Automatic Streaming Processing of XSLT Transformations Based on Tree Transducers

Jana Dvořáková
Department of Computer Science
Faculty of Mathematics, Physics and Informatics
Comenius University, Bratislava, Slovakia
E-mail: dvorakova@dcs.fmph.uniba.sk

Keywords: XML transformation, XSLT, streaming processing, tree transducer

Received: March 15, 2008

Streaming processing of XML transformations is practically needed especially when large XML documents or XML data streams are to be transformed. In this paper, the design of an automatic streaming processor for XSLT transformations is presented. Unlike other similar systems, our processor guarantees bounds on the resource usage for the processing of a particular type of transformation. This feature is achieved by employing tree transducers as the underlying formal base. The processor includes a set of streaming algorithms, each of them is associated with a tree transducer with specific resource usage (memory, number of passes), and thus captures different transformation subclass. The input XSLT stylesheet is analyzed in order to identify the transformation subclass to which it belongs. Then the lowest resource-consuming streaming algorithm capturing this subclass is applied.

Povzetek: Obravnavano je avtomatično pretakanje XSLT transformacij.

1 Introduction

XML (28) is a meta-language defined by W3 Consortium in order to store structured data. The initial W3C recommendation for XML was published in 1998 and since then XML has become a popular format. It is commonly used for data exchange among applications since it enables them to add semantics to data explicitly. Furthermore, XML is a suitable tool in every field where it is necessary to create document standards. XML usage is still growing fast and new technologies for processing XML documents are emerging.

Transformations of XML documents are needed in many situations. For instance, let us consider two applications exchanging data in XML format, each of them requiring different structure for the same content. Then a transformation must be performed while the data are being transferred between these applications. A typical XML transformation processor reads the whole input document into memory and then performs particular transformation steps according to the specification. References to any part of the input document are processed in a straightforward way by traversing the in-memory representation, and the extracted parts are combined to form the required output fragment. This approach is called tree-based processing of XML transformations. In early days of XML, this kind of processing was sufficient since the existing XML documents were small and stored in files. However, nowadays it is quite common to encounter extensive XML data (e.g., database exports) or XML data streams in practice. In both cases, the tree-based processing is not suitable - in the former case it is not acceptable or even possible to store the whole input document in the memory, while in the later one the XML data become available stepwise and need to be processed "on the fly".

In our case, the research on efficient processing of XML transformations was in part motivated by processing large XML data in the semantic repository Trisolda (9). The repository contains semantic annotation for various web resources. A standard format for specify such semantics is Resource Description Framework (RDF) which is an instance of XML. Since the amount of web resources annotated tends to grow very fast, the transformation of RDF data into other representations/views and vice versa cannot be performed in the tree-based manner. At the same time, it is not suitable to write the transformations by hand using a SAX parser. The transformations needed are not trivial, especially due to the complexity of RDF format, and along with adding new functions into Trisolda repository new transformations may become necessary. Therefore, a more flexible approach is employed and a processor is proposed such that the most efficient strategy for performing a given transformation is automatically chosen.

A natural alternative to the classical tree-based processing of XML transformations is the streaming (event-driven) processing. Here the input document is read sequentially, possibly in several passes; and the output document is generated sequentially in one pass. Only a part of the input document is available at a time, and thus advanced techniques must be used to process references to the input document and connect the extracted parts to the proper position within the output document.

Currently, the most frequently used XML transformation languages are XSLT (27) and XQuery (29), both general (Turing-complete) languages intended for tree-based processing. There is a great interest to identify XSLT and XQuery transformations which allow efficient streaming processing. When designing an XSLT/XQuery streaming processor, the key task is to find the way of handling the non-streaming constructs of the languages. The streaming algorithms for XSLT and XQuery transformations are however still under development and the complexity issues such as memory requirements and the number of passes needed for specific, clearly defined transformation classes have not yet been analyzed.

The main contribution of this work is the design of a system for automatic streaming processing of XSLT transformations yielding the following properties:

– The transformation classes captured are clearly characterized. Each such class contains transformations with common properties - it represents an XSLT subset obtained by restricting constructs used in the XSLT stylesheet.

– A streaming algorithm is designed for each transformation class. The main design goal is to minimize the upper bound of memory usage, i.e., to use optimal (or close to optimal) amount of memory. Such upper bound is explicitly stated for each algorithm.

These features are achieved by employing tree transducers as the underlying formal base. Specifically, the design of the processor is based on the formal framework for XML transformations introduced in (10). In this paper, the framework is simplified and customized in order to facilitate the implementation. It contains a general model – an abstract model of general, tree-based transformation languages, and a set of streaming models that differ in the kind of memory used and the number of passes over the input allowed. Each streaming model can simulate some restricted general model. The framework contains a simulation algorithm for each such pair *streaming model → restricted general model*. The framework is abstract, and thus can be used to develop automatic streaming processors for other general transformation languages as well (e.g., XQuery).

