Determining XSLT Streamability Using New Hierarchical XSD Model

Jana Dvořáková and Filip Zavoral

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

Abstract. *We introduce a new compact model of XML Schema called a schema tree. Given an XSLT transformation xsl and an XML schema xsd, we present a method which statically analyzes the schema tree constructed according to xsd and determines whether xsl can be processed in a streaming manner on a set of XML documents defined by xsd. We consider streaming processing that uses a stack of the size proportional to the depth of the input document - this processing is highly efficient in practice since real-world XML documents are shallow. The schema analysis is performed via stepwise application of templates of xsl on the schema tree. We present the implementation of a schema tree and the static XSLT analyzer on .NET platform.*

1 Introduction

Many applications need to employ streaming approach when processing huge data in XML format. Most typically, the languages XSLT [10] and XQuery [13] 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.

The XSLT and XQuery tree-based processors are apparently not suitable when transforming XML streams or huge XML data. 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 [5]). 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.

We describe the implementation of the Xord framework which represents a prototype XSLT automatic streaming processor. The Xord framework is based on the formal framework for streaming XML transformations introduced in [3]. The framework is capable to process automatically a class of top-down XSLT transformations which captures a significant number of practically needed XML transformations. The processing is done using a stack of the size proportional to the depth of the input XML document - such processing is highly efficient in practice since real XML documents are shallow [9].

We focus especially on the schema-based analyzer which represents a powerful tool used within the Xord framework to determine the most efficient way of processing the given XSLT transformation. For a given XSLT stylesheet xsl and an XML schema xsd[1], it automatically analyzes the memory usage of the streaming processing of xsl on a set of documents defined by xsd.

The existing models for XML schemas (DOM, .NET XmlSchema) appeared inconvenient for the purpose of the streamability analysis, we therefore introduce the Xord Schema Model - a new compact model for schema representation. The model is abstract, and thus not bounded to a particular schema language. However, in the prototype implementation we employ W3C XSD notation [11, 12] for XML schemas.

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) [8] 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 [4] 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 pro-

[1] We use the term *XML schema* for a general schema for XML documents.

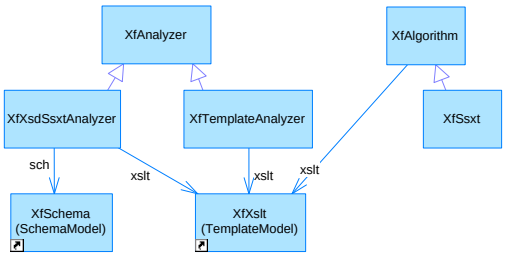

Fig. 1. The Xord framework.

cessor 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 [7] 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) [6] 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.

2 Xord Framework

The Xord framework for analyzing and transforming XML data is implemented on .NET platform. Its application interface is formed by a set of interface classes for traversing analyzed data structures. The core of the framework consists of these abstract models (see Fig. 1):

1. **Template Model** for transforming templates implemented by the *XfXslt* classes,
2. **Schema Model** for XML schemas implemented by the *XfSchema* classes,
3. **Algorithm Model** for streaming algorithms implemented by the *XfSsxt* classes,
4. **Analyzer Model** for static analyzers implemented by the *XfXsdSsxtAnalyzer* and *XfTemplateAnalyzer* classes.

Since the models are abstract, the Template Model may be adopted to model templates of any template-based XML transformation language and the Schema Model may be adopted to model any XML schema language based on structure definition.

Furthermore, the framework is complemented by a set of auxiliary helper classes. The algorithmic part of the API supports:

SsxtAlgorithm algorithm derived from the abstract Algorithm model, and

XsdSsxtAnalyzer algorithm derived from the abstract Analyzer model, and using the Schema Model and the Template Model.

The implementation of the above mentioned models are described in more detail in following sections.

3 XSLT representation

The Xord framework is currently restricted to process simple XSLT transformations on XML documents without data values.

