Abstract: We present an implementation of the Xord framework for streaming processing of XSLT transformations based on .Net technologies. Within the framework, we implement an efficient streaming algorithm, called the SSXT algorithm, capable to process a significant subset of top-down XSLT transformations. We exactly characterize this class of transformations and implement schema-based analyzer which, for a given transformations, determines containment in this class. The SSXT algorithm uses a stack of the size proportional to the depth of the input XML document. Such memory usage is highly efficient in practice since real-world XML documents are shallow. The evaluation of the algorithm supports our expectations and shows that a transformation of large but shallow XML documents is processed using a constant size of memory.

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

1. Introduction

Common processors of XSLT [14] and XQuery [17] (Saxon, Xalan, AltovaXML) are tree-based, i.e., they store the whole input in the memory and then perform the transformation itself (see Fig. 1a). Such approach is sufficient for small XML documents stored in files. However, this classical processing is apparently not suitable for large XML documents and XML data streams - in the former case, it is not acceptable or even possible to store the whole input document in the memory, while in the later one, the XML data become available stepwise and need to be processed “on the fly”.

It is thus natural to employ the streaming processing, i.e., to read the input document sequentially in the document order as well as to generate the output document sequentially (see Fig. 1b). It is easy to see that, for certain classes of XML transformations, such streaming processor is substantially less memory-consuming than the tree-based processors.

The simplest approach to obtaining a program for streaming XML transformation is to write it completely by hand in some common programming language using an event-based parser such as SAX or StAX. Another possibility is to specify the transformation in a low-level streaming transformation language such as STX [1], XStream [10] or XTiSP [19]. However, both of these approaches are cumbersome for two reasons:

- The user must be familiar either with a common programming language and corresponding event-based API for XML processing, or with a special streaming transformation language. In both cases, a low-level approach to writing the transformation is used that is basically more difficult comparing to writing the transformation in popular high-level languages XSLT and XQuery.

- An extra effort is needed in order to obtain efficient streaming algorithm as the result - it is necessary to analyze the transformation by the user and to optimize the size of the memory buffers manually. Still, it is not guaranteed that the designed algorithm is close to the optimal one since the optimizations are typically done in an ad-hoc way.

In this paper, we are interested in the automatic approach, i.e., we focus on the problem how to design an automatic streaming processor for the commonly used transformation languages such as XSLT and XQuery. Such approach brings several advantages – the transformation is written in high-level languages that are familiar to users working in XML domain. At the same time, the process of determining an efficient streaming algorithm for given transformation is accomplished by the processor itself and is hidden to the user.

There are some significant challenges to design an automatic processor for high-level transformation languages. Both XSLT and XQuery are based on tree-manipulation and contain several non-streaming constructs. The 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...
The main contributions of this paper are the following:

1. We present a modular implementation of the Xord framework for streaming processing of XSLT transformations. The implementation consists of the Template Model, the Schema Model and the Algorithm Model. The framework has a solid formal base [6, 7]. It is intended to contain several streaming algorithms, each of them is first designed as a formal model called streaming XML transducer (SXT). For such model, the transformation class captured and the memory requirements are known.

2. Using both the Template Model and the Algorithm Model, we implement an efficient streaming algorithm for processing a subset of top-down XSLT transformations. We exactly characterize this class of transformations. The algorithm uses a stack of the size proportional to the depth of the input XML document. Such memory usage can be considered constant in practice, since real-world XML documents are shallow [18].

3. We mention experimental results for the algorithm implemented. The results show that the algorithm indeed consumes constant amount of memory when processing transformations from associated class on large and shallow XML documents.

The rest of this paper is organized as follows: In Section 2 we list automatic streaming processors for XSLT and XQuery. Section 3 presents the formal base of the framework. The class of top-down XSLT transformations and the schema model considered are described in Section 4 and Section 5, respectively. In Section 6, we introduce the stack-based algorithm, its basic principles, implementation and memory issues. Section 7 contains detailed description of the Xord framework implementation. Implementation of the SSXT algorithm is presented in Section 7. In Section 8 we mention the evaluation results. We conclude with summary and comments on future work.

2. State of the Art

Much of the previous work is devoted to streaming evaluation of XPath expressions, [2, 3, 4, 11, 12, 14, 16], The techniques used for XPath may be partially used also in the streaming algorithms for the transformations, but some significant problems are still to be solved (e.g., different order of the elements in the input document and in the output document).