The implementation level of the framework for XSLT language includes the implementation of streaming models and two modules: (1) an analyzer that associates the input XSLT stylesheet with the lowest resource-consuming streaming model that is able to process it, and (2) the translator that automatically converts the XSLT stylesheet into the streaming model chosen according to the associated simulation algorithm. The processor based on the framework is easily extensible since new transducers and algorithms may be specified and implemented, as well as optimizable since the current algorithms may be replaced by the more efficient ones. Although there are some XML transformations such that their streaming processing is always high resource-consuming (e.g., complete reordering of element children), most of the practical transformations can be processed with reasonable bounds on the resource usage and thus, more effectively than when processed in the tree-based manner.

The rest of this paper is organized as follows: Section 2 contains description of both approaches to processing XML transformations and the complexity measures related to streaming processing. In Section 3, the customized formal framework for XML transformations is introduced and the underlying tree transducers are described. An example algorithm designed within the framework is presented in Section 4. In Section 5, the design of our automatic streaming processor for XSLT transformations is introduced. In Section 6, the relation to other work is discussed. Section 7 briefly introduces the implementation of the example streaming algorithm and Section 8 concludes with summary and comments on future work.

2 Complexity of streaming processing

In this section, the relevant complexity measures for the streaming algorithms for XML transformations are specified.

An XML document contains the following basic constructs: elements, element attributes, and text values. The document may be represented as a tree that is obtained by a natural one-to-one mapping between elements and internal nodes of the tree. The text values appear in the leaves of such tree. Reading a document in document order then exactly corresponds to the preorder traversal of the constructed tree.

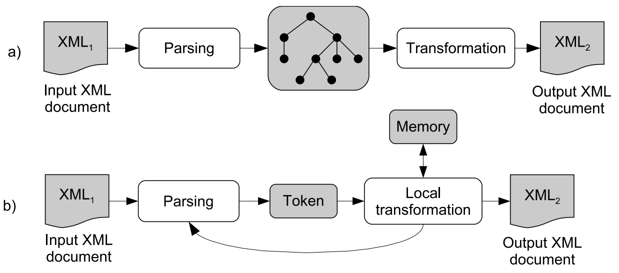

Figure 1: Two types of XML transformation processing: (**a**) tree-based processing, (**b**) streaming processing.

The tree-based processing of XML transformations (Fig. 1a) is flexible in the sense that the input document is stored in the memory as a tree and can be traversed in any direction. On the contrary, during the streaming processing (Fig. 1b) the elements of the input document become available stepwise in document order and similarly the output elements are generated in document order. The actual context is restricted to a single input node. Clearly, one-pass streaming processor without an additional memory is able to perform only simple transformations, such as renaming elements and attributes, changing attribute values, filtering. It must be extended to perform more complex restructuring. The common extensions are (1) allowing more passes over the input document, (2) adding an additional memory for storing temporary data. The extensions can be combined[1]. We obtain the corresponding complexity measures for streaming processing of XML transformations:

1. the number of the passes over the input tree,
2. the memory size.

It is reasonable to consider the complexity of the streaming processing in relation to the tree-based processing. As mentioned in Section 1, all XML transformations can be expressed in both XSLT and XQuery, and processed by their tree-based processors. Various transformation subclasses can be then characterized by putting restrictions on these general transformation languages, typically by excluding certain constructs.

When designing streaming algorithms, we have a choice regarding three settings – the type of the memory used (none, stack, buffers for storing XML fragments), and the values of the two complexity measures mentioned. Streaming algorithms with different settings may capture different transformation subclasses. Since the transformation subclasses are characterized as some subsets of the general transformation language considered, the key issue in the algorithms is to realize a streaming simulation of the non-streaming constructs included in the restricted language (see Fig. 2).

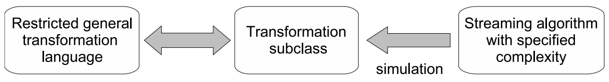
Restricted general transformation language
Transformation subclass
simulation
Streaming algorithm with specified complexity

Figure 2: The streaming simulation of subsets of a general transformation language.

We use tree transducers to design the streaming algorithms formally and to model transformation subclasses. They are included in the formal framework for streaming XML transformations that are described in the next section.

3 Formal framework

The framework is intended as a formal base for automatic streaming processors of the general transformation languages. It does not cover all XML transformations. In order to keep the models employed simple and comprehensible, it is restricted to model primarily the transformations that capture the relevant problems of streaming processing. In Section 5, a way how some of the restrictions on the transformation set can be overcome in the implementation is described.

The framework consists of the following formal models:

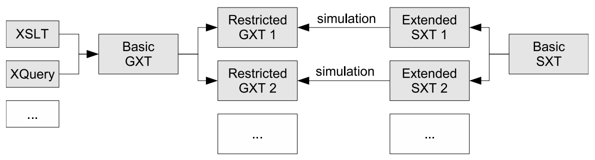

Figure 3: A schema of the formal framework.

1. a basic general model for tree-based processing of XML transformations and its restrictions,
2. a basic streaming model for streaming processing of XML transformations and its extensions.

The design of both models results from an analysis of various tree transformation models, XML transformation models as well as real-world XML transformation languages and systems. They are based on tree transducers, models for tree transformations (25) originated in the formal language theory. We introduce two novel models – a *general XML transducer (GXT)* used as the general model, and a *streaming XML transducer (SXT)* used as the streaming model. They are defined in common terms in order to facilitate development of the simulation algorithms.

