Towards Analyzing Space Complexity of Streaming XML Transformations

Jana Dvořáková
Department Of Software Engineering
Faculty of Mathematics and Physics
Charles University in Prague, Czech republic
Email: Jana.Dvorakova@mff.cuni.cz

Abstract—We present a formal framework that enables us to analyze space complexity of automatic streaming processing of XML transformations. Within the framework, the classes of XML transformations as well as the streaming algorithms are represented as formal models. The efficiency of the algorithms designed is proved with mathematical rigor by simulations of transformation models by streaming models. Subsequently, we design an efficient stack-based streaming algorithm for processing top-down XML transformations. It demonstrates the usage of the framework, but at the same time it can be directly incorporated into applications if needed.

I. Introduction

In this paper, we deal with streaming processing of transformations of XML documents. XML transformations are typically processed in a tree-based manner, i.e., the whole input document is read into memory and then particular transformation steps are performed following the specification. On the contrary, during streaming processing, the input document is read as a stream, and the output document is generated as a stream. Such processing of XML transformation is needed in the case the input XML data are in the form of stream as well as in the case the input is too large to fit into memory (e.g., database exports, RDF data).

Currently, the most frequently used XML transformation languages are XSLT and XQuery that are high-level, Turing-complete transformation languages intended for tree-based processing. In this paper, we focus on their automatic streaming processing. The traditional tree-based processors of XSLT and XQuery process the transformation directly by manipulation the in-memory representation. Obviously, it is much harder to program a streaming processor - efficient methods for handling the non-streaming constructs of the languages must be found.

The main contribution of this paper is two-fold. First, we present a formal framework that enables us to analyze space complexity of automatic streaming processing, and second, we present a stack-based streaming algorithm for XML transformations developed within the framework. The algorithm demonstrates the usage of our framework, but, at the same time, it can be directly incorporated into applications if needed.

The framework consists of two models: (1) the general model for tree-based processing of XSLT and XQuery, and (2) the streaming model for streaming processing. The framework allows us to specify natural restrictions of the general model which represent the subclasses of XML transformations. The key task is to identify restrictions of the streaming model (specifically, the size of the memory and the memory operations) such that the restricted streaming model can still simulate the given restricted general model. The resulting simulations represent the desired automatic streaming algorithms. An important feature of such algorithms is that we exactly know which classes of XML transformations they capture and how much memory they consume. The framework is easily extensible since new restricted transducers may be added, as well as optimizable since the transducer for a given transformation class may be replaced by more efficient one if found. When designing the models, we started from the transducers presented in [5], [6]. These, however, capture only subsets of XML transformations while the new models presented in this paper are Turing-complete.

The stack-based streaming algorithm designed in our framework captures top-down XML transformations. It uses stack size proportional to the depth of the input XML document - this is not a big restriction, since real-world XML documents contains only few levels of elements.

Related work. To our best knowledge, ours is the first approach to analyzing computational complexity of streaming XML transformations with mathematical rigor. Most of the earlier work was devoted to analyzing the streaming processing of the querying language XPath, e.g. [1], [2], [8]. Also several automatic streaming processors for XQuery [7], [10] and XSLT [9] have been implemented and their effectiveness was examined through empirical tests. The results obtained in all of these works show that streaming processors indeed consume less memory than tree-based processors - however, the results hold only for one or more specific transformations chosen for experiments.

II. Formal framework

The framework contains two novel models for both ways of processing XML transformations:
1) general XML transducer (GXT) and its restrictions,
2) streaming XML transducer (SXT) and its restrictions.

The transducers are defined in common terms in order to facilitate development of the streaming algorithms. The overall schema of the framework is shown in Fig. 1.

A. Notions and notations

Prior to defining the transducers formally, we introduce the abstract model of XML document and the subset of XPath expressions considered.

XML tree. Let \( \Sigma \) be an alphabet of element names. The set of XML trees over \( \Sigma \) is denoted by \( T_\Sigma \). An indexed XML tree may in addition have some leaves labeled by symbols from a given set \( I \). A set of XML trees indexed by \( I \) is denoted by \( T_\Sigma(I) \). We identify nodes of XML tree by dynamic level numbers - the root is identified indexed by \( I \) symbols from a given set. XML tree may in addition have some leaves labeled by symbols from a given set.

