Processing and Delivery of Multimedia Metadata for Multimedia Content Streaming[1]

Michael Ransburg, Christian Timmerer, Hermann Hellwagner

Klagenfurt University
Universitätsstraße 65-67
A-9020 Klagenfurt
<first name>.<last name>@itec.uni-klu.ac.at

Sylvain Devillers

France Telecom R&D
BP 91226
F-35512 Cesson Sévigné CEDEX – France
sylvain.devillers@orange-ftgroup.com

Abstract: Today's increasing variety of media data results in a great diversity of XML-based metadata, which describes the media data on semantic or syntactic levels, in order to make it more accessible to the user. This metadata can be of considerable size, which leads to problems in streaming scenarios. Other than media data, XML metadata has no concept of "samples", thus inhibiting streamed (and timed) processing, which is natural for media data. In order to address the challenges and requirements resulting from this situation, the concept of *streaming instructions* is introduced. In particular, streaming instructions address the problem of fragmenting metadata, associating media segments and metadata fragments, and streaming and processing them in a synchronized manner. This is achieved by enriching the metadata with additional attributes to describe media and XML properties. Alternatively, a style sheet approach provides the opportunity to dynamically set such streaming properties without actually modifying the XML description.

1 Motivation and Scope

The role of XML-based metadata for describing distributed, advanced multimedia content gains more and more popularity in order to increase the access of such contents from anywhere and anytime. In the past, two main categories for this kind of metadata have become apparent [Fo06]. The first category of metadata aims to describe the semantics of the content such as keywords, violence ratings, or classifications. Metadata standards supporting this category are MPEG-7, TV Anytime, and SMPTE among others [AKS03]. The second

[1] Work partly supported by the IST European projects DANAE (IST-1-507113) and ENTHRONE (n° 038463)

category of metadata does not describe the semantics, but rather the syntax and structure of the multimedia content. This category spans a wide range of research activities enabling codec-agnostic adaptation engines for scalable contents by providing languages for describing the bitstream syntax. Examples for such languages are the Bitstream Syntax Description Language (BSDL) as specified in MPEG-21 DIA [Ve04], BFlavor [De06], and XFlavor [HE02]. Note that MPEG-7 also provides means for describing syntactical aspects of multimedia bitstreams [BS06].

Both categories of metadata (semantic and syntactic descriptions) have in common that they are desired to become more and more detailed, as this increases the accessibility of the media content. They often describe the content per segment or even per access unit (AU), which are the fundamental units for transport of media streams and are defined as the smallest data entity which is atomic in time, i.e., to which a single decoding time can be attached. For example, a single violence rating for the whole movie might exclude many potential consumers if it contains only one or two extremely violent scenes. However, if the violence rating is provided per scene, for instance, the problematic scenes could simply be skipped for viewers who do not wish to see them. Similarly, if a scalable multimedia content only describes the temporal enhancement layers, terminals requiring spatial adaptation (e.g., a mobile device) are excluded. Again, more descriptive metadata (i.e., describing spatial, temporal, and fine-grained scalability) would increase the accessibility of the content. As a consequence, this metadata is often of a considerable size, which – even when applying compression – is problematic in streaming scenarios. That is, transferring entire metadata files – if possible at all – before the actual transmission of the media data, could lead to a significant startup delay. Additionally, there is no information on how this metadata is synchronized with the corresponding media, which is necessary for streamed (i.e., piece-wise) processing thereof. The concept of piece-wise (and timed) processing is natural for media data. For example, a video consists of a series of independent pictures which are typically taken by a camera. These independent pictures are then encoded, typically exploiting the redundancies between these pictures. The resulting AUs can depend on each other (e.g., in the case of bidirectional encoded pictures) but are still separate packets of data. Although the characteristics of content-related metadata are very similar to those of timed multimedia content, no concept of “samples” exists for this metadata today.