The overall schema of the framework is shown in Fig. 3. The basic SXT represents a simple one-pass streaming model without an additional memory. Following the ideas from Section 2, it can be extended by memory for storing temporary data and by allowing more passes over the input document. The basic GXT represents the most powerful general model. As already mentioned, it does not capture all XML transformations, but only a subset significant in the case of streaming processing.

For each extended SXT, the transformation subclass captured is identified by imposing various restrictions on the basic GXT. The inclusion is proved by providing an algorithm for simulating this restricted GXT by the given extended SXT.

3.1 Notions and Notations

XML Document Abstraction. In what follows, element attributes and data values are not considered[2]. Let Σ be an alphabet of element names. The set of *XML trees* over Σ is denoted by $\mathcal{T}_\Sigma$, the empty XML tree is denoted by ε. An *indexed XML tree* may in addition have some leaves labeled by symbols from a given set X. A set of XML trees over Σ indexed by X is denoted by $\mathcal{T}_\Sigma(X)$. In the *rightmost indexed XML tree*, the element of the indexing set occurs only as the rightmost leaf. The set of rightmost indexed XML trees is denoted by $\mathcal{T}_\Sigma(X)_r$.

A particular XML tree $t \in \mathcal{T}_\Sigma(X)$ is uniquely specified as a triple (V_t, E_t, λ_t) where V_t is a set of nodes, $E_t \subseteq V_t \times V_t$ is a set of edges, and $\lambda_t : V_t \to \Sigma \cup X$ is a labeling function.

[1]More passes over the input tree are not possible for XML data streams that must be processed "on the fly".

[2]We refer the reader to (10) for the definition of the extended framework including both element attributes and data values.

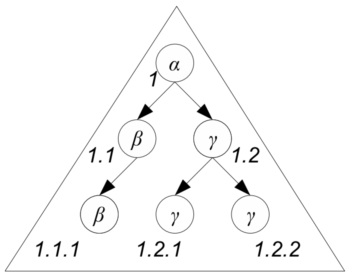

Figure 4: An example of the XML tree.

Example 3.1. An XML tree $t = (V_t, E_t, \lambda_t)$ over the alphabet $\Sigma = \{\alpha, \beta, \gamma\}$ and the empty indexing set $X = \emptyset$ is shown in Fig. 4. The nodes of t are uniquely identified by dynamic level numbering. The sets V_t, E_t and the labeling function λ_t are defined as follows:

$$\begin{array}{lcl} V_t & = & \{1, 1.1, 1.2, 1.1.1, 1.2.1, 1.2.2\}, \\ \lambda_t(1) = \alpha, & & \lambda_t(1.1.1) = \beta, \\ \lambda_t(1.1) = \beta, & & \lambda_t(1.2.1) = \gamma, \\ \lambda_t(1.2) = \gamma, & & \lambda_t(1.2.2) = \gamma. \end{array}$$

Selecting Expressions. Simple *selecting expressions*, derived from XPath expressions (26), are used to locate the nodes within the XML tree. The selecting expression is a *path* consisting of a sequence of *steps*. It can be either absolute (starting with /), or relative. The *step* consists of two components – an axis specifier $axis$ and a predicate $pred$. They are specified as outlined below. Comparing to the XPath language, the set of expressions is restricted and the syntax of some constructs is simplified – the meaning is explained in parentheses. The semantics of the selecting expressions follows the semantics of the equivalent XPath expressions.

$step$: $axis\,[pred]$
$axis$: $\times$ (self),
$\downarrow$ (child), $\downarrow*$ (descendant),
$\uparrow$ (parent), $\uparrow*$ (ancestor),
$\leftarrow$ (left sibling), $\overset{*}{\leftarrow}$ (preceding),
$\rightarrow$ (right sibling), $\overset{*}{\rightarrow}$ (following)

$pred$:	$*$	(select all elements)
	$name$	(select the elements named $name$)
	i	(select the element on i-th position within siblings)
	$step$	(select the elements having context specified by $step$)

The names of the elements are taken from an alphabet Σ. The set of selecting expressions over Σ is denoted by $\mathcal{S}_\Sigma$. The evaluation of a selecting expression in the context of some XML tree t and one of its nodes $u \in V_t$ returns the same set of nodes of t as the evaluation of the corresponding XPath expression. Note that the context set contains a single node only.

3.2 XML Transducers

General XML Transducer (Fig. 5a)**.** The input heads of GXT traverse the input tree in any direction and the output is generated from the root to the leaves. At the beginning of a transformation, the transducer has only one input head, which aims at the root of the input tree, and one output head, which aims at the root position of the empty output tree. During a single transformation step, the whole input tree is available as a context. One or more new computation branches can be spawned and the corresponding input control is moved to the input nodes specified by selecting expressions. At the same time, the output heads may generate a new part of the output.

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

– Q is a finite set of states,

– Σ is an input alphabet,

– Δ is an output alphabet,

– $q_0 \in Q$ is an initial state, and

– R is a set of rules of the form

$$Q \times \Sigma \rightarrow \mathcal{T}_\Delta(Q \times \mathcal{S}_\Sigma)\ .$$

For each $q \in Q$ and $\sigma \in \Sigma$, there is exactly one rhs such that $(q, \sigma) \rightarrow rhs \in Q$.