XPath expressions considered.

XML stream. XML stream is a flat representation of XML document. Let \( \Sigma \) be an alphabet of names. The corresponding alphabet of XML tags is defined by \( \Sigma_{tag} = \{ \sigma^1 \mid \sigma \in \Sigma \} \cup \{ \sigma^1 \mid \sigma \in \Sigma \}^2 \). We define a conversion function \( \text{stream} : T_\Sigma \to \Sigma_{tag}^* \) by

\[
\text{stream}(t) = \sigma^1 \text{stream}(t_1) \ldots \text{stream}(t_n) \sigma^1
\]

where \( \sigma = \lambda_i(1) \) is the label of the root of \( t \) and \( t_1, \ldots, t_n \) are the subtrees of the root. Then XML stream is a balanced sequence of start-tags and end-tags.

XPath expressions. We use a subset of XPath expressions to navigate within XML trees. Since XML trees contain element nodes only, we have excluded the constructs for handling other types of nodes. For simplicity, we use the non-abbreviated XPath syntax only. We consider a fixed set of names \( \Sigma \) and a fixed set of variable \( X \), and we denote the set of XPath expressions with names from \( \Sigma \) and variables from \( X \) by \( XPath(\Sigma, X) \).

A function evaluating the selecting expression \( \theta \) in an XML tree \( t \in T_\Sigma \) at a context node \( u \in V_t \) with variable values \( val_1, \ldots, val_m \) is denoted by

\[
\text{eval}(\theta, t, u)[x_1 \leftarrow val_1, \ldots, x_m \leftarrow val_m].
\]

The context set always contains a single node only. We call the nodes returned by evaluation the matched nodes.

B. General XML transducer

The general XML transducer (GXT) is a tree-to-tree transducer with one or more input heads, and one or more output heads. Each input head is associated with exactly one output head. The input heads traverse the input tree

\[
1\text{We consider XML trees without data values.}
2\text{The start-tags and the end-tags, respectively.}
\]

\[
\text{in any direction and the output tree is generated in the top-down manner from the root to the leaves. During a transformation step, one or more new computation branches may be spawned and the input control is moved to the input nodes selected by the XPath expressions. At the same time, the output heads may generate a new part of the output tree.}
\]

The transducer is equipped with a finite set of variables to store string values. Let \( X \) be a set of variables and \( \Gamma \) be a set of symbols for the variable values. The values of the variables are expressed by a value-function \( \mathcal{E}_{val} : X \to \Gamma^* \). Modification of the variable values is expressed by an expression-function \( \mathcal{E}_{exp} : X \to XPath(\Sigma, X) \). The computation of the new value for a variable \( x \in X \) is specified by an XPath expression \( \theta = \mathcal{E}_{exp}(x) \). Evaluation of \( \theta \) gives us the resulting string value. Formally, the GXT is an 8-tuple

\[
T = (Q, \Sigma, \Delta, \Gamma, q_0, \mathcal{E}_{val_0}, R, X)
\]

where \( Q \) is a finite set of states, \( \Sigma \) is an input alphabet, \( \Delta \) is an output alphabet, \( \Gamma, X \) are as described above, \( q_0 \in Q \) is the designated initial state, \( \mathcal{E}_{val_0} \) is the initial value-function, and \( R \) is a finite set of rules of the form

\[
(q, \sigma) \to c[(q_1, \theta_1, \mathcal{E}_{exp_1}), \ldots, (q_n, \theta_n, \mathcal{E}_{exp_n})].
\]

In the left-hand side (lhs), \( q \) is the current state and \( \sigma \in \Sigma \) is an input name. In the right-hand side (rhs), \( c \) is an XML tree over \( \Delta \) that represents the part of the output tree (output fragment) to be generated. It is indexed by \( n \) rule calls - triples of the form \( (q_i, \theta_i, \mathcal{E}_{exp_i}) \) where the XPath expression \( \theta_i \) specifies nodes to be processed subsequently in the state \( q_i \), and the expression-function \( \mathcal{E}_{exp_i} \) defines new values for the variables during this processing. Application of such rule spawns \( n \) new computation branches. In case \( n = 0 \), the current computation branch is terminated. The GXT contains exactly one rule for each possible lhs.

