Using Input Buffers for Streaming XSLT Processing

Jana Dvořáková, Filip Zavoral
Charles University in Prague, Czech Republic
(dvorakova, zavoral)@ksi.mff.cuni.cz

ABSTRACT
We present a buffering streaming engine for processing top-down XSLT transformations. It consists of an analyzer and a transformer. The analyzer examines given top-down XSLT and XSD, and generates fragments which identify parts of XSD need to be buffered when XSLT is applied. The fragments are passed to the transformer which processes XSLT on an input XML document conforming to XSD. It uses auxiliary memory buffers to store temporary data and buffering is controlled according to the fragments. We describe implementation of the engine within the Xord framework and provide evaluation tests which show that the new engine is much more memory-efficient comparing to the common XSLT processors.

1. INTRODUCTION
XSLT is typically processed by tree-based processors which store the whole input document in the memory and then apply the transformation. XML has started to be used extensively in domains where such traditional processing is not suitable, e.g.:

- data streams needed to be processed "on the fly",
- data processed in portable devices with limited memory,
- huge database exports exceeding available memory.

In this paper we focus on automatic streaming processing of XSLT transformations since currently there does not exist an appropriate alternative to the traditional automatic XSLT tree-based processors.

During the previous work on the Xord project in [5, 6], the Xord framework for the streaming processing of XSLT transformations was designed and implemented. The framework is intended to contain several streaming engines for processing XSLT. Each engine consists of an analyzer and a transformer. The analyzer analyzes given XSLT transformation and it determines whether it can be processed by given engine. It may pass some information collected during the analysis to the transformer which performs the transformation itself. The transformers are based on formal models called streaming XML transducers. This formal base enables us, for each transformation algorithm, to explicitly determine the class of XSLT transformations captured and the memory consumed. Within the framework, the SSXT\(^1\) engine was implemented. The SSXT transformer [5] processes a subset of top-down XSLT using stack of the size proportional to the depth of the input XML document. The SSXT analyzer [6] takes a schema and a top-down XSLT stylesheet, and it determines whether transformation can be processed by the SSXT algorithm on the XML documents defined by schema.

Although the SSXT transformation algorithm is highly memory efficient, the class of possible transformations is markedly restricted. The most important restriction is the order-preserving condition - the ordering of the output nodes must follow the order of the input document. In this paper, we present a BUXT\(^2\) engine that overcome these limitations. Some parts of the input document can be stored in buffers for future processing so that the output can be potentially in any order according to the input document.

The main contributions are the following:

- We design and implement the BUXT transformer which is able to process all top-down XSLT transformations. The base of the algorithm is the SSXT transformer. The BUXT transformer is extended with buffers for temporary storage and it is able to process more complex transformations.
- We design and implement the BUXT analyzer which is an extension of the SSXT analyzer. The BUXT analyzer statically computes the information about moments when buffering is needed based on an analysis of given schema\(^3\) and XSLT stylesheet. The information is provided in the form of schema fragments (shortly fragments) and passed to the BUXT transformer. Moreover, by examining fragments, it is possible to compute maximal amount of memory needed for processing the stylesheet on XML documents defined by given schema. See Fig. 1 for overall schema of the BUXT engine.
- We provide evaluation tests of the space complexity of the BUXT transformation algorithm and a comparison to commonly available XSLT processors.

Related work. Existing automatic XQuery streaming processors (BEA/XQRL [7], FluXQuery [9], XSM [10]) are typically designed for specific purpose. Moreover, they appear as black boxes - the streamability is achieved by ad-hoc optimizations and the amount of memory used for certain types of transformations is not known. XSLT automatic streaming processor SPM [8] uses known amount of memory, but it can process only very simple transformations and the class of transformations captured is not clearly characterized. Low-level streaming languages (STX [1], StAX [2]) based on event-based programming represent another alternative for handling transformations in the streaming manner. This approach, however, requires the user to write the transformation explicitly. Other research direction deals with streaming queries [11], but this is only one of the subproblems of the whole transformation process.

\(^1\)SSXT stands for simple streaming XML transducer.

\(^2\)BUXT stands for buffering XML transducer.

\(^3\)We use the terms XSD and schema interchangeably.


2. XSLT AND SCHEMA REPRESENTATION

We briefly describe subsets of XSLT and XSD considered in this work as well as the way how both structures are modeled in the BUXT engine.