The right-hand side of a rule contains an XML tree over the output alphabet indexed by *rule calls* – pairs of the form (q, exp), where q is a state and exp is a selecting expression that returns a sequence of input nodes to be processed recursively. A simple example of a GXT transformation follows.

Example 3.2. Let $T = (Q, \Sigma, \Sigma, q_0, R)$ be a GXT where $Q = \{q_0\}$, $\Sigma = \{\alpha, \beta, \gamma\}$. and R consists of the rules

$$(q_0, \alpha) \rightarrow \varepsilon\ , \qquad (3.1)$$
$$(q_0, \beta) \rightarrow \alpha((q_0, \downarrow[*]))\ , \qquad (3.2)$$
$$(q_0, \gamma) \rightarrow \gamma((q_0, \downarrow[2]), (q_0, \downarrow[1]))\ . \qquad (3.3)$$

The transducer processes the input trees over alphabet Σ. The subtrees at nodes named α are completely removed (rule 3.1), the nodes named β are renamed and get a new name α (rule 3.2), and at last, when encountering a node named γ, the first two children are processed in reversed order (rule 3.3).

The GXT is inspired mainly by the tree-walking tree transducer (TWR) (4) and data tree transducer (DTT) (23). It works on unranked trees, but does not handle data values. Similarly to TWR, the computation is high-level and based on rule calls. However, the XPath language is used for pattern matching on the paths of the input tree as it is in DTT. This choice is natural since XPath is used in the common general transformation languages (XSLT, XQuery). The GXT is tree-walking, i.e., the input tree is traversed in any direction. It allows more computation branches, but it is still sequential model in the sense that during a transformation step only a single rule call is processed.

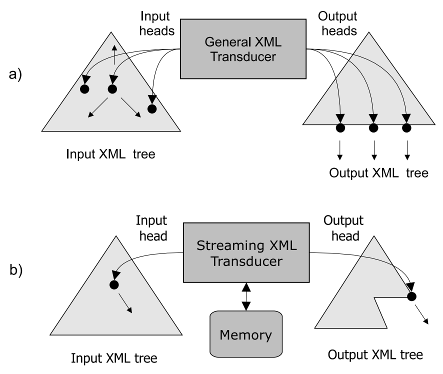

Figure 5: The processing model of the transducers: (**a**) the GXT; (**b**) the SXT.

Streaming XML Transducer (Fig. 5b)**.** The SXT has a single input head that traverses the input tree in preorder, and a single output head that generates the output tree in preorder. Each node is visited twice during a single pass – once when moving top–down, and once when moving bottom–up. Thus, two types of SXT states are recognized (1) the states indicating the first visit of nodes and (2) the states indicating the second visit of nodes. During a single transformation step, the input head either moves one step in preorder or stays at the current position. At the same time, an output action is performed, depending on the type of rule applied. When applying a *generating rule*, a new part of the output is connected to the current position of the output head, and then the output head moves to the position under the rightmost leaf of the new part. When applying a *closing rule*, no output is generated, only the output head is moved one step upwards in preorder within the output tree.

Formally, the streaming XML transducer (SXT) is a 5-tuple $T = (Q, \Sigma, \Delta, q_0, R)$, where

- $Q = Q_1 \cup Q_2, Q_1 \cap Q_2 = \emptyset$ is a finite set of states,
- Σ, Δ are the same as in the case of GXT,
- $q_0 \in Q_1$ is the initial state, and
- $R = R_g \cup R_c, R_g \cap R_c = \emptyset$ is a finite set of rules of the form:

$$\begin{aligned} R_g : Q \times \Sigma \times Pos &\rightarrow \mathcal{T}_\Delta(Q \times \mathcal{S}_\Sigma)_r \\ R_c : Q \times \Sigma \times Pos &\rightarrow Q \times \mathcal{S}_\Sigma \end{aligned}$$

  where $Pos = \{leaf, no\text{-}leaf\} \times \{last,\ no\text{-}last\}$[3]. For each $q \in Q$ and $\sigma \in \Sigma$ there is at most one rhs such that for each $pos \in Pos$ there is a rule $(q, \sigma, pos) \rightarrow rhs \in R$[4]. Furthermore, for each $(q, \sigma, pos) \rightarrow rhs \in R$, $rec(rhs) = (q', exp)$[5], one of the following preorder conditions holds:

  1. *moving downwards*: $q \in Q_1$, and
     – $pos[1] = no\text{-}leaf$, $q' \in Q_1$, $exp = \downarrow[1]$, or
     – $pos[1] = leaf$, $q' \in Q_2$, $exp = \times[*]$,
  2. *moving upwards*: $q \in Q_2$, and
     – $pos[2] = no\text{-}last$, $q' \in Q_1$, $exp = \rightarrow[1]$, or
     – $pos[2] = last$, $q' \in Q_2$, $exp = \uparrow[*]$,
  3. *no input move*: q, q' are of the same kind, $exp = \times$.