The transformation induced by GXT \( T \) is the function \( \mu_T : T_\Sigma \to T_\Delta \) such that \( \mu_T(t_{in}) = t_{out} \) if and only if \( t_{out} \in T_\Delta \) is generated by a computation of \( T \) starting at the initial configuration (with respect to \( t_{in} \)) of the form \( (q_0, 1, \mathcal{E}_{val_0}) \) and terminating in the final configuration \( t_{out} \).

C. Streaming XML transducer

The streaming XML transducer (SXT) is a stream-to-stream transducer. It has a single input head that

\[
\text{Fig. 1. A schema of the formal framework.}
\]
reads the input stream sequentially, and a single output head that generates the output stream sequentially. The sequential reading of input XML stream $s$ can be alternatively viewed as preorder traversal of XML tree $t$ such that $\text{stream}(t) = s$. Each input node is visited exactly twice during a single pass - once when moving top-down, and once when moving bottom-up. The first visit corresponds to reading an start-tag, and the second visit represents reading of an end-tag. The SXT is equipped with a memory buffer. Formally, the SXT is a 7-tuple

$$T = (Q, \Sigma, \Delta, \Gamma, q_0, val_0, R)$$

where $Q$ is a set of states, $\Sigma$ is an input alphabet, $\Delta$ is an output alphabet, $\Gamma$ is a finite set of buffer symbols, $q_0 \in Q$ is the designated initial state, $val_0 \in \Gamma^+$ is the initial (finite) content of the buffer, $R$ is a set of rules. The rules of SXT are of the form

$$(q, \sigma, \tau, \tau_a, z) \rightarrow s(q', a_{in}, a_{out})$$

In the lhs, $q$ is the current state, $\sigma \in \Sigma$ is an input name, $\tau, \tau_a \in \{1, \} \]$ is a type of the current tag and the following tag (lookahead tag), respectively, and $z$ is the current buffer symbol. In the rhs, $q'$ is a new state, $s$ is a constant string that represents the part of the output to be generated, $a_{in}$ is an action of the input head: (1) $\odot$: no move, (2) $\sim$: preorder move; and $a_{out}$ is a buffer write operation: (1) $\wedge$: write $a$ and move left, (2) $\wedge^{-}$: write $a$ and move right, or (3) $a$: write $a$ and do not move.

The configuration of SXT with respect to the input XML stream $\text{stream}(t_{in})$ is of the form

$$s_{out}(q, u, \tau, val, ptr),$$

where $s_{out}$ is the output XML stream generated so far, $q$ is the current state, $u$ is the current input node of $t_{in}$, $\tau$ is the type of the current tag, $val \in \Gamma^+$ is the current content of the buffer and $ptr \in \mathbb{N}$ is the pointer representing the current position within the buffer. The transformation induced by $T$ is the function

$$\mu_T : \text{stream}(T_\Sigma) \rightarrow \text{stream}(T_\Delta)$$

such that $\mu_T(s_{in}) = s_{out}$ if and only if $s_{out} \in \text{stream}(T_\Delta)$ is generated by computation of $T$ starting at the initial configuration (with respect to $s_{in}$) of the form $(q_0, 1, \text{start}, val_0)$ and terminating in the final configuration $s_{out}(q, 1, \text{end}, val)$. 

III. Stack-based simulation

We present a particular streaming simulation designed within our framework. We restrict the GXT to capture simple top-down XML transformations only. We show that, under two further input-dependent conditions, the restricted GXT can be simulated by an SXT with stack size proportional to the height of the input tree. The stack-based simulation is efficient - the buffer-size proportional to the length of the branches is needed to evaluate top-down XPath expressions in the branches of the XML tree.

A. Restricted models

**Simple GXT (SGXT)** contains only top-down XPath expressions and it does not use variables. The location step is now of the form $(\text{child}|\text{descendant})::\text{ElementName}$

Formally, the simple GXT is a 5-tuple $T = (Q, \Sigma, \Delta, q_0, R)$ where $R$ contains rules of the form

$$(q, \sigma) \rightarrow c[(q_1, \theta_1), \ldots, (q_n, \theta_n)]$$

where $\theta_i$ is a top-down XPath expression. All other symbols have the same meaning as in the full GXT.