2.1 XSLT representation

We consider a top-down fragment of XSLT language. It allows matching XSLT templates with modes and top-down XPath axes. A transforming template is called by an element name and a mode:

```xml
<xsl:template match="a" mode="m1">
  ... body ...
</xsl:template>
```

The template body consists of output elements (possibly nested) and template calls which call application of other templates by an XPath expression and a mode. The template calls are of the form:

```xml
<xsl:apply-templates select="child::a/descendant::b" mode="m2"/>
```

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

```
XPath := Step | Step/XPath
Step := (child | desc)::name
```

**Template model.** A template in BUXT engine is a structure `tmp` that consists of the following components:

- `tmp.match-name` - the name of the matching element,
- `tmp.mode` - the matching mode,
- `tmp.calls` - a sequence of template calls, a single call `call` consists of two components: `call.expression` and `call.mode`,
- `tmp.output-parts` - a sequence of output parts, the `i`-th output part is a sequence of tags to be generated between calls `i−1` and `i` (see the example below),
- `tmp.fragments` - a set of fragments.

**Example.** Let us consider the following XSLT template with two template calls:

```xml
<xsl:template match="a" mode="m0">
  <output-a1> <!-- output part 1 -->
    <output-a2> <!-- output part 2 -->
      <output-a3> <!-- output part 3 -->
        <!-- body ... -->
      </output-a3>
    </output-a2>
  </output-a1>
</xsl:template>
```

We obtain a BUXT template `tmp` of the form:

- `tmp.match-name = a`
- `tmp.mode = m0`
- `tmp.calls = (child:b, m1), (desc:c, m2)`
- `tmp.output-parts = <output-a1>, <output-a2>, <output-a2>output-a1>`
- `tmp.fragments = ∅`

The set of fragments is initially empty, it is filled up first during the analysis (see Section 4).

2.2 Schema representation

We consider schemas without `choice` constructor and recursive definitions. We represent such schema hierarchically as a schema tree. It consists of two kinds of nodes:

- `element nodes`: correspond to element types 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. An example of schema tree is depicted on the left-hand side in Fig. 2.

**Schema model.** A schema node in the BUXT engine is a structure that consists of the following components:

- `node.label` - an element name or a constructor symbol,
- `node.type` - either element node or constructor node,
- `node.children` - a sequence of references to child nodes.

Some subtrees of the schema tree may be identical - this situation occurs if we derive the schema tree from XSD containing shared element types. However, during the analysis, the order of the particular schema nodes is important. Therefore, such DAG structure is transformed to a tree during the analysis by duplicating shared nodes.

3. BUXT ENGINE OVERVIEW

The buffering streaming engine consists of two components: a BUXT analyzer and a BUXT transformer (see Fig. 1).

**Analyzer.** The analyzer takes two inputs: an XSLT stylesheet `xsl`, and a schema `xsd`. It accomplishes a static analysis of both inputs. As a result, it generates a set of fragments. Fragments basically store information on which parts of an XML document defined by `xsd` need to be buffered when the transformation `xsl` is processed on this document in the streaming manner. The fragments are passed to the transformer.

**Transformer.** The transformer takes two inputs: an XSLT stylesheet `xsl`, and an XML document `xml` valid with respect to `xsd`. It is based on the non-buffering SSXT streaming algorithm [5] which is extended by a possibility to store parts of the input temporarily in memory buffers. The decisions on when to start buffering and when to process buffer content are taken according to the information stored in the fragments.

Although we consider XSD format of schema, note that it is the BUXT schema model can be applied easily to another common format DTD as well.
The fragments represent the most key structure of the BUXT engine. A single fragment consists of the following components:

- \( \text{frag.tmp} \) - a reference to a template of \( xsl \),
- \( \text{frag.node} \) - a reference to a node of \( xsd \) (schema context node),
- \( \text{frag.items} \) - a set of fragment-items:
  - \( \text{item.call} \) - a reference to a call of template,
  - \( \text{item.sob-node} \) - a reference to a node of \( xsd \) (start-of-buffer node),
  - \( \text{item.eob-node} \) - a reference to a node of \( xsd \) (end-of-buffer node).

A fragment identifies a subtree in the schema tree parts of which require buffering when processed by the referenced template. A fragment-item identifies one of these parts, which is a subtree as well, and a specific call within the template which invokes buffering of this subtree.