Simple XSLT stylesheets. 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:

```
<xsl:template match="/">
    <xsl:apply-templates mode="m0"/>
</xsl:template>
```

The transforming templates are of the form:

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

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

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

The `apply-templates` construct has a `select` attribute that contains selecting expression, and a `mode` attribute.

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

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

XPath := *Step* | *Step/XPath*
Step := `child::`*name* | `descendant::`*name*

where *name* refers to an element name.

Xord Template Model. In the Xord framework, XSLT stylesheets are represented by a set of classes, an *Xord Template Model*. Its simplified object structure is depicted in Fig. 2.

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. The input template file is parsed into these structures before the analysis. Then the analysis algorithm directly traverses the DAG, evaluates the expressions etc.

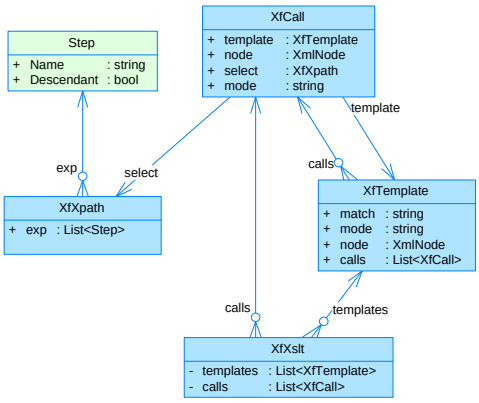

Fig. 2. The Xord Template Model.

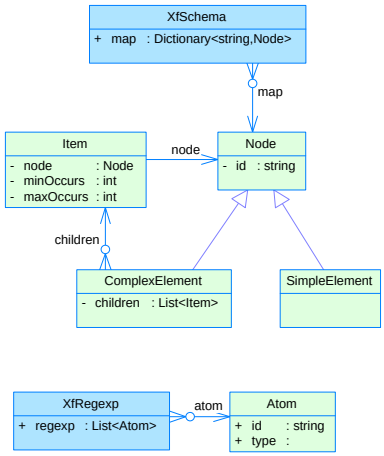

Fig. 3. The Xord Schema Model.

4 Hierarchical XML schema representation

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.

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 Schema Object Model below).

In the schema-based analysis, we consider XML schemas without the choice constructor and recursive definitions. Such schema can be represented as a single regular expression. This representation is useful in the extraction part of the analyzer algorithm (see Section 5).

Xord Schema Model. 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.

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. 3.

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.

5 Schema-based analyzer

The schema-based analyzer applies the given XSLT stylesheet xsl to the schema tree xsd, starting at the root node. First, let us remind the principles of the XSLT application to the XML document tree. Let tmp be the current template of the XSLT stylesheet (at the beginning of the transformation, it is the template matching the root element in the initial mode m_0)

1. The node sequence selected by the XPath expressions in the rule calls of the current template are found.
2. The templates called by the rule calls are applied to the selected nodes.

However, in case of the schema tree, a modification of the first step of this simple algorithm is needed:

1. *All possible node sequences* selected by the XPath expressions in the rule calls of the current template are found.

2. The templates called by the rule calls are applied to the selected nodes.

Since the set of all possible node sequences selected by XPath expressions in the first step may be infinite, we represent it in the form of regular expression $regexp$. Such regular expression is basically a fragment of the schema tree, i.e., a set of its nodes (not necessarily connected) which is a fragment of the regular expression representing the whole schema tree.

The regular expression $regexp$ may contain both element nodes and constructor nodes. It is extracted as follows: First, the node sequence selected by the XPath expressions are found in the same way as in the XML document tree (constructor nodes are skipped). Second, all constructors appearing in the branch of the schema tree from the root to the selected nodes are added to the sequence. The hierarchy of the nodes is preserved by delimiting the nodes appearing at the same level of the schema tree by parentheses.

We say that $regexp$ 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. Now let $names$ be a sequence of element names in the order they are called in tmp - clearly, the sequence can be constructed statically by examining the last steps of the XPath expressions in tmp. The $names$ 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 and the SSXT algorithm is not applicable[2]. It is thus only necessary to compare $regexp$ to the $names$ sequence in order to check applicability ot the stack-based algorithm.