**Simple SXT (SSXT)** differs from the SXT in the rule definition. The rules are now of the form

$$(q, \sigma, \tau, \tau_a, z) \rightarrow s(q', a_{in}, \gamma)$$

where $z \in \Gamma$ is the current top stack symbol, and $\gamma \in \Gamma^*$ is a sequence of stack symbols to be put to the stack. All other symbols have the same meaning as in the SXT.

**Input-dependent conditions.** Let $r : (q, \sigma) \rightarrow c[(q_1, \theta_1), \ldots, (q_n, \theta_n)]$ be an SGXT rule. We define an auxiliary function for evaluating all XPath expressions of $r$ in XML tree $t$ at context node $u \in V_t$ by:

$$\text{eval-rhs}(r, t, u) = \text{eval}(\theta_1, t, u) \ldots \text{eval}(\theta_n, t, u).$$

The conditions are then defined as follows.

1. The SGXT is order-preserving if and only if, for each of its rules, the input nodes returned by the XPath expressions are in preorder for any input tree $t$ and $u \in V_t$:

$$\text{eval-rhs}(r, t, u) \text{ is a subsequence of preorder}(t).$$

2. The SGXT is branch-disjoint if and only if, for each of its rules, the input nodes returned by the XPath expressions are disjoint for any input tree $t$ and $u \in V_t$:

$$v_1, v_2 \in \text{eval-rhs}(r, t, u) \text{ implies } v_1 \text{ is not a prefix of } v_2 \text{ nor vice versa.}$$

Intuitively, if any of the conditions is not satisfied, it may happen that a part of the input tree disproportional to the height of the input tree must be stored in the memory and a simulation by the SSXT is not possible.

B. Basic principles

Let $T$ be the simulated SGXT and $T'$ the simulating SSXT. The simulation proceeds in cycles.

**Simulation cycles.** During a cycle, a single transformation step of $T$ is simulated, called the current transformation step. A cycle is driven by the cycle configuration that consists of the following items:

1. **current context node** - the current input node of $T$ during the current transformation step,
2. **current rule** - the rule of $T$ applied during the current transformation step,
3. **matched rule call** - a rule call of the current rule.
The matched rule call represents the leftmost\textsuperscript{3} rule call, for which a match has been already found. At the beginning of the simulation, the current context node is the root node of the input tree, the current rule is the rule of \(T\) of the form \((q_0, \sigma) \rightarrow \text{rhs}\) where \(q_0\) is the initial state of \(T\) and \(\sigma\) is the name of the root of the input tree. The matched rule call is set to the initial value 0.

A cycle proceeds as follows: The input head traverses the subtree at the current context node in preorder, and at the same time it evaluates all selecting expressions in the rule calls of the current rule. The evaluation is accomplished by means of finite automata\textsuperscript{4}. We distinguish three cases depending on the result of the evaluation:

1. A matching node is found for one rule call which is positioned to the right of the matched rule call or it equals the matched rule call.

The input head of \(T'\) is moving downwards, and a new rule call of the current rule is matched and “recursively” processed. The output head generates a part of the output fragment of the current rule delimited by the matched rule call and the newly-matched rule call. Then the current cycle configuration is stored in the stack. The matched node becomes the new current context node. The rule of the form \((q', \sigma') \rightarrow \text{rhs}\) where \(q'\) is the state in the newly-matched rule call and \(\sigma'\) is the name of the matched node becomes the new current rule, and the left sentinel rule call becomes the new current rule call. A new cycle starts driven by the new cycle configuration.

2. A matching node is found for two or more rule calls, or a matching node is found for a rule call that is positioned to the left of the matched rule call.

This situation occurs in case \(T\) is non-order-preserving and an error is reported.

3. No matching node is found and the whole subtree at the current context node has been traversed.

The control moves back to the processing of the rule containing this rule call. We denote the current rule by \(r\). The last part of the output fragment of \(r\) is generated. The top stack configuration becomes the new cycle configuration, and the new cycle starts.

C. Algorithm

We outline the stack-based algorithm. The algorithm uses three statements for controlling the simulating SSXT:

- “push \(x\)” - pushes \(x\) - either initial stack symbol \(z_0\), sequence of FSA states, or pair (sequence of FSA states, cycle configuration) - on the top of the stack,
- “pop” - removes one symbol from the top of the stack,
- “advance” - advances to the next tag in the input stream,
- “error” - reports an error.

