Schema-Based Analysis of XSLT Streamability

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

Abstract—We present an algorithm which analyzes memory requirements of the streaming processing of a given XSLT stylesheet on the set of XML documents defined by a given XML schema. This schema-based analyzer represents an integral part of the Xord framework for the streaming processing of XSLT - it determines the most efficient streaming algorithm for a given transformation and the class of XML documents defined by a schema. We describe the implementation of the analyzer for the stack-based streaming algorithm. The analysis is performed using the W3C XSD format for schemas. We introduce a new compact structure for XSD representation used in the analysis.

I. INTRODUCTION

Many applications need to employ streaming approach when processing semantic data in XML format. There are several reasons for this need - on one hand, the monitoring devices, such as sensors, has become extremely cheap, and on the second hand, the need for near real-time processing of both transactional and measurement data has been growing fast in many business areas [2]. As a consequence, it is currently one of the key research challenges in the area of semantic data management to develop efficient algorithms for streaming processing of XML data, and, at the same time, to determine the resource usage of this processing.

Most typically, the languages XSLT and XQuery are used to specify XML transformations. Both of them enable the user to write a high-level specification based on tree manipulation. Common processors of these languages (e.g., Saxon, Xalan, AltovaXML) are tree-based, i.e., read the whole input document into memory and then perform the transformation itself. Such processing makes it possible to apply the tree-manipulation functions in a straightforward way.

The XSLT and XQuery tree-based processors are apparently not suitable when transforming XML streams or large XML documents. In this case, the transformation can be either written by hand using an event-base parser (e.g., SAX, StAX) or using some streaming transformation language (STX [1], XStream [6]). In both cases, writing the specification is a non-trivial task since the user must explicitly handle storing parts of the input document in the memory buffers for later processing.

In this paper we focus on the problem how to enable the user to write a tree manipulation specification in the XSLT language, and at the same time to process it in a streaming manner automatically. Such automatic streaming processor is supposed to apply the tree-manipulation functions over a continuous stream of data while the buffering is treated automatically. An important issue is to design the processor in such a way that the size of memory buffers is minimized for the given transformation and the input document.

In our previous work [4] we designed the Xord framework for the streaming processing of XSLT transformations (see Fig. 1). The framework is intended to contain several streaming algorithms for processing XSLT. The algorithms are based on formal models called streaming XML transducers. This formal base enables us to explicitly determine the class of XSLT transformations captured and the memory requirements for each algorithm. Within the framework, we have implemented and evaluated the SSXT algorithm\(^1\) which processes a subset of top-down XSLT using stack of the size proportional to the depth of the input XML document. It is represented by the class $xjSsxt$. Such processing can be considered as pure streaming processing in practice since real-world XML documents are shallow [10].

The Xord framework provides a unified interface and thus acts as an automatic streaming XSLT processor. It currently incorporates a simple analyzer which determines applicability of implemented algorithms only by a rough static analysis of the input XSLT stylesheet. The main contribution of this paper is two-fold:

- We design a much more powerful schema-based analyzer for the Xord framework. For a given XSLT stylesheet $xsl$ and an XML schema $xsd$, it automatically analyzes the memory usage of the streaming processing of $xsl$ on a

---

\(^1\)SSXT stands for simple streaming XML transducer - the underlying formal model of the algorithm.

\(^2\)We use the term XML schema for a general schema for XML documents. Only in Section V we consider a particular XML schema format W3C XSD.
set of documents defined by xsd. Based on the result of the analysis, the streaming algorithm using the memory buffers of the computed size is applied.

• We present the implementation of the analyzer for the SSXT algorithm represented by the class $XF_XsdSsst-Analyzer$. The analysis is performed using a particular representation of XML schema - W3C XSD [11], [12]. We introduce the Xord Schema Model - a new compact structure for XSD representation used in the analysis.

Related work. Several streaming processors for XSLT and XQuery have been implemented. However, their efficiency was demonstrated only by experiments on a small number of XML transformations and input XML documents. It is thus not known how much memory is consumed on clearly characterized transformation classes.