Implementation. The core of the schema-based analyzer is the *AnalyzeNode* function which takes two arguments - a template of xsl and a node of the schema tree xsd. It performs the application of the template to the schema tree node as described above. Using the Template Model and the Schema Models allows the analyzer algorithm to be simple and straightforward - see Fig. 4.

The comparison of the $regexp$ to the $names$ sequence is accomplished by the $Compare$ function. Its implementation 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 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.

[2] See [3] for further details.

```
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. 4. The code of the *AnalyzeNode* function.

6 Stack-based streaming algorithm

The stack-based algorithm is based on a formal model called *simple streaming XML transducer (SSXT)*, therefore we call it the *SSXT algorithm*. 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_{in} and a top-down XSLT stylesheet xsl as the input. It reads d_{in} sequentially in one pass and apply the stylesheet xsl stepwise. First, the template matching the root element of d_{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_{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.

[3] We refer the reader to [2] for a more detailed description of this evaluating method.

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

Implementation. The implementation of the SSXT algorithm uses both Template Model and Algorithm Model classes. Since the algorithm is stack-based, the main data structure used is a polymorphic stack stk of sequences of DFA states (*SIDfaSequence*) and cycle configurations (*SICycleConfig*), see Fig. 8.

Until the transformation is finished the top of stack is checked and the stack item is processed, see the function $RunSsxt$ in Fig. 5. In case of an empty stack and nonempty remaining input new DFA sequence is pushed on the stack.

```
void RunSsxt(XfXml xml)
{
  XfTemplate currTemplate = xslt.Start();
  XfCall currCall = null;
  bool transformed = false;
  while(!transformed) {
   if(!stk.Empty()) {
    switch(stk.Type()) {
     case XfStack.ItemType.DfaSequence:
      ProcessDfaSequence();
      break;
     case XfStack.ItemType.CycleConfig:
      ProcessCycleConfig();
      break;
    }
   } else {
    switch(xml.currType) {
     case XmlNodeType.Element:
      stk.Push(new SIDfaSequence(currTemplate));
      xml.Advance();
      break;
     case XmlNodeType.EndElement:
      currTemplate.Generate(currCall, null);
      transformed = true;
      break;
    }
   }
  }
}
```

Fig. 5. The code of the SSXT algorithm.

The core of the DFA sequence processing (Fig. 6) is accomplished when start tags of elements are encountered. A new DFA sequence is generated on the stack in case the current DFA sequence contains no final states. Otherwise the output is generated and a new cycle configuration is placed on the stack. In case of a template without calls, its output is generated immediately.

```
void ProcessDfaSequence(XfXml xml)
{
  SIDfaSequence ds = stk.GetDfaSequence();
  switch(xml.currType) {
   case XmlNodeType.Element:
    SIDfaSequence new_ds = ds.Transition(xml.currName);
    if(!new_ds.HasFinalStates()) {
     stk.Push(new_ds);
     xml.Advance();
    }
    else {
     XfCall myCall = new_ds.GetCallWithFinalState();
     currTemplate.Generate(currCall, myCall);
     XfTemplate calledTemplate =
       xslt.SelectTemplate(xml.currName, myCall.mode);
     if(calledTemplate.Empty) {
      calledTemplate.Generate(null, null);
      currCall = myCall;
      if(xml.laType == XmlNodeType.Element)
       stk.Push(new_ds);
      xml.Advance();
     } else {
      stk.Push(new SICycleConfig(currTemplate,
       myCall));
      currTemplate = calledTemplate;
      currCall = null;
     }
    }
    break;
   case XmlNodeType.EndElement:
    if(xml.laType == XmlNodeType.EndElement)
     stk.Pop();
    xml.Advance();
    break;
   default:
    stk.Pop();
    break;
  }
}
```

Fig. 6. The code of the *ProcessDfaSequence* function used in the SSXT algorithm.

The cycle configuration processing (Fig. 7) depends on the current XML node type. A start tag pushes a new DFA sequence while an end tag generates output.