The cycle configuration consists of the current rule and the matched rule call identified by unique numbers. The current context node is stored implicitly\textsuperscript{5}. The algorithm uses following functions:

- \textit{top} - returns the symbol on the top of the stack,
- \textit{label} - returns the label of the current tag,
- \textit{tag} - returns the type of the current tag,
- \textit{la_tag} - returns the type of the lookahead tag.

1: set \textit{current rule} to \((q_0, \text{label}) \rightarrow \text{rhs}\);
2: set \textit{matched rule call} to 0;
3: push initial stack symbol \(z_0\); \{Iterates over all nodes of the input tree in preorder\}
4: \textbf{while} \textit{top} is not initial stack symbol \(z_0\) or \textit{tag} is not end \textbf{tag} \textbf{do}
5: \textbf{if} \textit{top} is a sequence \(S\) of FSA states \textbf{then}
6: \textbf{if} \textit{tag} is start tag \textbf{then}
7: let \(S'\) be a sequence of FSA states obtained after transition from \(S\) on symbol \textit{label};
8: \textbf{if} \(S'\) contains no final state \textbf{then}
9: push sequence of FSA states \(S'\); \{Match found, generating phase of entering cycle starts\}
10: \textbf{else if} \(S'\) contains final state for single rule call \(\text{rc} \geq \text{matched rule call}\) \textbf{then}
11: generate fragment part of \textit{current rule} between \textit{matched rule call} and \text{rc};
12: push pair \((\textit{current rule}, \text{rc})\);
13: set \textit{current rule} to \((q, \text{label}) \rightarrow \text{rhs}\) where \(q\) is the state of \text{rc};
14: \textbf{else}
15: \textbf{error};
16: \textbf{end if}
17: \textbf{else if} \textit{tag} is end tag and \textit{la-tag} is end tag \textbf{then}
18: \textbf{pop}
19: \textbf{end if}
20: \textbf{else if} \textit{top} is a pair (sequence of DFA states \(S\), cycle configuration \((r, rc))\) \textbf{then}
21: if \textit{tag} is start tag \textbf{then}
22: push sequence of initial states of FSAs for \textit{current rule}; \{No more matches, generating phase of returning cycle starts\}
23: \textbf{else if} \textit{tag} is end tag \textbf{then}
24: generate fragment part of \textit{current rule} between \textit{matched rule call} and the end of the output fragment;
25: set \textit{current rule} to \(r\);
26: set \textit{matched rule call} to \(rc\);
27: \textbf{pop};
28: \textbf{if} \textit{la-tag} is start tag \textbf{then}
29: push sequence of FSA states \(S\);
30: \textbf{end if}
31: \textbf{end if}
32: \textbf{else if} \textit{top} is initial stack symbol \(z_0\) \textbf{then}
33: \textbf{if} \textit{tag} is start tag \textbf{then}
34: push sequence of initial states of FSAs for \textit{current rule};
35: \textbf{end if}
36: \textbf{end if}
37: advance;
38: \textbf{end while}

\textsuperscript{3}With respect to the preorder of the rhs of the rule.

\textsuperscript{4}See [3] for a more detailed description of the method.

\textsuperscript{5}The SSXT stores as many configurations of FSAs in the stack as is the number of levels between the current context node and the actual position of the input head.
IV. AUTOMATIC XSLT AND XQUERY PROCESSOR

We describe an automatic streaming processor for the languages XSLT and XQuery that is based on the framework introduced. First we show that GXT is indeed closely related to these languages - we identify expressive subsets of XSLT and XQuery languages that can be directly translated to GXT. The translation methods ignore several constructs of the languages. However, since both XSLT and XQuery provide rich set of constructs with overlapping functionality, the methods could be possibly extended to capture more constructs. We consider an input alphabet Σ, output alphabet Δ and a finite set of variables \( X = \{ x_1, \ldots, x_m \} \).

