On an Evaluation of Transformation Languages in a Fully XML-driven Framework for Video Content Adaptation

Davy De Schrijver, Wesley De Neve, Davy Van Deursen, Jan De Cock, and Rik Van de Walle
Department of Electronics and Information Systems - Multimedia Lab
Ghent University - IBBT
Gaston Crommenlaan 8 bus 201, B-9050 Ledeberg-Ghent, Belgium
davy.deschrijver@ugent.be

Abstract

Bitstream Structure Descriptions (BSDs) allow taking the complexity of transforming scalable bitstreams from the compressed domain to the semantic domain. These descriptions are an essential part of an XML-driven video adaptation framework. The performance of a BSD transformation engine is very important in such an architecture. This paper evaluates the efficiency of XML-based transformation languages in our video adaptation framework. XSLT, STX, and a hybrid solution are compared to each other in terms of execution times, memory consumption, and user-friendliness. Our experiments show that STX is the preferred solution when speed and low-memory are important. The hybrid solution is competitive in terms of memory consumption and is more user-friendly than STX. Although XSLT is relative fast, its memory consumption is very high.

1. Introduction

The increasing use of the device-independent Extensible Markup Language (XML) in numerous applications (e.g., configuration files, databases, MPEG-7 metadata descriptions,...) has lead to the need of efficient transformation languages. In this paper, the most common XML-based transformation languages are compared to each other, in particular Extensible Stylesheet Language Transformations (XSLT), Streaming Transformations for XML (STX), and a hybrid solution based on the just mentioned languages.
The advantage of expressing transformations by making use of these languages is the fact that the resulting stylesheets are also well-formed XML documents. Hence, Bitstream Structure Descriptions (BSDs), together with transformation stylesheets, are the core of a fully XML-driven framework for video content adaptation. Such a novel architecture will be used in this paper to evaluate the efficiency of the transformation languages. Its workflow is as follows. Firstly, the high-level structure of a scalable video bitstream is described in XML, resulting in a BSD. Subsequently, these BSDs are transformed in order to take into account different usage environment characteristics (e.g., screen resolution of a device, network bandwidth, etc.). Finally, starting from the transformed BSD, a new bitstream is generated (containing a lower resolution or visual quality). The efficiency of the transformation technology chosen is very important, on the one hand due to the time-constrained nature of such an adaptation framework, and on the other due to the typically large size of the BSDs. In this paper, the performance of the transformation languages is evaluated for the first time in the context of BSDs, resulting in a distinct preference for STX.
The outline of the paper is as follows. In Section 2, an overview is given of different XML-based transformation languages. The advantages and disadvantages are mentioned and the differences between the languages are explained by means of a small example. Our use case, in particular an XML-driven video adaptation framework, will be explained in Section 3. A discussion of the results is provided in Section 4. Finally, Section 5 concludes this paper.

2. Transformation languages

There are two approaches to interpret and to transform XML documents. Firstly, traditional procedural programming languages such as Java or C++, together with a parser, can be used to read in and consume XML data. Two main types of XML parsers exist. One built on top of tree-based models (e.g., Simple API for XML, SAX) and one on event-based models (e.g., Document Object Model, DOM). A comparison between these two technologies for large XML documents is given in [2]. Secondly, transformations can be implemented by using a standardized (XML-based) language together with a generic engine for interpreting

```
<library>
  <film>
    <id>0</id>
    <genre>Romantic</genre>
    <title>Romeo and Juliet</title>
  </film>
  <film>
    <id>1</id>
    <genre>Science Fiction</genre>
    <title>Star Wars</title>
  </film>
  ... and so on ...
</library>
```

Figure 1. Example of an XML document

the stylesheets. The main difference between the two approaches is the possibility to make use of generic software modules (transformation engines) in the latter case. These generic engines have the advantage that the received stylesheets (representing a transformation) are written in a well-known (standardized) XML-based language. Since these stylesheets are also well-formed XML documents, it is possible to obtain a complete XML-driven architecture where all communication is based on XML documents and where the architecture is built on top of generic software modules. Such a framework is explained in Section 3.