```
void ProcessCycleConfig(XfXml xml)
{
  SICycleConfig cc = stk.GetCycleConfig();
  switch(xml.currType) {
   case XmlNodeType.Element:
    if(xml.laType == XmlNodeType.Element)
     stk.Push(new SIDfaSequence(currTemplate));
    xml.Advance();
    break;
   case XmlNodeType.EndElement:
    currTemplate.Generate(currCall, null);
    currTemplate = cc.template;
    currCall = cc.call;
    stk.Pop();
    break;
  }
}
```

Fig. 7. The code of the *ProcessCycleConf* function used in the SSXT algorithm.

7 Evaluation

The evaluation and measurements of the SSXT algorithm implementation confirmed our expectation that it requires

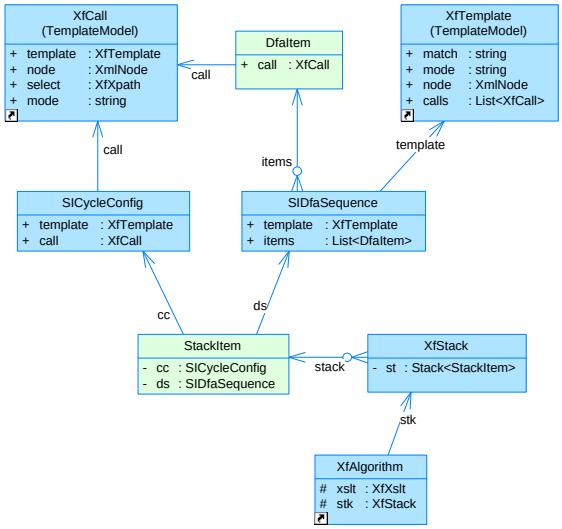

Fig. 8. The Xord SSXT Model.

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 or hanged after allocating about 1.5 GB of memory.

8 Conclusion and future work

We have presented a prototype implementation of the Xord framework which represents an automatic streaming processor for the XSLT language. It incorporates a powerful schema-based analyzer 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. The analyzer employs a special hierarchical model of XML schema called Xord Schema Model. We have implemented the Xord framework on .NET platform for a specific streaming processing using stack of the size proportional to the depth of the input XML document.

Our schema-based analyzer is restricted in several aspects - first, a subset of XSLT and XML schema definitions is considered, and second, it currently gives us only true/false answer whether the stack-based processing is applicable. However, we intend to extend it in the future research - 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.

Acknowledgments. This work was supported in part by the National programme of research (Information society project 1ET100300419).

References

1. O. Becker. Transforming XML on the Fly. In *Proceedings of XML Europe 2003*, 2003.
2. 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.
3. J. Dvořáková. Towards Analyzing Space Complexity of Streaming XML Transformations. In *The Second IEEE International Conference on Research Challenges in Information Science*. IEEE Computer Society, 2008.
4. 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.
5. A. Frisch and K. Nakano. Streaming XML Transformations Using Term Rewriting. In *Proceedings of PLAN-X 2007*, 2007.
6. 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.
7. 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.
8. B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou. A Transducer-Based XML Query Processor. In *Proceedings of VLDB 2002*, pages 227–238, 2002.
9. 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, New Delhi, India, 2006. Tata McGraw-Hill Publishing Company Limited.
10. W3C. *XSL Transformations (XSLT) Version 1.0, W3C Recommendation*, 1999. `http://www.w3.org/TR/xslt`.
11. W3C. *XML Schema Part 1: Structures Second Edition, W3C Recommendation*, 2004. `http://www.w3.org/TR/xmlschema-1`.
12. W3C. *XML Schema Part 2: Datatypes Second Edition, W3C Recommendation*, 2004. `http://www.w3.org/TR/xmlschema-2`.
13. W3C. *XQuery 1.0: An XML Query Language, W3C Recommendation*, 2007. `http://www.w3.org/TR/xquery`.