A simple fragment is shown in Fig. 2. On the left-hand side, the tree of input XSLT stylesheet \( xsl \) is depicted and on the right-hand side, a schema tree of input schema \( xsd \) is depicted. The fragment identifies a subtree in the schema at node \( n_1 \) and associates it with \( \text{tmp}_1 \) of \( xsl \). The fragment-item depicted identifies the subtree at \( n_2 \) (which is a part of subtree at \( n_1 \)) and associates it with the call \( \text{call}_1 \) of \( \text{tmp}_1 \). The semantics of such fragment-item is as follows: If, during the transformation processing,

- the currently processed template is \( \text{tmp}_1 \),
- the current context tag corresponds to the schema node \( n_1 \),
- a match has been found for \( \text{call}_2 \),
- the current tag corresponds to the schema node \( n_2 \),
then the subtree at the current tag is stored in a new buffer \( \text{buf} \). During consequent processing, when the first two conditions above holds and moreover

- the current tag is end-tag,
- the current tag corresponds to the schema node \( n_3 \),
then the content of \( \text{buf} \) is processed. See Section 5 for more detailed description of the buffer manipulation.

4. ANALYZER

The analysis is driven by the structure of the schema tree, starting at the root node and continuing downwards to the leaves. The analyzer searches template of \( xsl \) which matches the current schema node at the current node and applies \( \text{analyzeNode} \) recursively, see Algorithm 1.

Algorithm 1. \( \text{analyzeNode} \) (tmp, schemaNode)

1: if \( \text{tmp.calls} \) is empty then \{end of analysis in current subtree\}
2: \{ do nothing \}
3: else if schemaNode is leaf then \{end of analysis in current branch\}
4: \{ do nothing \}
5: else
6: \( \text{frag} = \text{createFrag} \)(tmp, schemaNode);
7: if \( \text{frag.items} \) is not empty then
8: \quad add frag to \( \text{tmp.fragments} \);
9: end if
10: for each call in \( \text{tmp.calls} \) do
11: \quad for each node in \( \text{getCall} \)(call, schemaNode) do
12: \quad \quad let \( \text{calltmp} \) be template called by call;
13: \quad \quad \text{analyzeNode} \)(\( \text{calltmp} \), \( \text{node} \))
14: \quad end for
15: end for
16: end if

In case the current template does not contain any call (1), or the current schema node is leaf (3), the analysis terminates. Otherwise, the function \( \text{createFrag} \) for creating fragment is called (6), see Algorithm 2. It finds a fragment for the current template \( \text{tmp} \) which refers to the current schema node \( \text{schemaNode} \), and adds it to the set \( \text{tmp.fragments} \) in case the item set is not empty. After that, the analyzing function is called recursively (13) for all pairs \( (\text{calltmp}, \text{node}) \) such that

- \( \text{calltmp} \) is a template called by a call \( (\text{call.expression}, \text{call.node}) \) of \( \text{tmp} \),
- \( \text{node} \) is a schema node selected by \( \text{call.expression} \) in the schema tree if the evaluation starts at \( \text{schemaNode} \).

The evaluation of \( \text{call.expression} \) is accomplished by the function \( \text{getCall} \)(call, schemaNode) (11). It corresponds to the evaluation against the XML tree which is formed from the schema tree by omitting all schema constructors.

Algorithm 2. \( \text{createFrag} \)(tmp, schemaNode) : fragment

1: create new fragment \( \text{frag} \) such that
2: \( \text{frag.node} = \text{schemaNode} \), \( \text{frag.tmp} = \text{tmp} \), \( \text{frag.items} = \emptyset \);
3: for each call \( \text{call}_i \) in \( \text{tmp.calls} \) do
4: \quad set \( \text{matchedNodes}_i = \text{getCall} \)(\( \text{call}_i \), \( \text{schemaNode} \));
5: end for
6: for each call \( \text{call}_k \) do
7: \quad for each node in \( \text{matchedNodes}_k \) do
8: \quad \quad set \( \text{eobCandidates} = \{ \text{ candNode } \in \text{matchedNodes}_i \mid i < k, \text{ candNode } > \text{ precedent node} \} \}
9: \quad \quad \text{if} \( \text{eobCandidates} \) is not empty then
10: \quad \quad \quad set \( \text{eobNode} = \text{max}_\text{precedent}(\text{eobCandidates}) \);
11: \quad \quad \quad \text{add fragment-item (call}_k, \text{node}, \text{eobNode}) \) to \( \text{frag} \);
12: \quad \quad end if
13: \quad end for
14: end for
15: end for