A. Translation of XSLT stylesheet

The GXT models the XSLT transformations driven by the structure of the input document. An XSLT stylesheet \( xsl \) convertible to GXT consists of (1) an initializing template and (2) several rule templates. The initializing template sets the current mode to the initial state of the GXT and initializes the GXT variables using the XSLT parameters.

\[
\text{xsl:template match="/"}
\]

\[
\begin{align*}
&\quad \text{xsl:apply-templates select="child::*" mode="q0">} \\
&\quad \quad \ldots \\
&\quad \quad \text{xsl:with-param name="x1"/>} \\
&\quad \quad \text{xsl:with-param name="xm"/>} \\
&\quad \text{</xsl:apply-templates>}
\end{align*}
\]

Each rule template is of the following form.

\[
\text{xsl:template match="name" mode="q">}
\]

\[
\begin{align*}
&\quad \text{xsl:with-param name="x1"/>} \\
&\quad \ldots \\
&\quad \text{xsl:with-param name="xm"/>} \\
&\quad \text{</xsl:template>}
\end{align*}
\]

It is translated to a GXT rule \( r \) is of the form \( (q, name) \rightarrow rhs \). The rhs is created by translation of template body which contains a list of parameter declarations followed by a sequence of (possibly nested) output elements and \text{apply-templates} constructs. An output element named \text{name} is specified directly as a pair of tags:

\[
\text{name> ... element content ... </name>}
\]

The \text{apply-templates} construct has a select attribute that contains selecting expression, and a mode attribute that represents a state of the resulting GXT. It contains a list of with-param constructs that sets values for all parameters initialized in the initializing template. Each value is defined by means of a selecting expression in the select attribute.

\[
\begin{align*}
&\quad \text{xsl:apply-templates select="selexp" mode="q'">} \\
&\quad \quad \text{xsl:with-param name="x1" select="exp1"/>} \\
&\quad \quad \text{xsl:with-param name="xm" select="expm"/>} \\
&\quad \text{</xsl:apply-templates>}
\end{align*}
\]

Each \text{apply-templates} construct is translated to the rule call \( (q', selexp, f_{exp}) \) where \( f_{exp} \) maps \( x_i \) to the selecting expression \( exp_i \). The rhs of the rule \( r \) is created from the template body so that each output element corresponds to a single node of \( rhs \) and each \text{apply-templates} construct corresponds to a single rule call of \( rhs \). The structure of \( rhs \) is determined by nesting of the output elements and \text{apply-templates} constructs.

B. Translation of XQuery query

The GXT models the XQuery queries consisting of a simple recursive function. An XQuery query \text{xq} convertible to GXT consists of (1) an initializing expression, and (2) a processing function \text{local:process}. The initializing expression contains a \text{let} clause where the root node is stored in the variable \( \$root \) and the variables \( \$x_1, \ldots, \$x_m \) are initialized to empty strings. A return clause follows where the function \text{local:process} is called with the initial settings - the initial state \( q_0 \), the root node \( \$root \), and the variables with initial values.

\[
\begin{align*}
&\quad \text{let \$root := /child::*, \$x1 := "", \ldots, \$xm := ""} \\
&\quad \text{return local:process("q0", \$root, \$x1, \ldots, \$xm)}
\end{align*}
\]

The processing function is recursive and it captures the processing of all GXT rules. It has \( m + 2 \) parameters: a state, a node to be processed, and current values of the variables.

\[
\text{declare function local:process (}
\]

\[
\text{\quad \$state as xs:string, \$node as node(),}
\]

\[
\text{\quad \$x1 as xs:string, \ldots, \$xm as xs:string )}
\]

\[
\{ \quad \text{... function body ...} \}
\]

The function body contains several conditional branches - exactly one for each necessary element name (except output elements) and each value of the \$state parameter appearing in \text{xq}. The condition consists of a state test and an element name test.

\[
\begin{align*}
&\quad \text{if (}$\$state = "q") \text{ then} \\
&\quad \quad \text{if (name($\$node) = "name") then (...branch content ...)}
\end{align*}
\]

\[
\text{else ()}
\]

Each conditional branch can be directly translated to a single rule of GXT, let denote it \( r \). For the case above, we obtain a rule of the form \( (q, name) \rightarrow rhs \). The branch content is then translated to \( rhs \). It contains (possibly nested) output elements and simple FLR expressions. An output element named \text{name} is specified by a pair of tags, it may contain other output elements or simple FLR expressions.

\[
\text{name> ... element content ... </name>}
\]