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 these processors consume for a given type of XML transformation. The only exception it the XSLT processor based on SPM [13] which by definition does not have any additional memory. Due to the lack of ability to store data temporarily the processor is significantly less powerful than the SSXT algorithm that we introduce in Section 5. Moreover, the class of transformations captured is not characterized.

XML Streaming Machine (XSM) [17] processes a subset of XQuery queries on XML streams. It is based on XML streaming transducer that is a model for translating one or more input stream fragments into and output stream fragment. The transducer uses input buffers, an output buffer and one or more working buffers for realizing the translation. First, a given XQuery query is translated into a network of XSM’s. A single basic XSM corresponds to a subexpression of the XQuery query. Then the XSM’s are appropriately connected following the nesting of subexpression in the whole XQuery query. The resulting network of XSM’s is reduced into a single XSM by consequent application of composition algorithm and the resulting XSM is further optimized and translated into a C program. The processor has several limitations. Only XML documents without attributes and recursive structure definition are considered. Second, XSM is able to process only a simple, non-recursive XQuery query, however, the key aspects are included.

BEA/XQRL [9] is a streaming XQuery processor that is a central component of BEA’s WLI product. It implements full XQuery. The authors have compared the processor with Xalan-J processor on the set of 25 typical WLI transformations as well as on XMark Benchmarks. BEA processor appeared to be fast on small databases, however, the processing of large databases was slower due to the fact that the optimizations specially designed for XML streams are limited in this engine.

FluXQuery [15] is a streaming XQuery transformation engine based on a new internal query language called \texttt{FluX}. FluX extends the XQuery with new constructs for event-based processing. XQuery query is converted into
this internal representation and the buffer size is optimized by examining the query syntax as well as the input DTD. FluXQuery supports a subset of XQuery including nested for-loops and joins, but it does not handle aggregation functions. The engine was benchmarked against XQuery engines Galax and AnonX on selected queries of the XMark benchmark. These experiments showed that FluXQuery consumes less memory and runtime than other two engines.

SPM (Streaming Processing Model) [13] is a model of simple one-head, one-pass streaming processor without an additional memory. Authors present a procedure that tries to convert a given XSLT stylesheet into SPM. The streamable XSLT stylesheet is then defined to be such stylesheet for which the procedure does not report a failure. However, an algorithm for testing the streamability of XSLT is not introduced, and thus the class of XSLT transformations captured by SPM is not clearly characterized.

3. Formal base

The implementation framework for XSLT is based on the abstract framework introduced in [6, 7]. It consists of two groups of formal models:

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

All models are based on deterministic tree transducers, formal models for tree transformations [20] originated in the formal language theory. Both general and streaming XML transducers are defined in common terms in order to facilitate development of the streaming algorithms.

We define the general XML transducer (GXT) as the basic general model (see Fig. 4a). It is used to model all algorithmically computable XML transformations and their tree-based processing. 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 XPath expressions. At the same time, the output heads may generate a new part of the output.

We define the streaming XML transducer (SXT) as the basic streaming model (see Fig. 4b). It is used to model one-pass streaming processing without an additional memory. The SXT 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. By imposing various restrictions on the GXT, different XML transformation classes can be defined. On the other hand, the SXT can be extended by a memory to store temporary data or by allowing more passes over the input document. This way, we obtain models of several streaming processors – the corresponding streaming algorithms are basically the simulation of a restricted GXT by an extended SXT. The overall schema of the framework is shown in Fig. 3.

When designing a new streaming algorithm within the framework, it is necessary to find a correspondence between a restricted GXT and an extended SXT. In this paper, we present one such algorithm in which we consider the simple order-preserving branch-disjoint GXT and the simple SXT (SSXT). The algorithm is called SSXT algorithm according to the streaming XML transducer employed.

The GXTs are language-independent and in case a conversion method between GXT and a specific language is found, an implementation framework for this language can be developed in a straightforward way starting from the abstract framework. In this paper, we focus on XSLT transformations and therefore we directly consider XSLT transformation classes instead of GXTs.

At the design level, the implementation framework for the XSLT language consists of two basic modules:

1. Static analyzer analyzes the given XSLT stylesheet xsl and determines to which of the known classes of XSLT transformations it belongs. Specifically, a schema-based analyzer is used which performs the analysis according to
the information stored in an XML schema for input XML documents.