In this paper we introduce the concept of “samples” for metadata by employing streaming instructions for both XML metadata and media data. The XML streaming instructions specify the fragmentation of the content-related metadata into meaningful fragments and their timing. These fragments are referred to as process units (PUs), which introduce the “samples” concept – known from audio-visual content – to content-related metadata. The media streaming instructions are used to locate AUs in the bitstream and to time them properly. Both types of streaming instructions enable time-synchronized, piece-wise (i.e., streamed) processing and delivery of media data and its related metadata. Furthermore, the fragmentation mechanism helps to overcome the startup delay introduced by the size of the metadata. Another, less obvious, benefit is described in an application scenario (see Section 5) where the streaming instructions enable to extend an existing static media adaptation approach to dynamic and distributed use cases.

Section 2 summarizes the requirements for the streaming instructions. Related work is discussed in Section 3. Section 4 describes the streaming instructions in detail. An application scenario, which illustrates the benefits of the streaming instructions, is presented in Section 5. Section 6 provides a performance evaluation of the streaming instructions and of an adaptation server which facilitates the streaming instructions to enable dynamic and distributed adaptation. Section 7 concludes this paper and points out possible future work items.

2 Requirements

This section lists the basic requirements, which we identified for the streaming of metadata and related media data:

- The streaming instructions need to describe how metadata and/or associated media data should be fragmented into PUs (for metadata) and AUs (for media data) respectively, for processing and/or delivery.
- A PU has to be well-formed (w.r.t. an XML schema) and needs to be able to be consumed as such by a terminal (i.e., no other fragments are needed to consume it). This enables piece-wise processing and it also enables to re-use existing tools for processing the metadata (see Section 5 for an example).
- The streaming instructions shall enable to assign a timestamp to a PU and/or an AU indicating the point in time where the fragment shall be available to a terminal for consumption.
- The streaming instructions need to provide mechanisms, which allow a user to join a streaming session that is in progress. This means that one needs to be able to signal when a PU and/or AU shall be packaged in such a way that random access into the stream is enabled.
- It shall be possible to apply the streaming instructions without modifying the original XML document as there may be use cases, where it is not possible or feasible to modify the multimedia content and its metadata, e.g., due to digital rights management issues.
- A streaming instructions processor shall work in a memory and runtime efficient way.

3 Related Work

In this section we review related work in the literature that deals with mechanisms enabling streamed processing and transport of multimedia content and related metadata. Multiple mechanisms for specifying the fragmentation and timing of media content are well known, e.g., the sample tables of the ISO Base Media File Format [BMF]. The difference is that in our approach this information is specified as a part of the metadata. This coupling provides a common way for a user to specify the fragmentation and timing of both media and metadata.

MPEG is currently standardizing so called Multimedia Application Formats, which aim at combining technology from MPEG and other standardization bodies to specify a specific application, e.g., a photo player and a music player [DPC05]. All these applications employ XML metadata and currently either use it only on a track/movie level or they use mechanisms from the ISO Base Media File Format to provide the timing of more dense metadata. However, this requires that the metadata is already fragmented beforehand and that the metadata is therefore no longer available in its original format for non-streamed processing.

Wong et al. [WCL03] define a method for fragmenting an XML document for optimized transport and consumption, preserving the well-formedness of the fragments. However, what is consumed are not the fragments themselves but rather the document resulting from the aggregation of the fragments. Furthermore, the fragmentation is achieved according to the size of the Maximum Transport Unit (MTU) and not based on the semantics of the fragment, i.e., no syntax is provided for a content author to specify which fragments should be consumed at a given time.

Alternatively, MPEG-7 provides an encoding method (Binary Format for XML) to progressively deliver and consume XML documents in an efficient way [Ni02]. Therefore, so-called Fragment Update Units (FUUs) provide means for altering the current description tree by adding or removing elements or attributes. However, MPEG-7 only specifies the syntax of FUUs and its decoding, whereas our work concentrates on the composition of XML fragments.

Interestingly, in both cases above, no timing information is provided which enables the synchronized use of the metadata and the corresponding multimedia content.

The Continuous Media Markup Language (CMML) [PPP04] is an XML-based mark-up language for time-continuous data similar to MPEG-7. Together with the Annodex file format [PPP05] it allows to interleave time-continuous data with CMML mark-up in a streamable manner. This approach is specific to CMML whereas in our work we aim to offer a generic solution for time-synchronized, streamed processing and transport for media and related metadata.