The left-hand side of a rule consists of a state, an element name and a node position. The position is used to determine the preorder move within the input tree and it consists of two predicates – the first one indicating a leaf node, and the second one indicating a last node among the siblings. The right-hand side is an XML tree rightmost indexed by a rule call.

4 An example algorithm

In this section, a particular streaming simulation designed within our framework is presented. In particular, a top-down GXT is simulated by an SXT extended with stack of the size proportional to the height of the input tree.

The stack-based simulation is efficient - in order to evaluate simple top-down selecting expressions in the branches of the input XML tree the memory size proportional to the length of the branches, which equals height of the input tree, is needed. However, it is shown that a restriction to stack is sufficient. First, the models considered are described formally.

Restricted GXT. The restricted GXT, called the top-down GXT (TGXT) differs from GXT in the rule definition - R is a set of rules of the form

$$Q \times \Sigma \rightarrow \mathcal{T}_\Delta(Q \times top\text{-}\mathcal{S}_\Sigma)$$

where $top\text{-}\mathcal{S}_\Sigma$ is a set of simple top-down selecting expressions. It is a subset of selecting expressions such that only top-down axis ($child$ and $descendant$) and name predicates ($[name]$) are allowed. The simulated TGXT must in addition satisfy two input-dependent conditions:

1. The TGXT is *order-preserving* if and only if, for each of its rules, the input nodes returned by the selecting expressions in the rhs are in preorder for arbitrary input tree t and $u \in V_t$.
2. The TGXT is *branch-disjoint* if and only if, for each of its rules, the input nodes returned by the selecting expressions in the rhs are disjoint for arbitrary input tree t and $u \in V_t$.

[3]If $pos \in Pos$ is a node position, its first component is referred by $pos[1]$ and to its second component is referred by $pos[2]$.

[4]This condition is necessary to keep the model deterministic.

[5]If rhs is a particular right-hand side, its rule call is referred by $rec(rhs)$.

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 thus the stack-based simulation is not applicable.

Extended SXT. The extended SXT, called the stack-based SXT (SSXT) is a 7-tuple $T = (Q, \Sigma, \Delta, \Gamma, q_0, z_0, R)$ where

- Q, Σ, Δ, q_0 are the same as in the case of SXT,
- Γ is a finite set of stack symbols,
- $z_0 \in \Gamma$ is the initial stack symbol, and
- R is a finite set of rules of the form:

$$\begin{aligned} R_g : Q \times \Sigma \times Pos \times \Gamma \quad &\to \\ &\mathcal{T}_\Delta(Q \times \mathcal{S}_\Sigma \times \Gamma^*)_r \\ R_c : Q \times \Sigma \times Pos \times \Gamma \quad &\to \quad Q \times \mathcal{S}_\Sigma \times \Gamma^* \end{aligned}$$

The lhs now contains, in addition, the current top stack symbol, and the rhs contains a sequence of stack symbols to be put on the top of the stack. All other symbols have the same meaning as in the SXT.

4.1 Construction of Simulating SSXT

The formal proposition follows. It says that, for each order-preserving and branch-disjoint TGXT, it is possible to construct an SSXT inducing equivalent translations.

Proposition 4.1. *Let $T = (Q, \Sigma, \Delta, q_0, R)$ be an order-preserving and branch-disjoint TGXT. Then an SSXT T' exists such that, for each $t_{in} \in \mathcal{T}_\Sigma$ and $t_{out} \in \mathcal{T}_\Delta$, if T translates t_{in} to t_{out} then T' translates t_{in} to t_{out}.*

The simulation proceeds in cycles. During a cycle, a single transformation step of T is simulated, called the *current transformation step*. Such simulation consists of several transformation steps of T'. A cycle is driven by the *cycle configuration* that consists of three 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.

During the whole simulation, the matched rule call represents the leftmost[6] 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 $(q_0, \sigma) \to rhs$ where q_0 is the initial state of T and σ is the name of the root of the input tree. The matched rule call is the left sentinel rule call which is a virtual rule call positioned to the left from all other rule calls. This special rule call is used to initialize a new cycle. In case no error is encountered, a cycle includes two phases - an *evaluation phase* and a *generation phase*.

Phase Alternation. During the evaluation phase the input head of T' 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 in a standard way by means of finite automata[7]. Three cases are distinguished depending on the result of the evaluation phase:

1. *A matching node is found for exactly one rule call, and this rule call (newly-matched rule call) is either positioned to the right of the matched rule call or it equals the matched rule call.*

This type of cycle is called an *entering cycle* since it takes place when the input head of T' is moving downwards, and a new rule call of the current rule is matched and "recursively" processed. The generation phase follows: The output head generates a specific part of the output fragment of the current rule. The part is a set of nodes that appear between the matched rule call and the newly-matched rule call. After the generation, the current cycle configuration is stored in the stack. The matched node becomes the new current context node. The rule of the form $(q', \sigma') \to rhs$ where q' is the state in the newly-matched rule call and σ' 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.*

This type of cycle is called a *returning cycle* since it takes place when the input head of T' is moving upwards, the processing of some rule call is finished, and the control moves back to the processing of the rule containing this rule call. The current rule is denoted by r. The generation phase follows directly: 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.