XML Streaming Machine (XSM) [9] 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. More complex queries have not been tested.

BEA/XQRL [5] 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 [8] 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) [7] is a one-pass streaming XSLT processor without an additional memory. Authors present a procedure that tries to converts a given XSLT stylesheet into SPM. No algorithm for testing the streamability of XSLT is introduced, and therefore the class of XSLT transformations captured by SPM is not clearly characterized.

II. XSLT AND XML SCHEMA REPRESENTATION

We first describe the representation of the XSLT stylesheet and the XML schema used in the schema-based analyzer. Let us note that we consider XML documents without data values throughout this paper.

A. Simple XSLT stylesheet

Simple XSLT stylesheet consists of an initializing template and several transforming templates. The initializing template sets the current mode to the initial mode $m_0$ and calls processing of the root element of the input document. It is of the form:

\[
\begin{align*}
\text{<xsl:template match="/"/>}
\text{<xsl:apply-templates mode="q0"/>}
\text{</xsl:template>}
\end{align*}
\]

The transforming templates are of the form:

\[
\begin{align*}
\text{<xsl:template match="name" mode="q">}
\text{... template body ...}
\text{</xsl:template>}
\end{align*}
\]

The template body contains output elements (possibly nested) and apply-templates calls. Output elements are of the form:

\[
\begin{align*}
\langle\text{name}\rangle...\text{element content}...\langle/\text{name}\rangle
\end{align*}
\]

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

\[
\begin{align*}
\text{<xsl:apply-templates select="selexp" mode="q'"/>}
\end{align*}
\]

A subset of XPath expression is allowed in templates - they contain child and descendant axis, and select nodes by name:

\[
\begin{align*}
\text{XPath} & := \text{Step} | \text{Step/XPath} \\
\text{Step} & := \text{child::name} | \text{descendant::name}
\end{align*}
\]

where name refers to an element name. The function for evaluating XPath expressions $exp$ at a node $u$ of an XML document $d$ is denoted by $eval-doc(exp,d,u)$. An example of simple XSLT stylesheet is depicted in Fig. 2 (the initializing template is not depicted).

\[
\begin{align*}
\text{<!-- tmp1 -->}
\text{<xsl:template match="a" mode="m0"/>}
\text{<output_a1/>}
\text{<xsl:apply-templates select="child::i/child::j/child::c" mode="m1"/>} <!-- calls tmp2 -->
\text{</output_a2/>}
\text{<xsl:apply-templates select="child::d" mode="m2"/>} <!-- calls tmp3 -->
\text{</output_a3/>}
\text{</xsl:template>}
\text{<!-- tmp2 -->}
\text{<xsl:template match="c" mode="m1"/>}
\text{<output_c/>}
\text{</xsl:template>}
\text{<!-- tmp3 -->}
\text{<xsl:template match="d" mode="m2"/>}
\text{<output_d/>}
\text{</xsl:template>}
\end{align*}
\]

Fig. 2. An example XSLT stylesheet.

The SSXT algorithm is able to process order-preserving and branch-disjoint simple XSLT transformations. Prior to defining these conditions, we define an auxiliary function $eval-exp$. Let $tmp$ be a transforming template, $d$ be an XML document, and $n$ be a node of $d$.

\[
\begin{align*}
eval-expseq-doc(tmp,d,n) = eval-doc(exp_1,d,n) \ldots eval-doc(exp_n,d,n)
\end{align*}
\]
where \( \exp_1, \ldots, \exp_n \) is a sequence of XPath expressions appearing in the template calls of \( \tmp \) (in this order). Thus, the \( \text{eval-expseq-doc} \) function returns the concatenation of the node sequences returned by individual XPath expressions. Then simple XSLT xsl is

- **order-preserving** on a set of XML documents \( D \) if and only if,
  - for each transforming template \( \tmp \) of xsl,
  - for each XML document \( d \in D \),
  - for each node \( n \) of XML document \( d \),
  it holds \( \text{eval-expseq-doc}(\tmp, d, n) \) returns a sequence of nodes of \( d \) in document order.
- **branch-disjoint** on a set of XML documents \( D \) if and only if,
  - for each transforming template \( \tmp \) of xsl,
  - for each XML document \( d \in D \),
  - for each node \( n \) of XML document \( d \),
  it holds \( \text{eval-expseq-doc}(\tmp, d, n) \) does not contain two nodes located within the same branch of \( d \).