2. Transformer processes xsl using the algorithm associated with the determined transformation class.

4. Simple XSLT transformations

In the SSXT algorithm, we only consider simple XSLT transformations which are order-preserving and branch-disjoint.

Simple XSLT stylesheet contains an initializing template and several transforming templates. The initializing template sets the initial mode to m0 and call processing of the root element (node). It is of the form:

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

A transforming template is called by an element name and a mode:

```xml
<xsl:template match="a" mode="m1">
... template 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 output elements are of the form:

```xml
<element name>
... element body ...
</element name>
```

The template calls are of the form:

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

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

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

where name refers to an element name. The evaluation function for XPath expression exp with respect to the XML document d and one of its nodes u is denoted by eval(exp, d, u). The semantics of the eval function directly follows the semantics of evaluation of XPath expressions, the only difference is that in our case it is sufficient to consider a single node as the current context set.

We define an auxiliary function eval-exp that is needed later when defining restrictions put on simple XSLT transformations. Let tmp be a transforming template, d be an XML document, and u be a node of d. We define an auxiliary function eval-exp by

```
eval-exp(tmp, d, u) = eval(exp1, d, u) : : : eval(expn, d, u)
```

where exp1, ..., expn is a sequence of XPath expressions appearing in the template calls of tmp (in this order). Thus, the eval-exp function returns the concatenation of the node sequences returned by individual XPath expressions.

In order to determine whether a stylesheet is simple, it must be checked that it conforms the structure described. Now we describe order-preserving simple XSLT transformations and branch-disjoint simple XSLT transformations. In case a simple XSLT transformation does not conform to these conditions, the SSXT algorithm is not applicable - moreover, in majority of cases, additional memory buffers are needed in order to process the transformation.

Order-preserving simple XSLT. A 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 ∈ D,
- for each node u of XML document d,

it holds eval-exp(tmp, d, u) returns a sequence of nodes of d in document order.

Branch-disjoint simple XSLT. A simple XSLT xsl is 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 ∈ D,
- for each node u of XML document d,

it holds eval-exp(tmp, d, u) does not contain two nodes located within the same branch of d.

5. Schema model

As already mentioned, the static analyzer for the SSXT algorithm uses information stored in input XML schema (defining the set of input XML document for the transformation) in order to determine whether the transformation is processable by the algorithm. We consider XML schemas without the choice constructor and recursive definitions. Such schema can be represented as a schema tree in a straightforward way. The schema tree contains two types of nodes:

- Element type nodes - each represents particular element type defined within the schema
- Constructor nodes – each represents a constructor applied on an element type or a set of element types.

The structure of the tree naturally reflects the relationships among element types as well as the applications of the
Figure 5. An example schema tree

constructors to element types. An example schema tree is shown in Fig. 5.

The schema model considered is abstract, and thus not bounded to a particular schema language. However, in the prototype implementation we employ W3C XSD notation [22, 23] for XML schemas.

6. SSXT algorithm

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 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.

6.1. Processing cycle.

All XPath expression within a template are evaluating concurrently. The evaluation is realized by deterministic finite automata (DFA). A similar evaluation technique has been used in [5]. 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:

1. new sequence contains no final state - the input head continues in evaluation,
2. 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,
3. 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.

6.2. Memory usage

In the SSXT algorithm, a single sequence of DFA states is pushed on the stack when reading start-tags and no match is found, and a single sequence of DFA states is popped from the stack when reading end-tags and moving upwards in the element hierarchy. The sequences are obviously of constant length since the number of states in a sequence depends on the number of XPath expressions in the templates of $xsl$.

A new processing cycle starts when a match is found for some template call. Here an extra item - a cycle configuration - is pushed on the stack. However, when returning from processing the call, the cycle configuration is popped. Based on this observation, it is easy to see that the size of the stack never exceeds the number (depth of $d_{in} + 2$).

We treat two boundary situations in a special way - processing templates that do not contain any template calls and processing matches at leaves of $d_{in}$.

7. Xord framework

The Xord framework was designed for analyzing and transforming large XML data and XML streams. 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. 6):

1. Template Model for transforming templates implemented by the $XfXslt$ classes,
2. Schema Model for XML schemas implemented by the $XfSchema$ classes,
3. Analyzer Model for static analyzers implemented by the $XfXsdSxstAnalyzer$ and $XfTemplateAnalyzer$ classes.
6.1. Template model

In the Xord framework, XSLT stylesheets are represented by a set of classes, a Xord Template Model (see Fig. 7). 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.

6.2. 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 a 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. 8. The whole schema is represented as a recursively referenced polymorphic associative array of simple and complex type nodes. Each complex node contains a list of references to its child nodes with their cardinality. Using this recursive structure that forms a DAG (or a tree with one particular node selected as a root), the parsed schema could be easily traversed and processed.

6.3. Static analyzer

The static analyzer for SSXT algorithm checks the order-preservation and branch-disjointness of simple XSLT stylesheet in case the set of input XML documents is not restricted by a schema, i.e., any XML document over a given alphabet of element names may appear as the input. 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 set of documents defined by xsd. The analyzer is described in [8] in more detail.

4. AlgorithmModel for streaming transforming algorithms implemented by the XfSxst 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:

- SsxtAnalyzer - algorithm derived from the abstract Analyzer model, and using the Schema Model and the Template Model, and
- SsxtAlgorithm – transforming algorithm derived from the abstract Algorithm model.

Originally, since early experiments with analyzing and transforming algorithms required often substantial changes both in algorithms and data structures used, the Xord framework was implemented in C# on .Net platform because of its comfortable and effective development. But, in later phases when exact impact of different methods should have been measured, the runtime, especially the garbage collector, behaved unexpectedly – only minor implementation-level changes could significantly influence the results. Moreover, we wanted the framework to be as much platform independent as possible. Therefore we reimplemented the framework in the ISO/ANSI C++. Beside the stability of experiments we got even better measured values compared to the .Net version.
The implementation of the SSXT algorithm follows the Xord Ssxt Model (see Fig. 9) and it uses both the Template model and the Algorithm model classes. The Algorithm Model is an abstraction of all streaming algorithms to be designed within the Xord framework. Since the algorithm is stack-based, the main data structure used is a polymorphic stack of type XfStack which stores sequences of DFA states and cycle configurations.

Until the transformation is finished the top of stack is checked and the stack item is processed, see the method below. In case of an empty stack and nonempty remaining input new DFA sequence is pushed on the stack. Following listings of simplified pseudo-code demonstrate main ideas of described methods without nonessential technical details.