5 Design of XSLT streaming processor

An automatic streaming processor for XSLT transformations is described which is based on the framework intro-

[6]The positions of rule calls are always considered with respect to the preorder of the rhs of the rule.

[7]This method was, for example, presented in the Y-Filter algorithm (5; 8).

duced. The models within the framework are abstract, and thus the framework provides means to develop efficient streaming algorithms for XML transformation subclasses at abstract level, and to adapt them to an arbitrary general transformation language. First, the general issues regarding the framework implementation are described, and then an adaptation for the XSLT transformation language is discussed in more detail.

5.1 Framework Restrictions

As mentioned in the previous section, the formal framework is restricted in several ways. Some of the restrictions can be easily overcome in the implementation, while others require more complex handling.

1. *Restrictions on the XML document.* Attributes and data values are associated with elements. They can be easily added to the implementation – if such construct needs to be processed, it is accessed using the same path like the parent element. On the other hand, if the construct needs to be generated in the output, the action is performed together with the generation of the parent element.

2. *Restrictions on the selecting expressions.* The simple selecting expressions used capture the typical problems that arise during the streaming location of the nodes in XML document (context references in predicates, backward axis). Other constructs must be handled separately – however, the techniques used for constructs included in our restricted set may be often exploited. Moreover, there has been already carried on a research on the streaming processing of large subsets of XPath language (see Section 6 for overview).

3. *Restrictions on the general transformation language.* A part of the restrictions in GXT results from the restrictions on selecting expressions, and others are caused by excluding certain general transformation constructs, such as loops, variables, functions. However, the GXT models transformations that reorder the nodes within an XML tree with respect to the document order, which is probably the most important issue in streaming processing of XML transformations if the specific issues concerning selecting expression evaluation are not considered.

5.2 Adaptation for XSLT

Let us now describe the design of the prototype XSLT streaming processor. The GXT represents an abstract model for general transformation languages. Since our intention is to adapt the framework for the XSLT language, it does not need to be implemented directly. Instead, we are looking for a correspondence between restricted GXTs and XSLT subsets. The GXT models the XSLT transformations driven by the structure of the input document. Thus, each XSLT stylesheet consisting of a list of simple templates activated by structure and mode can be directly converted to GXT. The matching element of such simple template is referenced by a name only and the body of the template may contain several output elements (possibly nested) and calls for applying another templates. The template is called by a selecting expression and a mode.

Specifically, 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.

```
<xsl:template match="/">
   <xsl:apply-templates
         select="child::*" mode="q0"/>
</xsl:template>
```

Each *rule template* can be directly translated to a single rule of GXT. It is of the following form.

```
<xsl:template match="name" mode="q">
    ...  template body ...
</xsl:template>
```

The resulting GXT rule r is of the form $(q, name) \to rhs$ Thus, the left-hand side consists of the element name in the `match` attribute and the state in the `mode` attribute. The rhs is created by translation of the template body as described below.

The template body contains a sequence of (possibly nested) output elements and `apply-templates` constructs. An output element named $name$ is specified directly as a pair of tags (alternatively, the `element` construct might be used):

```
<name>
    ...element content ...
</name>
```

The `apply-templates` construct has a `select` attribute that contains a selecting expression, and a `mode` attribute that represents a state of the resulting GXT.

```
<xsl:apply-templates
         select="selexp" mode="q'"/>
```

Each `apply-templates` construct can be translated to a single rule call. For the case above, a rule call of the form $(q', selexp)$ is obtained.

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 `apply-templates` construct corresponds to a single rule call of rhs. The structure of rhs is determined by nesting of the output elements and `apply-templates` constructs in the template body. The resulting GXT is of the form $T = (Q, \Sigma, \Delta, q_0, R)$ where

– Q contains the modes appearing in xsl,

– R contains the rules created by translation from particular rule templates as described above. Moreover, R contains rule of the form

$$(q, \sigma) \to (q, \texttt{child}[*])$$

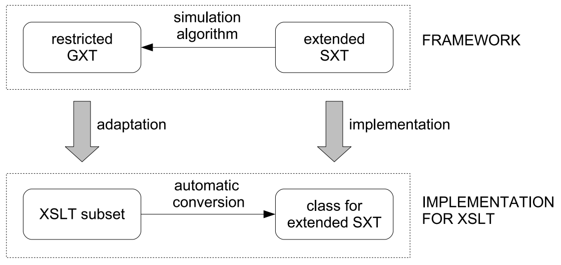

Figure 6: An implementation of the framework for XSLT language.

for each mode q and name $\sigma \in \Sigma$ such that xsl does not contain a template matching σ in the mode q. Such additional rules correspond to the XSLT implicit built-in rule templates[8].

In a similar way, XSLT subsets corresponding to restricted GXTs can be identified. According to the principle of the formal framework, a restricted GXT (GXT_r) can be simulated by some extended SXT (SXT_e) such that the simulation algorithm is known. Then XSLT stylesheets from the XSLT subset associated with GXT_r can be converted to SXT_e using the simulation algorithm. The conversion can be performed automatically since the simulation algorithm exactly determines how to convert constructs of the given XSLT subset into the rules of SXT_e. The resulting SXT_e is constructed explicitly as an object and its method $transform()$ performs streaming processing of the transformation specified by the stylesheet. The relation between the formal framework and the implementation for XSLT is shown in Fig. 6.

