach. If a serial transducer cannot perform a certain transformation we enhance it by *subroutines* that are more powerful.

The translation defined by an enhanced serial tree transducer T on a tree t in state q , denoted by $T^q(t)$, is the same as the serial tree transducer with registers R but the rules can have also the form $q, a, R, f \rightarrow h$ where $a \in \Sigma_S$, $q \in Q$, $h \in \mathcal{F}_{\Sigma_T}^1(Q)$ and f is an arbitrary computable function $f(q, a, R) \rightarrow \mathcal{F}_{\Sigma_T}^1(Q)$. The translation step of this rule is defined by replacing each subtree with the result of the function f applied on the current state, the root a of every subtree and the set of registers R .

The result of the function f applied on these parameters is a forest with leaves in $\Sigma \cup Q$. This forest can be created by a Turing complete XML transformation language like XSLT. In STX we can implement these rules using the `stx:buffer` command that copies subtrees into a buffers that are used as input of an XSL transformation. This is shown by following example. We want to implement an enhanced serial tree transducer that works on child elements `a` or an root element `root` by copying the subtrees of the elements `a` after sorting the siblings. That means that the XML document on the left hand side will be copied to the element on the right hand side:

| | |
|--|--|
| <pre><root> <a> <z/> <y/> <w/> <a> <a/> </root></pre> | <pre><root> <a> <w/> <y/> <z/> <a> <a/> </root></pre> |
|--|--|

The following transformation defines rules for the element `root` and its children `a`. The subtrees of the elements `a` are copied to a buffer `input` that is processed using an XSL transformation and the result is copied to the output stream. This is done using the following transformation:

```
<stx:transform xmlns:stx="http://stx.sourceforge.net/2002/ns"
  version="1.0" pass-through="none" output-method="xml">
```

```

<stx:variable name="var" select="0"/>

<stx:template match="/">
  <stx:process-children group="p"/>
</stx:template>

<stx:group name="p">

  <stx:template match="root">
    <root>
      <stx:process-children/>
    </root>
  </stx:template>

  <stx:template match="a">
    <stx:buffer name="input"/>
    <stx:buffer name="output"/>

    <stx:result-buffer name="input">
      <stx:process-self group="copy"/>
    </stx:result-buffer>

    <stx:result-buffer name="output" clear="yes">
      <stx:process-buffer name="input"
        filter-method="http://www.w3.org/1999/XSL/Transform"
        filter-src="url('./sort.xml')"/>
    </stx:result-buffer>

    <stx:process-buffer name="output" group="copy"/>

    <stx:process-siblings/>

  </stx:template>
</stx:group>

  <stx:group name="copy" pass-through="all"/>

</stx:transform>

```

The XSL transformation `sort.xml` performs the sorting operation:

```

<xsl:transform version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="*">
    <xsl:copy>
      <xsl:apply-templates>
        <xsl:sort select="name()"/>
      </xsl:apply-templates>
    </xsl:copy>
  </xsl:template>
</xsl:transform>

```

```

        </xsl:apply-templates>
    </xsl:copy>
</xsl:template >
</xsl:transform>

```

This example shows that enhanced tree transducers can easily perform permutations of the input tree.

6 Conclusion and Further Research

In this paper three STX fragments have been defined in terms of top-down tree transducers. From a theoretical point of view it would be interesting to prove their basic properties: closedness under composition and the expressiveness in terms of language theory. Furthermore we can describe these transducers in terms of tree language theory or logic.

For practical purposes it could be interesting to define stronger transducers that use the STXPath language. Even more it would be interesting to find a fragment that is so expressive that it can be applied to real life data exchange scenarios. In those scenarios XML transformations called *XML mappings* are generated from schema mappings often using XSLT likes described in [Kuikka et al.]. Syntax directed translation schemas (see [Aho and Ullman]) and tree grammars (see [Keller et al.]) are formalizations of schema mappings. Following questions arise immediately:

- Can we recognize if an XML mapping for a schema mapping can be realized with a serial tree transducer?
- Can we find an algorithm that generates serial tree transducers from schema mappings?

For practical purposes in data exchange enhancing STX with XSLT seems to be promising, because XSLT does not perform well if the XML documents are huge. In most real world data exchange scenarios with mass data XML documents are flat and valid to non-recursive XML schemas. Instead they consist of extremely wide trees that contain a list of millions of serialized business objects like invoices.

Of course we can perform any transformation using a trivial enhanced serial transducer that contains only one rule and performs the whole transformation using a single subroutine. If we implement this subroutine with XSLT this would of course rise to performance problems so it would be better to restrict the subroutine functions to single business objects so that they work on small subtrees. Thus the main question is whether or not we can find an algorithm that generates *good* enhanced serial tree transducers from schema mappings.

References

- [Aho and Ullman] The theory of parsing, translation and compiling, Vol. I: Parsing. Prentice Hall, 1972

- [Becker] Serielle Transformationen von XML, Dissertation, 2004
- [Cimprich et al.] Streaming Transformations for XML (STX),
Version 1.0, Working Draft 1 July 2004,
<http://stx.sourceforge.net/documents/spec-stx-20040701.html>
- [Comon et al.] Tree Automata Techniques and Applications, 2005,
<http://www.grappa.univ-lille3.fr/tata>
- [Clark] XSL Transformations (XSLT) Version 1.0,
W3C Recommendation 16 November 1999,
<http://www.w3.org/TR/1999/REC-xslt-19991116>
- [Keller et al.] Tree transformation techniques and experience, Proceedings of the ACM SIGPLAN 84 Symposium on Compiler Construction, SIGPLAN Notices, 19(6), 1984
- [Kuikka et al.] An Approach to Document Structure Transformations, Proceedings of Conference on Software: Theory and Practice, pages 906-913, 16th IFIP World Computer Congress 2006, Beijing, China
- [Martens and Neven] Typechecking Top-Down Uniform Unranked Tree Transducers, Proceedings of the 9th International Conference on Database Theory (ICDT'03), Berlin, Springer, 2003, p. 64-78