The function \( \text{createFrag} \) first creates a fragment with empty set of items referencing to the current schema node and current template (1). Then particular fragment items are generated stepwise for each call of \( \text{tmp} \). We index the calls by an integer which corresponds to the order of the call in the sequence \( \text{tmp.calls} \). First,
the nodes matching call<sub>i</sub> are found for each i and stored in the sequence matchedNodes<sub>i</sub>. The order of the matched nodes conforms to the preorder with respect to the schema tree. Then the calls and their matching nodes are processed one by one (6-15). Let call<sub>k</sub> be the currently processed call and node<sub>k</sub> be the currently processed node from within the sequence. In next steps, the algorithm determines whether a fragment item exists such that

- item.sob-node = node<sub>k</sub>,
- item.call = call<sub>k</sub>,

The item exists if and only if some end-of-buffer node is found. First, all candidates for end-of-buffer node are collected. Each such candidate, let denote it cand-node, must conform to the following two conditions:

- cand-node ∈ matchedNodes<sub>i</sub>, i < k,
- cand-node > preorder node

It means, cand-node must appear after the currently processed node node and at the same time it must be a matching node of some call which appear before the currently processed call call<sub>k</sub>. This is exactly the situation when buffering is inevitable. The maximum candidate node (with respect to the preorder) is chosen as the end-of-buffer node (11) since it represents the position within the schema tree where all calls appearing before the current call have been definitely processed. In case no candidate has been found, the buffering is not needed and the fragment item is not generated.

Note that the overall fragment is added to the set tmp.fragments in the AnalyzeNode function if and only if the set of generated items is not empty.

Example. Let us consider the fragment shown in Fig. 2. The referenced template tmp contains the following two calls:

```
call1: child::b/child::c (matched at n<sub>3</sub>)
call2: child::c (matched at n<sub>2</sub>)
```

According to the order of calls in tmp.calls it holds

```
call1 <call-order call2,
```

but according to the preorder of the schema tree it holds

```
n<sub>3</sub> > preorder n<sub>2</sub>.
```

We thus obtain that call<sub>1</sub> must wait with generating output until call<sub>2</sub> is processed. The node n<sub>2</sub> is set as the start-of-buffer node since the subtree at node n<sub>2</sub> needs to be stored in the buffer buf for further processing. The node n<sub>3</sub> is the end-of-buffer node since at this node it is sure that call<sub>2</sub> has generated all of its output. This implies that buf can be processed by call<sub>1</sub> and the corresponding output can be generated.

5. TRANSFORMER

The algorithm of the BUXT transformer extends the original stack-based SSXT algorithm by memory buffers for temporary storage. It is based on the model called buffering XML transducer.

Buffering XML transducer. The transducer represents a combination of two models - the simple streaming XML transducer (SSXT) and the general XML transducer (GXT), both introduced in [4]. Following the SSXT behavior, it reads the input XML and generates the output XML in the streaming manner. It is equipped with a stack in order to enable stepwise evaluation of XPath expressions on the input stream SSXT. In addition, BUXT stores part of the input stream in buffers. These buffers are later processed in the traditional tree-based manner - the buffer content is stored in the memory as a tree and processed from the root to the leaves. The tree-based processing is accomplished by a GXT which is a straightforward formalization of the standard XSLT processors. The significant measures of space complexity are as follows:

- number of buffers,
- maximal size of the buffers utilized during the transformation.

Note that, for a given input document and XSLT transformation, the values of both measures can be computed statically. Such algorithm is however outside the range of this paper.

5.1 Structures and actions

We describe the structures and the actions used in the transformer algorithm.

Stack. Similarly to the original SSXT algorithm, the BUXT transformer uses a stack of the size proportional to the depth of the input document to remember information about particular element levels of the input XML document which is necessary to accomplish evaluation of XPath expressions. Two kinds of data are stored in the stack:

- DFA - a sequence of current DFA states,
- CC - cycle configuration.

A set of DFAs is used to evaluate XPath expressions in the current template concurrently - a single DFA is associated with a single expression. Such technique has been used for example in the Y-filter algorithm [3].