5.3 Modules of Streaming Processor

To sum up, the streaming processor works in three steps (see also Fig. 7):

1. *Analysis.* The analyzer examines the constructs in the input XSLT stylesheet (both XPath constructs and XSLT constructs themselves). It checks whether there is specified an XSLT subset that allows all the constructs encountered. If there are more such subsets, the smallest one is chosen.

2. *Translation.* The translator creates an object for the extended SXT associated with the XSLT subset chosen. The creation is automatic, following the simulation algorithm provided for the XSLT subset.

3. *Processing.* The method $transform()$ of the new SXT object is run on the input XML document. The streaming transformation performed is equivalent to the one specified by the input XSLT stylesheet.

[8]The built-in XSLT rules actually ensure that the resulting GXT is complete. Note that it is also deterministic since xsl cannot contain two templates matching the same name in the same mode by definition of the valid XSLT stylesheet.

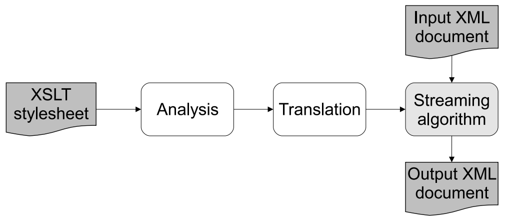

Figure 7: Modules os the automatic streaming processor.

6 Related work

Most of the earlier work was devoted to analyzing the streaming processing of the querying language XPath (1; 2; 6; 7; 14; 16; 22; 24). Recently, several streaming processors for the transformation languages XQuery and XSLT have appeared.

XML Streaming Machine (XSM) (19) processes a subset of XQuery on XML streams without attributes and recursive structures. It is based on a model called XML streaming transducer. The processor have been tested on XML documents of various sizes against a simple query. Using XSM the processing time grows linearly with the document size, while in the case of standard XQuery processors the time grows superlinearly. However, more complex queries have not been tested.

BEA/XQRL (12) is a streaming processor that implements full XQuery. The processor was compared with Xalan-J XSLT processor on the set of 25 transformations and another test was carried on XMark Benchmarks. BEA processor was fast on small input documents, however, the processing of large documents was slower since the optimizations specially designed for XML streams are limited in this engine.

FluXQuery (17) is a streaming XQuery processor based on a new internal query language *FluX* which extends XQuery with constructs for streaming processing. XQuery query is converted into FluX and the memory size is optimized by examining the query as well as the input DTD. FluXQuery supports a subset of XQuery. The engine was benchmarked against XQuery processors Galax and AnonX on selected queries of the XMark benchmark. The results show that FluXQuery consumes less memory and runtime.

SPM (Streaming Processing Model) (15) is a simple one-pass streaming XSLT processor without an additional memory. Authors present a procedure that tries to converts a given XSLT stylesheet into SPM. However, no algorithm for testing the streamability of XSLT is introduced, and thus the class of XSLT transformations captured by SPM is not clearly characterized.

The effectiveness of the processors mentioned was examined only through empirical tests. The test results show that streaming processors tend indeed to be less time and space consuming than tree-based processors. However, since no formal characterizations of the transformation class captured were given, the results hold only for a few (typically one or two) XML transformations chosen for experiments.

In other approaches (3; 13; 21), a new specification language is developed which supports streaming processing, and the streaming processor for this new language is designed. In all cases the connection to the commonly used transformation languages is not clearly stated and the computational complexity of the streaming processing is not addressed.

7 Implementation

The formal framework introduced has been implemented on .Net platform. The pilot implementation includes the stack-based algorithm described in Section 4. The evaluation of the algorithm implementation shows that it is highly efficient in practice - it requires memory proportional to the depth of the input XML document. Since this depth is generally not depending on the document size and common XML documents are relatively shallow (99% of XML documents have fewer than 8 levels whereas the average depth is 4 according to (20)), the memory requirements for most of the XML documents are constant, independent to the document size. On the contrary, standard XSLT processors are tree-based and thus require memory proportional to the document size. We refer the reader to (11) for a more detailed description of the stack-based algorithm implementation and evaluation.

8 Conclusion

A design of an automatic streaming processor for XSLT transformations have been presented. Comparing to other similar processors, the contribution of our approach is that the resource usage for streaming processing of particular types of XSLT transformations is known. Our processor includes several streaming algorithms, and it automatically chooses the most efficient one for a given XSLT stylesheet. The process of choice has a solid formal base – a framework consisting of tree transducers that serve as models both for the streaming algorithms and for the transformation types.

In the future work, we plan to include algorithms for the local and non-order-preserving transformations to obtain a processor for a a large subset of practically needed XML transformations. We intend to demonstrate the usage of such processor by integration into the Trisolda semantic repository and carry out performance tests and comparison to other implementations subsequently.

Acknowledgement

This work was supported in part by the grant VEGA 1/3106/06.

References

[1] Z. Bar-Yossef, M. Fontoura, and V. Josifovski. On the Memory Requirements of XPath Evaluation over XML Streams. In *PODS '04: Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 177–188, New York, NY, USA, 2004. ACM.