The two most common XML-based transformation languages, in particular XSLT and STX, together with a hybrid approach are compared in the next paragraphs. In order to explain the functioning of the different languages, we use a simple example of an XML document of which a fragment is given in Figure 1. This example contains a film library in which every film can be identified by a unique `id` tag. The transformations implemented must remove all films with odd `id`s.

```
<!--************************* XSLT Implementation ***************************-->
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="library">
    <xsl:copy>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>
  <xsl:template match="film">
    <xsl:if test="./id mod 2 = 0">
      <xsl:copy-of select="."/>
    </xsl:if>
    <!-- When the ID is odd, do not copy the film -->
  </xsl:template>
</xsl:stylesheet>
<!--************************ STX Implementation ****************************-->
<stx:transform version="1.0" pass-through="all" xmlns:stx="http://stx.
       sourceforge.net/2002/ns">
  <!-- Default, the input elements are copied to the resulting document -->
  <stx:variable name="keep_film"/>
  <stx:buffer name="current_film"/>
  <stx:template match="film">
    <stx:result-buffer name="current_film" clear="yes">
      <stx:copy>
        <stx:process-children/>
      </stx:copy>
    </stx:result-buffer>
    <stx:if test="$keep_film">
      <stx:process-buffer name="current_film" group="copy_film"/>
    </stx:if>
    <!-- When the ID is odd, do not copy the film -->
  </stx:template>
  <stx:template match="id">
    <stx:if test=". mod 2 = 0">
      <stx:assign name="keep_film" select="true()"/>
    </stx:if>
    <stx:else>
      <stx:assign name="keep_film" select="false()"/>
    </stx:else>
    <stx:process-self/>
  </stx:template>
  <stx:group name="copy_film"/> <!-- Copy film to the output stream -->
</stx:transform>
<!--********************* Hybrid Implementation ***************************-->
<stx:transform version="1.0" pass-through="all" xmlns:stx="http://stx.
       sourceforge.net/2002/ns">
  <!-- Default, the input elements are copied to the resulting document -->
  <stx:buffer name="xslt_stylesheet">
    <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/
          Transform">
      <xsl:template match="film">
        <xsl:if test="./id mod 2 = 0">
          <xsl:copy-of select="."/>
        </xsl:if>
        <!-- When the ID is odd, do not copy the film -->
      </xsl:template>
    </xsl:stylesheet>
  </stx:buffer>
  <stx:template match="film">
    <stx:buffer name="current_film">
      <stx:copy>
        <stx:process-children/>
      </stx:copy>
    </stx:buffer>
    <stx:process-buffer name="current_film" filter-method="http://www.w3.org
          /1999/XSL/Transform" filter-src="buffer(xslt_stylesheet)"/>
  </stx:template>
</stx:transform>
```

Figure 2. Transformation stylesheets in the different languages

2.1. XSLT

The easiest and most common way to implement an XML transformation is to make use of XSLT [3]. XSLT is based on an underlying tree-model such as DOM. This results in the fact that the first step in an XSLT engine is the construction of an internal tree representation of the corresponding XML document. As such, the memory consumption of the engine is in line with the size of the XML document. The in-memory generated tree closely reflects the content and structure of the XML document, which leads to an easy implementation of the transformation. It allows all kinds of modifications and all kinds of navigation possibilities. The engine iterates through the complete tree and for every node, the best matching XSLT template is executed. Out of each template, all other nodes of the tree can be reached by using XPath expressions and that information can be further used in the executed template. After the execution of the stylesheet, a resulting (DOM) tree is generated, which will be serialized to an XML document. This architecture leads to the impossibility to use XSLT in streaming applications because the complete XML document has to be converted into a tree model before the transformation can be started. Besides, the complete resulting tree has to be present before the transformed XML document can be generated.

An implementation in XSLT transforming our example is given in Figure 2. In this example, one can see the two templates needed. The first template is necessary to copy the outer tags and the second template copies the needed films to the result tree. In the second template, one can see that the test condition uses the value of a deeper lying tag in the tree by relying on XPath expression.