```
Ssxt::Run(xml)
{currTemplate = xslt.Start();
currCall = null;
transformed = false;
while(! transformed) {
    if(! stk.Empty()) {
        stk.Top.Process();
        break;
    } else {
        switch(currType) {
            case Element:
                new_ds = ds.Transition(currName);
                if(! new_ds.HasFinalStates()) {
                    stk.Push(new_ds);
                } else {
                    myCall = new_ds.CallWithFinalState();
                    Generate(currCall, myCall);
                    calledTemplate = SelectTemplate(currName, myCall.mode);
                    if(calledTemplate.Empty) {
                        calledTemplate.Generate();
                        currCall = myCall;
                        if(laType == Element)
                            stk.Push(new_ds);
                        xml.Advance();
                    } else {
                        stk.Push(CycleConfig(currTemplate, myCall));
                        currTemplate = calledTemplate;
                        currCall = null;
                    }
                    break;
            case EndElement:
                if(laType == EndElement)
                    stk.Pop();
                break;
            default:
                stk.Pop();
                break;
        }
    }
}
```

Listing 1. SSXT main loop

```
Ssxt::Run(xml)
{currTemplate = xslt.Start();
currCall = null;
transformed = false;
while(! transformed) {
    if(! stk.Empty()) {
        stk.Top.Process();
        break;
    } else {
        switch(currType) {
            case Element:
                if(laType == Element)
                    stk.Push(DfaSequence(currTemplate));
                Advance();
                break;
            case EndElement:
                Generate(currCall);
                transformed = true;
                break;
        }
    }
}
```

Listing 2. DFA sequences processing

```
Ssxt::Run(xml)
{currTemplate = xslt.Start();
currCall = null;
transformed = false;
while(! transformed) {
    if(! stk.Empty()) {
        stk.Top.Process();
        break;
    } else {
        switch(currType) {
            case Element:
                stk.Push(DfaSequence(currTemplate));
                Advance();
                break;
            case EndElement:
                Generate(currCall);
                transformed = true;
                break;
        }
    }
}
```

Listing 3. Cycle configuration processing

7.1. DFA sequences
The core of the processing DFA sequences 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.