**B. Schema tree**

We represent an XML schema hierarchically as a *schema tree*. The representation does not depend on a particular schema notation (DTD, XSD). The schema tree consists of two kinds of nodes:

- **element nodes**: correspond to an element type defined within schema
- **constructor nodes**: correspond to constructors used in the schema (sequence, choice, *, +, ?)

The relationships among element types and constructors are represented by the structure of the tree. We denote by \( D_{\text{xsd}} \) the set of XML documents defined by an XML schema \( \text{xsd} \). An example of schema tree is depicted in Fig. 3 a).

Some subtrees of schema tree may be identical - this situation occurs if we derive the schema tree from DTD or XSD containing shared element types. When designing the analyzer, the tree representation is more convenient. However in the implementation of schema-based analyzer each type is represented as a single node and the whole schema is represented as a DAG (see Section V).

In the SSXT analysis, we consider XML schemas without the choice constructor and recursive definitions. Such schema can be represented as a single regular expression. The schema represented by the tree in Fig. 3 can be alternatively written in the form

\[
a(i(bc(j(bc)+))) + c(d)\?
\]

The sequence constructor is represented simply by concatenation, the parentheses delimit particular levels of the schema tree and the repetition constructors (*, +, ?) are positioned after the corresponding closing parentheses.

**III. SSXT ALGORITHM**

The SSXT algorithm is based on a formal model called *simple streaming XML transducer* (SSXT). The transducer has a single input head that reads the input document sequentially, and a single output head that generates the output document sequentially. The SSXT is equipped with a stack to store temporary data.

The SSXT takes an input document \( d_{\text{in}} \) and a top-down XSLT stylesheet \( \text{xsl} \) as the input. It reads \( d_{\text{in}} \) sequentially in one pass and apply the stylesheet \( \text{xsl} \) stepwise. First, the template matching the root element of \( d_{\text{in}} \) in the initial mode \( m_0 \) is set to be the currently processed template (current template). The processing proceeds in cycles. During a single cycle, a single template call of the current template is processed.

**Processing cycle.** All XPath expression within a template are evaluating concurrently. The evaluation is realized by deterministic finite automata (DFA)\(^3\). A single DFA is constructed for each expression. When the processing of a template starts, the sequence of the initial states of DFAs is pushed on the stack. The input head of SSXT reads the elements of \( d_{\text{in}} \) in document order. When a start-tag is encountered, new sequence of DFAs is computed. Three situations may occur:

a) new sequence contains no final state - the input head continues in evaluation,

b) new sequence contains a single final state which belongs to the DFA evaluating the lastly-matched expression or an expression located after the lastly-matched expression - the corresponding template call is processed,

c) new sequence contains a final state which belongs to the DFA evaluating expression located before the lastly-matched expression, or it contains two or more final states - error.

In case b), the current cycle configuration (template id, matched expression id) is pushed on the stack and new

\(^3\)We refer the reader to [3] for a more detailed description of this evaluating method.
cycle for processing the called template starts. The cycle configuration is popped after the whole called template has been processed and the control moves back to the current template. In case a), the evaluation continues. Here if an end-tag is encountered, the sequence of the DFA states located at the top of the stack is popped. Hence, the XPath expression of the current template are evaluated on “branches” of \( d_{in} \).

We refer the reader to [4] for a detailed description of the algorithm.