2.2. STX

As mentioned in the previous section, XSLT prevents the streaming processing of XML documents. In this case, one can make use of the STX transformation language [1]. STX is directly built on SAX and resembles XSLT in terms of concepts and language constructions. The template-based XML transformation language processes the SAX events almost immediately by executing the best matching template. Consequently, an STX processor does not have random access to all subsequently and previously parsed nodes of the document; it maintains an ancestor stack (i.e., all ancestor nodes together with all properties of the current node). The only exception to this rule is that the processor already knows the next SAX event. This look-ahead mechanism can be useful to process node values immediately or to check whether an element has child nodes or not. Because an STX processor does not keep an internal tree, is it impossible to use XPath as match pattern to specify a certain node or context. Therefore, STX defines a new expression language, in particular STXPath, which is similar to XPath 2.0 and has to be evaluated to the ancestor stack.
In Figure 2, an STX implementation is given for our example. One can immediately see that STX is more complex than the XSLT variant. A first important difference between the two XML-based languages is the usage of variables. In STX, the variables are adjustable (as in procedural languages) whereas in XSLT the variables cannot be updated. Because of the streaming characteristic of STX, every event should be processed immediately, resulting in the fact that forthcoming information cannot be used during the processing of the event. Therefore, STX provides buffers to solve this issue. Buffers are used to store SAX events, which can be transformed afterward. In our example, a `film` can only be transformed after the `id` of the corresponding film is known. Therefore, every film is buffered and dependent on the `id`, the film might be written to the output stream.

2.3. Hybrid solution: XSLT and STX

From the above, we can conclude that the advantages of XSLT are the disadvantages of STX and vice versa. In order to exploit the advantages of both languages, a hybrid approach can be used. In such an approach, STX reads the SAX stream, identifies small well-formed XML fragments, and passes these fragments to an XSLT processor. As such, the use of STX makes it possible to acquire a low memory footprint during the processing of the XML document. The XSLT stylesheet is subsequently used for doing look-ahead operations in the small fragment.
The last transformation in Figure 2 illustrates such a hybrid solution. The first (static) buffer contains the XSLT stylesheet (identified by the corresponding namespace) used for transforming a single film. The STX template buffers one film after which the buffer is transformed by calling the XSLT stylesheet using the `process-buffer` instruction.

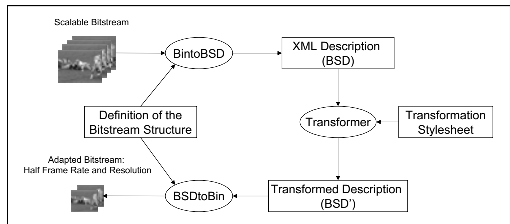

Figure 3. XML-driven adaptation framework

3. An XML-driven framework for video content adaptation

Scalable video coding is supposed to pave the way for several new multimedia architectures. They should make it possible to take the heterogeneity in the current plethora of devices and networks into account. In order to deliver scalable media in a diverse environment, it is important to be aware of the need of complementary logic that makes it possible to exploit the scalability properties of the compressed bitstream. This process typically involves the removal of certain data blocks and the modification of certain high-level syntax elements. One way to realize this scenario is sketched in Figure 3; it relies on automatically generated XML-based descriptions that contain information about the high-level structure of scalable bitstreams. Such a description can be generated by using a generic software module (`BintoBSD`). In a next step, these structural metadata can be transformed in order to reflect a desired adaptation of the scalable bitstream, and can subsequently be used to automatically create an adapted version of the bitstream (`BSDtoBin`). Typically, only a limited knowledge of the coding format is required in order to generate an XML-based Bitstream Structure Description (BSD). As such, a BSD acts as an abstraction of the coded bitstream. Technologies that can be used to obtain such BSDs are described in [2] and [5]. This paper focuses on the transformations.

4. Comparison and discussion

The bitstreams used in our XML-based adaptation framework are coded compliant with Joint Scalable Video Model 4 (JSVM4, [4]). This is a fully embedded scalable video coding specification that is currently under development. We have used 5 sequences of which XML descriptions (BSDs) have been generated. The coded bitstreams differ in length and so the generated XML documents as well. All bitstreams contain 4 spatial layers, 5 temporal

Table 1. Performance results: PU=Parse Units; BSD$_o$=Original BSD; BSD$_m$=Modified BSD; ET=Execution Time; MC=Memory Consumption