[2] C. Barton, P. Charles, D. Goyal, M. Raghavchari, M. Fontoura, and V. Josifovski. An Algorithm for Streaming XPath Processing With Forward and Backward Axis. In *Proceedings of ICDE 2003*, 2003.

[3] O. Becker. Transforming XML on the Fly. In *Proceedings of XML Europe 2003*, 2003.

[4] G. J. Bex, S. Maneth, and F. Neven. A Formal Model for an Expressive Fragment of XSLT. *Inf. Syst.*, 27(1):21–39, 2002.

[5] N. Bruno, L. Gravano, N. Koudas, and D. Srivastava. XML & Data Streams. In *Stream Data Management*, pages 59–82. Springer Verlag, 2005.

[6] F. Bry, F. Coskun, S. Durmaz, T. Furche, D. Olteanu, and M. Spannagel. The XML Stream Query Processor SPEX. In *Proceedings of ICDE 2005*, pages 1120–1121, 2005.

[7] Y. Chen, S. B. Davidson, and Y. Zheng. An Efficient XPath Query Processor for XML Streams. In *Proceedings of ICDE 2006*, pages 79–79, 2006.

[8] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path Sharing and Predicate Evaluation for High-performance XML Filtering. *ACM Trans. Database Syst.*, 28(4):467–516, 2003.

[9] J. Dokulil, J. Tykal, J. Yaghob, and F. Zavoral. Semantic Web Repository And Interfaces. In *International Conference on Advances in Semantic Processing SEMAPRO 2007*. IEEE Computer Society, 2007.

[10] J. Dvořáková and B. Rovan. A Transducer-Based Framework for Streaming XML Transformations. In *Proceedings of SOFSEM (2) 2007*, pages 50–60, 2007.

[11] J. Dvořáková and F. Zavoral. An Implementation Framework for Efficient XSLT Processing. In *Proceedings of IDC 2008, Studies in Computational Intelligence*, Springer-Verlag, 2008.

[12] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey, A. Sundararajan, and G. Agrawal. The BEA/XQRL Streaming XQuery Processor. In *Proceedings of VLDB 2003*, pages 997–1008, 2003.

[13] A. Frisch and K. Nakano. Streaming XML Transformations Using Term Rewriting. In *Proceedings of PLAN-X 2007*, 2007.

[14] P. Genevès and K. Rose. Compiling XPath for Streaming Access Policy. In *Proceedings of ACM DOCENG 2005*, pages 52–54, 2005.

[15] Z. Guo, M. Li, X. Wang, and A. Zhou. Scalable XSLT Evaluation. In *Advanced Web Technologies and Applications, LNCS 3007/2004*. Springer Berlin / Heidelberg, 2004.

[16] K. Jittrawong and R. K. Wong. Optimizing XPath Queries on Streaming XML Data. In *ADC '07: Proceedings of the eighteenth conference on Australasian database*, pages 73–82, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc.

[17] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. FluXQuery: An Optimizing XQuery Processor for Streaming XML Data. In *VLDB'2004: Proceedings of the Thirtieth International Conference on Very Large Databases*, pages 1309–1312, 2004.

[18] X. Li and G. Agrawal. Efficient Evaluation of XQuery over Streaming Data. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 265–276. VLDB Endowment, 2005.

[19] B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou. A Transducer-Based XML Query Processor. In *Proceedings of VLDB 2002*, pages 227–238, 2002.

[20] I. Mlýnková, K. Toman and J. Pokorný. Statistical Analysis of Real XML Data Collections. In *COMAD'06: Proc. of the 13th Int. Conf. on Management of Data*, pages 20–31, 2006.

[21] K. Nakano. An Implementation Scheme for XML Transformation Languages through Derivation of Stream Processors. In *Proceedings of the Second ASIAN Symposium on Programming Languages and Systems (APLAS'04)*, 2004.

[22] D. Oltenau, H. Meuss, T. Furche, and F. Bry. XPath: Looking Forward. In *Proceedings of XMLDM Workshop*, pages 109–127, 2002.

[23] T. Pankowski. Transformation of XML Data Using an Unranked Tree Transducer. In *EC-Web*, pages 259–269, 2003.

[24] F. Peng and S. S. Chawathe. XPath Queries on Streaming Data. In *Proceedings of ACM SIGMOD 2003*, 2003.

[25] J. Thatcher. Tree automata: An informal survey. In A. V. Aho, editor, *Currents in the Theory of Computing*, chapter 4, pages 143–172. Prentice-Hall, 1973.

[26] W3C. *XML Path Language (XPath), version 1.0, W3C Recommendation*, 1999. `http://www.w3.org/TR/xpath`.

[27] W3C. *XSL Transformations (XSLT) Version 1.0, W3C Recommendation*, 1999. `http://www.w3.org/TR/xslt`.

[28] W3C. *Extensible Markup Language (XML) 1.0 (Fourth Edition), W3C Recommendation*, 2006. `http://www.w3.org/TR/REC-xml`.

[29] W3C. *XQuery 1.0: An XML Query Language, W3C Recommendation*, 2007. `http://www.w3.org/TR/xquery`.