IV. SCHEMA-BASED ANALYSIS

The schema-based analysis takes two items as the input:

- an XML schema tree \( xsd \), and
- a top-down XSLT stylesheet \( xsl \),

and it determines whether the transformation specified by \( xsl \) can be processed on the set of XML documents defined by \( xsd (D_{xsd}) \) using the SSXT algorithm. Thus, it must be checked that \( xsl \) is order-preserving and branch-disjoint on \( D_{xsd} \). The branch-disjointness is however guaranteed since we do not consider descendant axis in the XPath expressions in the analyzing algorithm. We only need to check the order-preservation. Clearly, it cannot be checked by the definition stated in Section II since it would be necessary to check possibly infinite number of XML documents from \( D_{xsd} \). Therefore we develop a method based on a static analysis of the schema tree \( xsd \). During the analysis, the templates of \( xsl \) are applied stepwise at the nodes of the schema tree. As a running example, we consider the schema tree \( xsd \) shown in Fig. 3 a) and the simple XSLT stylesheet \( xsl \) shown in Fig. 2.

Initialization. At the beginning of the analysis, the template matching the name of the schema root in the initial mode \( m_0 \) and is applied at the schema root (see Function 1).

**Function 1** \( SSXTAnalysis() : boolean \)

1: set \( tmp \) to template with head
2: set \( node \) to the root node of schema;
3: return \( AnalyzeNode(tmp, node) \);

**Template processing.** Let \( tmp \) be the current template and \( node \) the current schema node. The processing of \( tmp \) at \( node \) is accomplished by the function \( AnalyzeNode \) (see Function 2).

We call the function \( AnalyzeNode \) with arguments \( tmp_1 \) (the first template of \( xsl \)) and \( root \) (the root node of \( xsd \)). Four other functions are called inside the \( AnalyzeNode \) function:

1. **LastNames**\( (tmp) : names \)
2. **AnalyzeNode**\( (tmp, node) : boolean \)
3. **Compare**\( (regexp, names) : boolean \)
4. **ExtractFragment**\( (node, tmp) : regexp \)

1. **LastNames**\( (tmp) : names \)

The function takes a single argument: an XSLT template. Let \( es \) be a sequence of XPath expressions appearing in the template calls in the body of \( tmp \). The **LastNames** function returns a sequence of names \( names \) which appear in last steps of XPath expressions in \( es \).

**Function 2** \( AnalyzeNode(tmp, node) : boolean \)

1: if \( tmp \) does not contain XPath expressions then
2: return true;
3: end if
4: set \( names := LastNames(tmp) \);
5: set \( regexp := ExtractRegexp(node, template) \);
6: if \( regexp \) is empty then
7: return true;
8: end if
9: if \( Compare(names, regexp) \) is false then
10: report "Error when applying \((tmp, node)\)";
11: return false;
12: end if
13: for each expression \( exp_i \) in \( tmp \) do
14: for each schema node \( n \) in \( EvalExp(node, exp_i) \) do
15: let \( tmp_i \) be template called by \( exp_i \);
16: if \( AnalyzeNode(tmp_i, n) \) is false then
17: return false;
18: end if
19: end for
20: end for
21: return true;

For \( tmp_1 \) in our example XSLT stylesheet \( xsl \), we obtain:

\[
\]

(1)

\[
\text{LastNames}(es) = (c, d)
\]

(2)

since the element name \( c \) appears in the last step of \( \text{child:1/child:2/j/child:3} \) and \( d \) appears in the last step of \( \text{child:4/child:2/d} \).

2. **ExtractFragment**\( (node, tmp) : regexp \)

The function takes two arguments: a schema node \( node \) and an XSLT template \( tmp \). It returns a fragment of subtree at \( node \). Similarly as in the case of the whole schema tree, the flat representation of such fragment is a regular expression. Let again denote the sequence of XPath expressions in \( tmp \) by \( es \). The fragment consists of the element nodes which are selected by the XPath expressions in \( es \) when evaluated at \( node \). Moreover, it contains all constructors that appear on the branch from \( node \) to the selected nodes. During the evaluation of XPath expressions on the schema tree, the constructor nodes are skipped. I.e., the schema tree is considered as a tree of XML document where all repetition constructors are omitted (i.e., the repetition value is 1).