The cycle configuration contains information about the currently processed part of the XSLT stylesheet xsl and the current position in schema of the input document. The position in the schema is determined according to the current position in the input XML stream, and it is updated at each advance action (see action description below). A configuration cc consists of the following components:

- cc.tmp - a reference to a template of xsl (current template),
- cc.call - a reference to the lastly matched call of the current template,
- cc.context-node - a reference to a schema node (context schema node).

During a single cycle, one template call in a template of xsl is processed. A special initial cycle handles initialization of the transformation. A CC is pushed on the stack when a match is found for some expressions (i.e., a final DFA state appears in the current sequence of DFA states). Here, new cycle for processing the called template starts. A CC is popped after the called template has been processed and the control moves back to the previous template. More detailed description of SSXT stack manipulation can be found in [4].

The context schema node is a new component added first in the BUXT algorithm. It represents a position in the schema tree which corresponds to the current context tag of the transformation, i.e., the tag at which the current evaluation has started. Moreover, the transformation itself keeps a reference to the current schema node which corresponds to the currently processed tag. The pair (context schema node, current schema node) is called the current context. The current context is necessary in order to make proper decision on when to start buffering and when to process buffer contents.

For simplicity of the presentation, when describing the algorithm itself, we do not explicitly mention keeping references to the cur-
current context. These references are supposed to be updated in a straightforward way at each advance action.

**Buffers.** The buffers are in-memory tree XML structures which are used to store temporary data for later processing. They are processed in the tree-based manner, mimicking behavior of the standard XSLT processors.

**Actions.** Based on the SSXT algorithm, three actions are available for stack manipulation, one action for manipulating the input XML stream and one action for generating the output XML stream:

- push DFA, push CC, pop,
- advance (advances to the next input tag),
- generate (call1, call2) (generates all output XML tags between call1 and call2).

The BUXT algorithm uses, in addition, two more actions for manipulating buffer content:

- fill buffer - stores the content of the current element in a buffer,
- process buffer - process a buffer in the tree-based manner.

The decision about the buffer actions is based on the information stored in the fragments. If the current context corresponds to the fragment context and current node (tag) corresponds to a start-of-buffer node of one of its fragment-items, then a new buffer is filled by the current subtree. Similarly, the decision about processing a buffer is made, but the end-of-buffer node is checked instead of the start-of-buffer node.

### 5.2 Algorithm

The transformer algorithm uses the following variables (common for all functions mentioned below):

- cc - the current cycle configuration,
- current-tag - the currently processed XML tag,
- la-tag - the lookahead tag.

After an initialization, the transformer calls a proper function depending on the symbol on the top of the stack (see Algorithm 3).

**Algorithm 3 Transform(xsl, xml)**

1. set current-tag to first tag of xml;
2. set cc.tmp = template matching current-tag.name in mode mc;
3. set cc.call = cc.tmp.start;
4. push initial DFA states for cc.tmp;
5. while stack is not empty do
6. if stack.top = sequence S of DFA states then
7. processDFA(S);
8. else if stack.top = cycle configuration stack-cc then
9. ProcessCycleConfiguration(stack-cc);
10. end if
11. generate fragment (cc.call, cc.tmp.end)
12. end while

A sequence of DFA states is processed by the ProcessDFA function (see Algorithm 4). When a start-tag is encountered, all DFAs perform a transition according to the tag name and a new sequence of states denoted by $S$.transition(name) is determined (2). In case a final state appears in the new sequence, the transformer checks whether buffering is needed by examining all fragment items (6). If some of them contains start-of-buffer node for the current context, a new buffer is filled with the content of the current tag. Otherwise, the content is processed in the streaming manner and a new cycle starts (9).

When an end-tag is encountered, the transformer first checks whether some of the buffers might be ready for processing. It then examines fragment items and selects those which contains end-of-buffer node for the current context (17). For each such item, all associated buffers are processed in the tree-based manner. Then the streaming processing continues.