The Synchronized Multimedia Integration Language (SMIL) [Ru01] provides a timing and synchronization module which can be used to synchronize the play-out of different media streams. However SMIL is only concerned with media as a whole and therefore no AU location, fragmentation, and timing for metadata are provided.

The Simple API for XML (SAX) is an event-based API which allows streamed processing of XML [Si03]. It allows to parse an XML document without loading the complete document into memory. This does help to avoid the startup delay for streamed processing. However, legacy applications which rely on DOM would need to be re-implemented (e.g., the example application in Section 5). Moreover, no timing or fragmentation information is provided for piece-wise and synchronized processing of media and metadata. However, SAX might further increase the performance of our current implementation where we currently use an XML Pull Parser (see Section 6).

Our concept is close to a mechanism provided by Scalable Vector Graphics (SVG) [Qu03] to indicate how a document should be progressively rendered: the `externalResourcesRequired` attribute added to an element specifies that the document should not be rendered until the sub-tree underneath is completely delivered. This mechanism is specific to SVG. With this mechanism, the last state of the output document is the input document itself. In contrast, our method allows isolating a fragment that can be consumed at a given time, but this fragment does not need to contain the previous one. In particular, it is possible to progressively consume a document without ever the need of loading the full document into memory since only a fragment is consumed at a time.

To the best of our knowledge, the concept of PU and in particular the method we developed for specifying their composition, processing, and their transport in conjunction with media fragments is therefore original.

4 Streaming Instructions

We introduce three different mechanisms to respond to the requirements described in Section 2:

1. The XML streaming instructions describe how XML documents shall be fragmented and timed.
2. The media streaming instructions localize AUs in the bitstream and provide related time information.
3. Finally, the properties style sheet provides means to describe all of the above properties in a separate document, rather than directly in the metadata.

The XML and media streaming instructions are defined as properties. The properties are abstract in the sense that they do not appear in the XML document, but augment the element information item in the document infoset [XIS04]. They can be assigned to the metadata by using XML attributes and/or by the properties style sheet specified in Section 4.3.

Additionally, an inheritance mechanism is defined for some of these properties: the value of the property is then inherited by all descendant elements until the property is defined with a different value which then supersedes the inherited value, and is itself inherited by the descendants. Lastly, a default value is specified for each property.

In the sequel, we will introduce the mechanisms listed above separately and then combine them as they are applied to a specific scenario in Section 5.

4.1 XML Streaming Instructions

The XML streaming instructions provide the information required for streaming an XML document by the composition and timing of PUs. The XML streaming instructions allow firstly to identify PUs in an XML document and secondly to assign time information to them. A PU is a set of connected XML elements. It is specified by one element named anchor element and by a PU mode indicating how other connected elements are aggregated to this anchor to compose the PU. Depending on the mode, the anchor element is not necessarily the root of the PU. Anchor elements are ordered according to the navigation path of the XML document. PUs may overlap, i.e. some elements (including anchor elements) may belong to several PUs. Additionally, the content provider may require that a given PU be encoded as a random access point, i.e. that the encoded PU (the AU) does not require any other AUs to be decoded.

Figure 1 illustrates how an XML document is fragmented and timed using the XML streaming instructions. The fragmenter uses as input the XML document to be streamed and a set of XML streaming instructions properties provided either internally (as XML attributes with the XMLSI namespace) and/or externally (with a properties style sheet as specified in Section 4.3). The output of the fragmenter is a set of timed PUs.

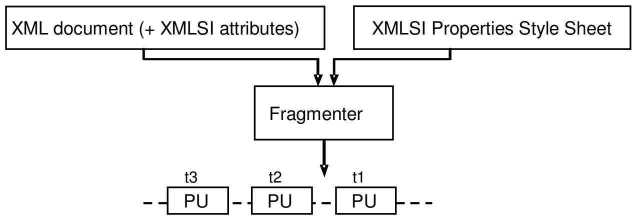

XML document (+ XMLSI attributes)
XMLSI Properties Style Sheet
Fragmenter
t3
t2
t1
PU
PU
PU

Figure 1: Processing related to XML streaming instructions

The fragmenter parses the XML document in a depth-first order. XML streaming instructions properties are computed as explained below. An element with the `pu` property set to true indicates an anchor element and a new PU. The PU then comprises connected elements according to the `puMode` property of the anchor element.