When extracting nodes selected by the XPath expressions in \( es \) computed in (1), we obtain a fragment of the subtree at the root node (i.e., fragment of the whole schema tree) shown in Fig. 3 b). The regular expressions is thus computed as follows:

\[
\text{ExtractFragment}(es, root) = ((c)^{+} + (d)?)
\]

(3)

3. **Compare**\( (regexp, names) : boolean \)

The function takes two arguments: a regular expression \( regexp \) generated by the \( \text{ExtractFragment} \) function and a sequence of element names \( names \) generated by the \( \text{LastNames} \) function. It checks whether two element names \( a, b \) exist such that

- \( a \) appears before \( b \) in \( names \), and
• a appears after b in some instance of regexp.

In case such a, b are found, the whole SSXT analysis reports fail since a violation of order-preservation in some instance of xsd has been detected and thus the SSXT algorithm cannot be applied. If we use the values of names and regexp computed in 2 and 3, respectively, we obtain

\[ \text{Compare}((c, d), ((c)^*) + (d)?) = \text{true} \]

since c cannot appear before b in any instance of ((c)*)+(d)?. The comparison is done in a linear time with respect to the length of names and regexp.

4. EvalExp(exp, node) : nodeSeq

The function takes two arguments: an XPath expression exp and a schema node node. It returns a sequence of element nodes of the schema tree (in preorder) which are selected by exp when evaluated at node. The principle of the evaluation in the schema tree is the same as in the case of the ExtractRegexp function. For the XPath expressions in es we obtain

\[
\begin{align*}
\text{EvalNodes}(\text{child::i/child::j/child::c, root}) &= n_1 \\
\text{EvalNodes}(\text{child::d, root}) &= n_2
\end{align*}
\]

The nodes n_1 and n_2 are marked explicitly in Fig. 3 b).

Corectness. The key operation in the schema-based analysis is the extraction of the schema tree fragment by the function ExtractFragment(node, tmp). The resulting regexp represents all possible sequences of element names which may be selected by the XPath expressions from tmp at schema node node. Thus, the function represents a generalization of the eval-expr-doc function mentioned in Section II. The eval-expr-doc function returns, for a given XSLT tmp tmp, XML document d, and one of its nodes n, a sequence of nodes of d selected by the XPath expressions in tmp.

We say that the regexp represents possible reading orders of the element names selected by the expressions in tmp, i.e., the order in which the elements are accessed when a document defined by the schema xsd is read sequentially. On the other hand, the LastNames function returns a sequence of element names in the order they are called in tmp. Thus, the resulting names sequence represents the processing order of the elements. In case one of the reading orders does not conform to the processing order, the order-preservation of the xsl is violated on some document from \( D_{xsd} \) according to the definition of this condition in Section II.

V. IMPLEMENTATION

The schema-based analyzer is implemented within our Xord framework on .Net platform. It is using the W3C XSD format for XML schema.

Although there are well established and widely used XML parsers, we have found no suitable parser for XSD. To perform schema manipulation, the .NET Framework provides a set of classes called the Schema Object Model, or SOM for short. The SOM is for schemas what DOM is for XML documents: the SOM classes represent various parts of a schema, for example XmlSchemaSimpleType, XmlSchemaElement, there are many other classes that represent attributes, facets, groups, complex types, and so on. This model is especially useful for creating schemas programmatically, but its application interface is not very useful for parsing and analyzing existing schemas.

Xord Schema Model. Since the schema analysis using standard XML schema DOM model would be very complicated and tangled, we have designed an Xord Schema Model which is targeted to effective representation and analysis of existing schemas. A simplified object structure of that model is depicted in Fig. 4.

