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
dvorakova@dcs.fmph.uniba.sk

Summary. Streaming processing of XML transformations is practically needed especially if we have large XML documents or XML data streams as the transformation input. In this paper, we present the design of an automatic streaming processor of transformations specified in XSLT language. 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.

1 Introduction

A typical XML transformation processor (e.g., processors for the popular transformation languages XSLT and XQuery) is tree-based, i.e., it 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 a required output fragment. The output document may be constructed either in the memory or sequentially as a data stream.

This approach has been sufficient for most of XML documents. However, now there appear more and more XML documents that require specific handling, such as large XML documents and XML data streams. A natural solution is to employ streaming processing, when the input document is read as a stream, possibly in several passes; and the output document is generated as a stream in one pass. Only a part of the input document may be accessed 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 [10] and XQuery [9], both general (Turing-complete) languages intended for tree-based processing. There is great interest in the identification of XSLT and XQuery transformations that allow efficient streaming processing. The key issue in the design of a streaming processor is to find the way of handling the non-streaming constructs of the languages. The goal of this work is to propose a system that performs automatic streaming processing of XSLT transformations, so that the upper bounds on the resource usage for particular transformation subclasses are guaranteed. This feature is achieved by employing tree transducers as the base of the processor. More precisely, the processor represents a demonstration implementation of the formal framework for XML transformations introduced in [4]. 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 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* $\rightarrow$ *restricted general model*. The framework is abstract, and thus can be used to develop automatic streaming processors for other general transformation languages as well.

The implementation 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.

Related Work. Much of the previous work is devoted to streaming evaluation of XPath expressions, e.g. [1, 2]. Besides, several automatic streaming processors for XSLT and XQuery have been implemented. XSLT processor based on SPM (Streaming Processing Model) [3] is a simple one-head, one-pass transformer without an additional memory. The conversion of XSLT stylesheet to a streaming algorithm is well-described, however, only a small subset of XSLT is captured. Existing processors for XQuery language [5, 6] are equipped with memory buffers. Therefore they are able to handle large subsets of XQuery, but the algorithms employed are not provided with a complexity analysis, and therefore the resource requirements for processing a particular type of transformation are not known. Empirical studies presented in [3, 5, 6] show that the streaming processors introduced tend to be less resource-consuming than the tree-based processors, but the results hold only for ad-hoc transformations chosen for the experiments. To our best knowledge, our work is the first attempt to design an automatic streaming processor for a large subset of a general XML transformation language that guarantees specific resource usage for a given transformation class.

2 Complexity of Streaming Processing

In this section, we specify the relevant complexity measures for the streaming algorithms for XML transformations.

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

The tree-based processing of XML transformations 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 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, and
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 Sect. 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

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

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.

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 we describe 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, we restrict it to model mainly the transformations that capture the relevant problems of streaming processing. In Sect. 4, we mention how some of the restrictions on the transformation set can be overcome in the implementation.

The framework consists of the following formal models:

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.

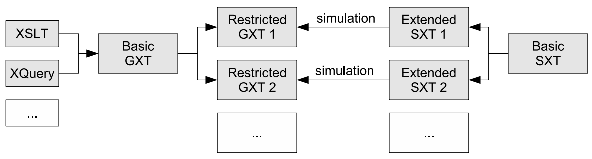

Fig. 1. A schema of the formal framework

Both models are based on tree transducers, models for tree transformations [7] 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. 1. The basic SXT represents a simple one-pass streaming model without an additional memory. Following the ideas from Sect. 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.

XML Document Abstraction. In what follows, we do not consider element attributes and data values[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 $\mathcal{T}_\Sigma(X)_r$.

Selecting Expressions. We use simple *selecting expressions* derived from XPath expressions [8] 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 – *axis* and *predicate*. 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 – we explain the meaning in parentheses. The semantics of the selecting expressions follows the semantics of the equivalent XPath expressions.

step :	*axis* [*predicate*]	
axis :	$\times$ (self), $\downarrow$ (child), $\downarrow *$ (descendant), $\uparrow$ (parent), $\uparrow *$ (ancestor), $\leftarrow$ (left sibling), $\overset{*}{\leftarrow}$ (preceding), $\rightarrow$ (right sibling), $\overset{*}{\rightarrow}$ (following)	
predicate :	$*$	(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. We denote the set of selecting expressions over Σ by $\mathcal{S}_\Sigma$.

General XML Transducer (Fig. 2a)**.** 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.

[2] We refer the reader to [4] for the definition of the extended framework including both element attributes and data values.

Formally, the GXT is a 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 *recursive 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 1. 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

$$(q_0, \alpha) \rightarrow \varepsilon , \tag{1}$$
$$(q_0, \beta) \rightarrow \alpha((q_0, \downarrow[*])) , \tag{2}$$
$$(q_0, \gamma) \rightarrow \gamma((q_0, \downarrow[2]), (q_0, \downarrow[1])) . \tag{3}$$

The transducer processes the input trees over alphabet Σ. The subtrees at nodes named α are completely removed (rule 1), the nodes named β are renamed and get a new name α (rule 2), and at last, when encountering a node named γ, the first two children are processed in reversed order (rule 3).