**Algorithm 4 ProcessDFA(S)**

1. if current-tag is start-tag then {Downwards evaluation}
2. let $S' = S$.transition(current-tag.name);
3. if $S'$ contains no final state then {No match}
4. push $S'$, advance;
5. else if $S'$ contains final state for call new-call then {Match found}
6. if fragment item item exists which contains start-of-buffer node for current context then {Buffer filling}
7. create new buffer buf and fill it with contents of current-tag;
8. add buf to item.buffers;
9. else
10. generate(cc.call, new-call);
11. push cc;
12. set cc.call = new-call.tmp;
13. set cc.call = cc.tmp.start;
14. end if
15. end if
16. else if current-tag is end-tag then {Upwards evaluation}
17. if fragment item item exists which contains end-of-buffer node for current context then {Buffer processing}
18. process all buffer in item.buffers;
19. end if
20. if la-tag is end tag then
21. pop; advance;
22. end if
23. end if

A cycle configuration is processed by the ProcessCycleConfiguration function (see Algorithm 5). In case a new cycle starts (1), a sequence of initial DFA states for the current template is pushed. In case a cycle ends (6), last output part of the current template is generated and the previous configuration is reset.

**Algorithm 5 ProcessCycleConfiguration(stack-cc)**

1. if current-tag is start-tag then {Cycle start}
2. if la-tag is start-tag then
3. push initial DFA states for cc.tmp;
4. end if
5. advance;
6. else if current-tag is end-tag then {Cycle end}
7. generate(cc.call, cc.tmp.end);
8. set cc = stack-cc;
9. pop;
10. end if

### 6. IMPLEMENTATION AND EVALUATION

The BUXT engine was implemented and tested in the Xord framework. The implementation is based on the SSXT engine by extending both its analyzer and transformer part according to the formal algorithms described above. Besides extending the algorithms themselves, there are two structural extensions:

- the set of fragments and their fragment items as the output of the analyzer phase and the input of the transformation phase
- keeping references to the relevant schema nodes while reading symbols from the input document.
We have compared the BUXT algorithm space complexity against the publicly available tree-based XSLT processors (Saxon, Xalan and XslProc) using both synthetic and real data. Fig. 3a. shows a comparison of transformation memory requirements of 10000 to 1 million entities. All the tree-based processors consumed large amounts of memory when processing large XML data (above 100K of entities) regardless the simplicity of the transformation.

The evaluation confirmed that the BUXT algorithm basically requires a memory proportional to the depth of the input XML. Fig. 3b. shows a net memory consumption of the algorithm (without libraries, runtime environment etc.) processing the input data of different depth. Since the document depth is generally not dependent on the document size and documents are relatively shallow [12], the memory requirements for most of the XML documents is low, independent to the document size. Even for large documents like DBLP (700 MB), the BUXT algorithm required below 100 KB of net memory while the above mentioned DOM-based processors crashed or hanged after allocating about 1.5 GB of memory.

Additionally, there is an extra memory required for each fragment item detected during the transformation. The size of such memory does not depend on the whole input size but on the schema and the XSLT structure. As long as the ordering of the output document remains close to the input document (the transformation is mostly local), the space complexity remains low. The most typical example of such processing is filtering, mapping and local reordering of a huge sequence of relatively small subtrees, such as logs, structured data streams or XML databases.

On the other side, the BUXT transformer is not very suitable for some classes of transformations. The example of inappropriate transformations is swapping two large subtrees or moving a little subtree from the end of the input document to the beginning of the output. For such transformations all of the input that should be processed later must be stored into the buffers and the space complexity may achieve the tree-based processors in the worst case.

7. CONCLUSION

We introduced an enhancement of the Xord framework for efficient XSLT processing. The functionality of the framework is currently based on the stack-based streaming algorithm which is able process a class of top-down XSLT transformations using stack of the size proportional to the depth of the input document. Additionally, some parts of the input document can be stored in buffers for later processing. The analyzer can detect the context when such buffering starts and when the content of such stored buffers should be processed instead the regular input. The results of our experiments show that the engine is much less memory-consuming when processing huge data sets or data streams comparing to the common tree-based processors for a wide class of transformations.

Several issues are left for the future work. First, we intend to overcome some restrictions to XSLT and schema constructs that can be processed by the Xord engine such as conditions and choices. The algorithms can be also improved by an appropriate schema inference strategies [13]. Next, we plan to design multipass algorithms that could be much more memory efficient for some classes of transformations at the cost of processing the input in several passes.

Acknowledgments.

This work was supported by the Grant Agency of the Czech Republic, grant number 201/09/0990 - XML Data Processing and by the Slovak Grant Agency, grant VEGA 1/3106/06. A part of the results presented comes from a PhD thesis of Comenius University in Bratislava, Slovakia.

8. REFERENCES