```
DfaSequence::Process(xml)
{ds = GetDfaSequence();
 switch(currType) {
    case Element:
        new_ds = ds.Transition(currName);
        if(! new_ds.HasFinalStates()) {
            stk.Push(new_ds);
        } else {
            myCall = new_ds.CallWithFinalState();
            Generate(currCall, myCall);
            calledTemplate = SelectTemplate(currName, myCall.mode);
            if(calledTemplate.Empty) {
                calledTemplate.Generate();
                currCall = myCall;
                if(laType == Element)
                    stk.Push(new_ds);
                xml.Advance();
            } else {
                stk.Push(CycleConfig(currTemplate, myCall));
                currTemplate = calledTemplate;
                currCall = null;
            }
        }
    break;
    case EndElement:
        if(laType == EndElement)
            stk.Pop();
        break;
    default:
        stk.Pop();
        break;
}
```

7.2. Cycle configurations
The cycle configuration processing depends on the current XML node type. A start tag pushes a new DFA sequence while an end tag generates output.

```
CycleConfig::Process(xml)
{cc = GetCycleConfig();
 switch(currType) {
    case Element:
        if(laType == Element)
            stk.Push(DfaSequence(currTemplate));
        Advance();
        break;
    case EndElement:
        Generate(currCall);
        currTemplate = cc.template;
        currCall = cc.call;
        stk.Pop();
        break;
}
```

9. Evaluation
We have evaluated the SSXT algorithm against the
publicly available tree-based XSLT processors Saxon, Xalan and XSLTProc using both synthetic and real data. The evaluation showed that the SSXT algorithm requires a memory proportional to a depth of the input XML document. Since this depth is generally not depending on the document size and documents are relatively shallow (99% of documents have fewer than 8 levels whereas the average depth is 4 according to [18]), our memory requirements for most of the XML documents are constant, independent to the document size.

![Figure 11. Memory comparison](image)

On contrary, XSLT processing using standard processors like Xalan or Saxon uses DOM for the whole document which implies memory requirements proportional to the document size. Our measurements confirmed this expectation; Fig. 11 shows a comparison of transformation memory requirements of 10000 to 10 million entities.

![Figure 12. Dependency on document depth](image)

We have not included comparison to the streaming processor SPM [13] and to the transformation programs written manually using some event-based parser. In the first case, experiments were not possible since the SPM does not capture the class of top-down XSLT transformations considered. In the second case, the comparison would be pointless since the effectiveness of a hand-written program depends solely on the programmer skills and thus may vary.

We have experimented with various sets of XSLT, XSD and XML combinations; all the experiments proved memory required for the transformation depends only on the size of XSLT and XSD which are completely stored in memory and the document depth. Having equal XSD and XSLT, memory consumption was equal for 1 KB and 1 GB document.

The net memory consumption for the transformation (without space needed for the schema and template model) is shown in the Fig. 12 – somewhat blurred line shows that all the experiments resulted in nearly the same values, or more precisely, all the results were within 1% range.

10. Conclusion

We introduced an implementation 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. In practice, the stack size can be considered constant since real XML document typically contain only few levels of elements. We supported this observation by experiments - the results show that the algorithm is much less memory-consuming comparing to the standard DOM processing. Moreover, the algorithm is much more powerful than existing streaming algorithms for XSLT.

A lot of issues are left for the future work. Since the class of transformations is markedly restricted (especially the order preserving condition is significant) we work on the extension of the current algorithm to a buffer-based streaming algorithm in order to make it possible to process more complex transformations. The other promising direction is research on multiple passing transformation algorithms that should process complex transformations without a need of large data buffers.

Acknowledgements

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

References


Author Biographies

Jana Dvořáková was born in Bratislava, Slovakia in 1980. She received M.Sc. in computer science in 2004 and Ph.D. in theoretical computer science in 2008 at Comenius University in Bratislava. Her major fields of interest include streaming processing, XML technologies and complexity analysis.

Filip Zavoral was born in Prague, Czech Rep. in 1968. He received M.Sc. in computer science in 1991 and Ph.D. in software systems in 1998 at Charles University in Prague. His major fields of interest include software systems, semantic data processing and distributed systems.