Name	#Frames	#PU	Bitstream Size (MB)	BSD$_o$ Size (MB)	BSD$_m$ Size (MB)	STX		XSLT		Hybrid	
						ET (s)	MC (MB)	ET (s)	MC (MB)	ET (s)	MC (MB)
sequence_1	50	663	0.84	1.4	0.8	2.1	2.7	1.8	4.2	9.0	2.9
sequence_2	100	1313	1.48	2.7	1.5	3.0	2.7	2.4	7.0	15.7	2.9
sequence_3	250	3263	8.67	6.1	3.7	5.6	2.7	4.1	15.5	35.4	2.9
sequence_4	500	6513	39.59	15.3	7.3	9.8	2.7	6.7	29.9	69.0	2.9
sequence_5	1000	13013	83.55	28.0	13.9	18.4	2.7	12.2	55.7	131.5	2.9

levels, and 3 quality layers. The characteristics of the bitstreams and corresponding BSDs are given in Table 1.
We have implemented transformation stylesheets along the 3 embedded scalability axes. Each stylesheet is implemented in (1) STX, where a complicated buffering mechanism is used to simulate look-ahead operations, (2) XSLT, where the information needed during the transformation can be drawn on immediately, and (3) a hybrid solution, in which every individual parse unit (called a `NALU`) is buffered by STX and transformed by XSLT. Because of space constraints, we only show the results for a temporal transformation where 1 temporal level is removed (resulting in a frame rate decreasing by a ratio 2, i.e., from 30Hz to 15Hz). The other transformations yield similar results. The results of the performance are given in Table 1 and Figure 4. The STX engine used in our tests is *Joost* (version 2005-05-21) and the XSLT engine is *Xalan 2.7.0*. From the results, it is clear that XSLT and STX are the fastest solutions and the ET is linear in terms of the number of PUs. The hybrid approach is slower but still linear. This is because an XSLT engine is called for every buffered `NALU`. XSLT is unarguably unusable for transforming BSDs because of the increasing memory consumption for larger XML documents (60MB for 30s of video). STX can transform a BSD in real time at a low memory footprint, which is a very important assessment in the context of an XML-driven framework for video content adaptation.

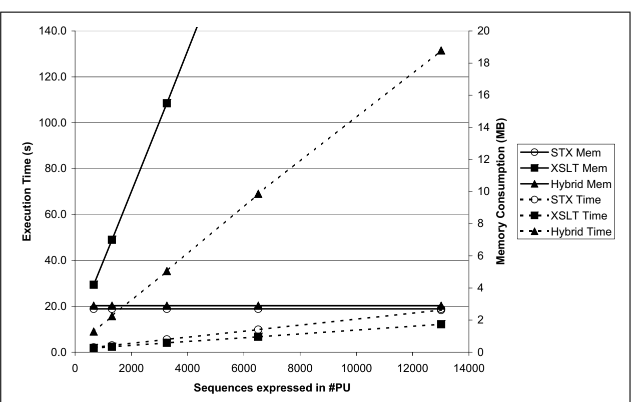

Figure 4. Evaluation results

5. Conclusions

In this paper, we performed an efficiency analysis of different XML-based transformation languages in the context of a fully XML-driven framework for video content adaptation. In such a framework, a scalable bitstream is adapted by transforming a high-level XML description instead of directly adapting the bitstream. These descriptions can be very large as a result that the performance of the transformation is important. We have compared XSLT, STX, and a hybrid solution. From our tests, we can conclude that STX is the best solution when considering execution times and memory consumption. Conversely, XSLT is fast enough but needs a lot of memory in case of larger XML documents and it cannot be used in streaming scenarios. The hybrid approach is more user-friendly and competitive with STX in terms of memory usage but it is multiple times slower.

6. Acknowledgements

The research activities that have been described in this paper were funded by Ghent University, the Interdisciplinary Institute for Broadband Technology (IBBT), the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT), the Fund for Scientific Research-Flanders (FWO-Flanders), the Belgian Federal Science Policy Office (BFSPO), and the European Union.

References

[1] P. Cimprich. Streaming transformations for XML version 1.0 working draft. *http://stx.sourceforge.net/documents/spec-stx-20040701.html*, July 2004.

[2] D. De Schrijver, C. Poppe, S. Lerouge, W. De Neve, and R. Van de Walle. MPEG-21 bitstream syntax descriptions for scalable video bitstreams. *Multimedia Systems*, In press.

[3] M. Kay. *XSLT Programmers's Reference, 2nd Edition*. Wrox Press Ltd., Birmingham, UK, 2001.

[4] J. Reichel, H. Schwarz, and M. Wien. Joint Scalable Video Model JSVM-4. *Doc. JVT-Q202*, October 2005.

[5] D. Van Deursen, W. De Neve, D. De Schrijver, and R. Van de Walle. BFlavor: an optimized XML-based framework for multimedia content customization. In *Proceedings of the 25th PCS*, pages 6 on CD–rom, Beijing, April 2006.