The whole schema is represented as an associative array of simple or complex type nodes. Each complex node contains a list of references to its child nodes with their cardinality. Using this recursive structure that form a DAG (or a tree with one particular node selected as a root), the parsed schema could be easily traversed and processed.

Xord Template Model. The analyzer uses another set of classes of the Xord framework, an Xord Template Model. Its simplified object structure is depicted in Fig. 5.

Each template from the XSLT contains a sequence of template calls. A template call consists of the parsed XPath expression and the template called by the apply-templates mechanism.

Using these object models representing the entities used by the analyzer algorithm allows the implementation to be simple and straightforward - see Fig. 6.

Comparison implementation. The implementation of the Compare function is based on inherent properties of its arguments. Instead of an expensive checking of swapping for each pair of names, the predicate is a compound of two simple

\[
\begin{align*}
\text{Comp}(a, b) &= \text{true} \\
\text{Comp}(b, a) &= \text{false}
\end{align*}
\]

Note that in case two or more XPath expressions in tmp select the same element name, the occurrences of this name both in regexp and in names must be distinguished by the index of the corresponding selecting expression.
Fig. 5. The Xord Template Model.

```csharp
bool AnalyzeNode(XfTemplate t, XfSchema.Node n) {
    if (t.Empty )
        return true;
    XfLastNames li = t.GetLastNames();
    XfRegexp re = sch.ExtractFragment( n, t );
    if( re.Empty() )
        return true;
    if( ! sch.Compare( re, li ) )
        return false;
    foreach( XfCall call in t.calls ) {
        List<XfSchema.Node> ln =
            new List<XfSchema.Node>();
        ln = sch.EvalExp( n, call.select );
        foreach( XfSchema.Node ni in ln ) {
            AnalyzeNode( call.template, ni );
        }
    }
    return true;
}
```

Fig. 6. The code of the `AnalyzeNode` function.

steps. First, regexp is checked for existence of two distinct names within any “+” or “*” sequence. Second, the last names in names are stripped to those contained in the schema being used, adjacent duplicities are reduced to a single name, and the resulting list is linearly compared to names contained in regexp. Since each name appearing regexp must be contained in names, any difference cause a fail.

**Evaluation.** The evaluation and measurements of the SSXT algorithm implementation confirmed our expectation that it requires a memory proportional to a depth of the input XML document. Since most documents are relatively shallow, our memory requirements are independent to the document size. Even for huge documents like DBLP, the SSXT algorithm required few hundreds KB while the commonly used XSLT processors like Saxon or Xalan crashed after allocating about 1.5 GB of memory.

**VI. CONCLUSION AND FUTURE WORK**

We have presented an algorithm which, for a given XSLT transformation xsl and an XML schema xsd, analyzes memory requirements of the streaming processing of xsl on a set of XML documents defined by xsd. We have implemented the analyzing algorithm in .NET platform for a specific streaming processing using stack of the size proportional to the depth of the input XML document, i.e., the analysis determines applicability of such processing. The implementation represents an integral part of the Xord framework for streaming processing of XSLT.

Our analyzer is restricted in several aspects. First, a subset of XSLT and XML schema definitions is considered, and second, the analysis gives us only true/false answer. However, the algorithm represents a solid base for deriving more complex schema-based analyzer and more expressive streaming algorithms. If we examine particular pairs of elements for which the comparing function returns false and the possible size of their content, we may compute exact size of the memory buffers needed for processing such elements. Then it is only necessary to extend the basic stack-based streaming algorithm with such buffers and we obtain much more powerful automatic streaming XSLT processor. This is the main direction of our future research. At the same time, we intend to overcome several limitations put on XSLT and XML schema.

**Acknowledgments.** This work was partially supported by the Ministry of Education of the Czech Republic (grant MSM0021620838) and by the grant VEGA 1/3106/06. A part of the results presented comes from a PhD thesis of Comenius University in Bratislava, Slovakia.

**REFERENCES**