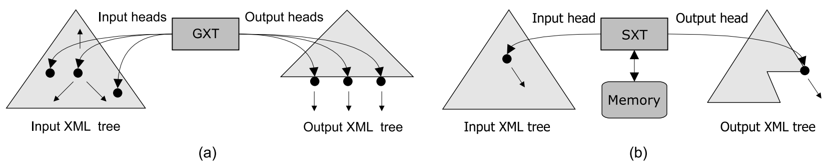

Fig. 2. The processing model of the transducers: (**a**) the GXT; (**b**) the SXT

Streaming XML Transducer (Fig. 2b)**.** 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, we recognize two types of SXT states (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 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 as above, $q_0 \in Q_1$ 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:

$$R_g : Q \times \Sigma \times Pos \rightarrow \mathcal{T}_\Delta(Q \times \mathcal{S}_\Sigma)_r \ , \quad R_c : Q \times \Sigma \times Pos \rightarrow Q \times \mathcal{S}_\Sigma \ ,$$

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 $(q, \sigma, pos) \rightarrow rhs \in R$. Furthermore, for each $(q, \sigma, pos) \rightarrow rhs \in R$, $rec(rhs) = (q', exp)$[4], 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 recursive call.

In [4], it is demonstrated how to design streaming algorithms within the original framework. Namely, there are identified restrictions that must be put on the GXT in order to make possible the simulation by the one-pass SXT using stack size proportional to the height of the input tree. As the result, the simulation algorithm for the *local and order-preserving* transformations is presented. Since in this paper we focus on the implementation of the framework, we do not mention particular simulation algorithms.

4 Design of XSLT Streaming Processor

We describe an automatic streaming processor for XSLT transformations based on the framework introduced. 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

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

[4] If rhs is a particular right-hand side, we refer to its recursive call by $rec(rhs)$.

adapt them to an arbitrary general transformation language. We first focus on the general issues regarding framework implementation, and then we deal with an adaptation for the XSLT transformation language in more detail.

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.

- *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, its content can be retrieved using selecting expressions similar to those used for recursive processing of elements.
- *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 [1, 2].
- *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 the important issue in streaming processing of XML transformations.

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 basic GXT can be easily converted into an XSLT stylesheet:

(1) The initial XSLT template is created. Its purpose is to set the initial mode that equals to the initial state q_0 of the GXT.

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

(2) Each rule $(q, name) \to rhs$ of GXT is translated into an XSLT template:

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

The template body depends on the rhs. If it is a single recursive call (q', exp), it is mapped to the `xsl:apply-templates` instruction, where the `select` expression contains the translation of the expression *exp* into a corresponding XPath expression exp':

```
<xsl:apply-templates select="exp’" mode="q’">
```

If *rhs* contains an indexed XML tree with some element nodes, all of them must be generated as new elements within the template body. A single element named *name* is generated using the `xsl:element` instruction:

```
<xsl:element name="name">
   ... element content ...
</xsl:element>
```

The element content contains instructions for generating child elements and child recursive calls, if exist. Recursive calls are translated using the `xsl:apply-templates` instruction as mentioned above. In a similar way, each restricted

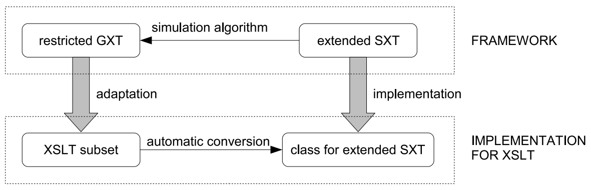

Fig. 3. An implementation of the framework for XSLT language

GXT can be translated to the corresponding XSLT subset. 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 framework and the implementation for XSLT is shown in Fig. 3.

To sum up, the streaming processor works in three steps:

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.

5 Conclusion

We have presented a design of an automatic streaming processor for XSLT transformations. 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.

We have already implemented tree transducers included within the framework, a major part of the analyzer, and the translator for processing the local and order-preserving XSLT transformations. In the future work, we plan to include algorithms for the local and non-order-preserving transformations to obtain a processor for a large subset of practically needed XML transformations. We intend to carry out performance tests and comparison to other implementations subsequently.

Acknowledgments. This work was supported in part by the grants VEGA 1/3106/06 and GUK 358/2007.

References

1. Bry F, Coskun F, Durmaz S, et al. (2005) The XML Stream Query Processor SPEX. In: ICDE 2005 1120–1121. IEEE Computer Society, Washington
2. Chen Y, Davidson S B, Zheng Y (2006) An Efficient XPath Query Processor for XML Streams. In: ICDE 2006 79. IEEE Computer Society, Washington
3. Guo Z, Li M, Wang X, Zhou A (2004) Scalable XSLT Evaluation. In: APWEB 2004, LNCS 3007/2004:190–200. Springer Berlin / Heidelberg
4. Dvořáková J, Rovan B (2007) A Transducer-Based Framework for Streaming XML Transformations. In SOFSEM (2) 50–60. Institute of Computer Science AS CR, Prague
5. Florescu D, Hillery C, Kossmann D, et al. (2003) The BEA/XQRL Streaming XQuery Processor. In: VLDB Journal 13/3 294–315. Springer-Verlag New York
6. Ludscher B, Mukhopadhyay P, Papakonstantinou Y. (2002) A Transducer-Based XML Query Processor. In: VLDB 2002 227–238. Morgan Kaufmann
7. Thatcher J W (1973) Tree Automata: An Informal Survey. Currents in the Theory of Computing 4:143–172. Prentice-Hall, Englewood Cliffs, NJ
8. W3C (1999) XML Path Language (XPath), version 1.0, W3C Recommendation. http://www.w3.org/TR/xpath
9. W3C (2007) XQuery 1.0: An XML Query Language, W3C Recommendation. http://www.w3.org/TR/xquery
10. W3C (1999) XSL Transformations (XSLT) Version 1.0, W3C Recommendation. http://www.w3.org/TR/xslt