The simple FLR expression contains a \text{for} clause, a \text{let} clause, and a \text{return} clause. In the \text{for} clause, the nodes to be processed subsequently are computed by means of the selecting expression \text{selexp}. In the \text{let} clause, the new values for the variables are computed using the selecting expressions \( exp_1, \ldots, exp_m \). Finally, in the \text{return} clause, the processing function is called recursively with new parameters - the state \( q' \), the node \$node', and new values of the variables.
for $node$ in $node/selexp$
let $x1 := exp1, ..., $xm := expm
return local:process("q'", $node', $x1, ..., $xm),

Each FLR expression is translated into the rule call
\[(q', selexp, e_{exp})\] where \(e_{exp}\) maps each \(x_i\) to the selecting
expression \(exp_i\). The rhs of the rule \(r\) is created from the
branch content so that each output element corresponds to
a single node of \(rhs\) and each FLR expression corresponds to
a single rule call of \(rhs\). The structure of \(rhs\) is
determined by nesting of the output elements and FLR
expressions in the branch content.

C. Design of automatic streaming processor

The streaming processor consists of the following basic
components (see also Fig. 2):

1. Analyzer. The analyzer examines the constructs in the
input XSLT stylesheet or XQuery query (both the XPath
constructs and the constructs specific for the transformation
language). It checks whether a subclass of XML transform-
ations is specified which captures all the constructs
encountered. In case more such subclasses are found, the
smallest one is chosen and the XSLT stylesheet/XQuery
query is passed to the corresponding converter. In case no
such subclass is found, the transformation is passed to the
classical tree-based processor.

2. A set of converters. A converter is associated with
each transformation subclass. The transformation subclass
corresponds to some restricted GXT. The converter then
creates an object of the restricted SXT associated with
this GXT. The object creation is automatic, following the
simulation algorithm provided for the restricted GXT. The
streaming processing of an input tree \(t\) is realized by calling
the method \(transform(t)\) of the new SXT object.

D. Implementation and evaluation

We have implemented the framework introduced and the
stack-based algorithm for the XSLT language in order to
be used in the semantic repository Trisolda [4]. Experimen-
tal results show that it requires a memory proportional
to the depth of the input XML documents which can be
considered as a constant for real-world XML documents
(99% of documents have fewer than 8 levels whereas the
average depth is 4 according to [11]).

V. Conclusion

We introduced a formal framework for the purpose of
analyzing space complexity of streaming processing of
XML transformations. The framework is based on
tree transducers which enables us to specify streaming
algorithms with various complexity formally, and clearly
characterize transformation classes captured. We consider
only XML documents without data values in order to
keep the models employed comprehensible. However, the
framework can be easily extended to handle data values
as entities (see [6]).

We designed implemented a stack-based streaming algo-
rithm in order to demonstrate usage of the new framework.
The theoretical results are supported by the experimental
results and show that the algorithm is indeed low memory-
consuming when processing transformations from associ-
ated transformation class.

In the future work, we intend to design streaming algo-
rithms for larger transformation subclasses, in particular
the whole class of simple top-down XML transformations.

Acknowledgment

This work was partially supported by the Ministry of
Education of the Czech Republic (grant MSM0021620838)
and by the grant VEGA 1/3196/06. The work contains
results from a PhD thesis of Comenius University in
Bratislava, Slovakia.

References

[1] F. Bry, F. Coskun, S. Durmaz, T. Furché, D. Olteanu, and
M. Spannagel. The XML Stream Query Processor SPEX.
Query Processor for XML Streams. In Proceedings of ICDE
Path sharing and predicate evaluation for high-performance
2003.
Web Repository And Interfaces. In International Conference
on Advances in Semantic Processing SEMAPRO 2007. IEEE
Transformations Based on Tree Transducers. In Proceedings of IDC
Streaming XSLT Transformations. In Proceedings of SOFSEM
The BEA/XQRL Streaming XQuery Processor. In Proceedings of VLDB
[9] Z. Guo, M. Li, X. Wang, and A. Zhou. Scalable XSLT Evalu-
ation. In Advanced Web Technologies and Applications, LNCS
A Transducer-Based XML Query Processor. In Proceedings of VLDB
Real XML Data Collections. In COMAD’06: Proc. of the 15th Int.
Conf. on Management of Data, pages 20–31, 2006.