In the following the XML streaming instructions properties, as listed in Table 1, are specified for:

- Fragmenting an XML document into PUs.
- Indicating which PUs shall be encoded as random access point.
- Assigning time information (i.e., processing time stamp) to these PUs.

Table 1: XML streaming instructions properties

Name	Possible Values	Inherited	Default Value
anchorElement	undefined, false, true	no	undefined
puMode	undefined, self, ancestors, descendants, ancestorsDescendants, preceding, sequential	yes	undefined
encodeAsRap	undefined, false, true	yes	undefined
timeScale	undefined, an integer value	yes	undefined
ptsDelta	undefined, an integer value	yes	undefined
absTimeScheme	undefined, a string value	yes	undefined
absTime	undefined, a string value	no	undefined
pts	undefined, an integer value	no	undefined

The `puMode` property specifies how elements are aggregated to the anchor element (identified by the `anchorElement` property) to compose a PU. Figure 2 gives an overview of the different `puModes`, which were derived by analyzing various types of metadata (as introduced above) and their applications (see Section 5 for a detailed description of an example application). The objective was to constrain ourselves to as few `puModes` as possible, while still supporting all sensible applications, in order to enable an efficient implementation. The semantics of the different `puModes` are as follows, given that the white node in Figure 2 contains an `anchorElement` property which is set to true:

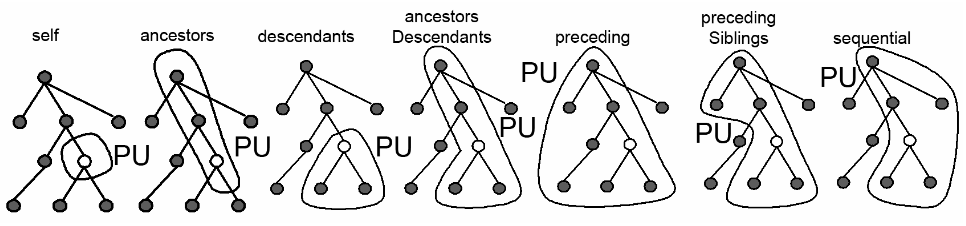


Figure 2: Examples of the different `puModes`

self: the PU contains only the anchor element.
ancestors: the PU contains the anchor element and its ancestors stack, i.e. all its ancestor elements.
descendants: the PU contains the anchor element and its descendant elements.
ancestorsDescendants: the PU contains the anchor element, its ancestor and descendant elements.

preceding: the PU contains the anchor element, its descendant and parent elements and all the preceding-sibling elements of its ancestor elements and their descendants.
precedingSiblings: the PU contains the anchor element, its descendant and parent elements and all the preceding-sibling elements (and their descendants) of its ancestor element.
sequential: the PU contains the anchor element, its ancestors stack and all the subsequent elements (descendants, siblings and their ancestors) until a next element is flagged as an anchor element.

The `encodeAsRAP` property is used to signal that the PU should be encoded as a random access point in order to enable random access into an XML stream. The `timeScale` property provides the number of ticks per second. The `ptsDelta` property specifies the interval in time ticks after the preceding anchor element. Alternatively, the `pts` property specifies the absolute time of the anchor element as the number of ticks since the origin. The timing can not only be specified in ticks: the `absTime` property specifies the absolute time of the anchor element. Its syntax and semantics are specified according to the time scheme used (`absTimeScheme` property), e.g., NPT, SMPTE or UTC.

4.2 Media Streaming Instructions

The media streaming instructions specify two sets of properties for annotating an XML document. The first set indicates the AUs and their location in the described bitstream, the random access points, and the subdivision into AU parts. The second set provides the AU time stamps.

Figure 3 illustrates how AUs in a bitstream are located and timed using the media streaming instructions. The fragmenter uses as input the bitstream to be streamed and a set of media streaming instructions provided either internally (as attributes) and/or externally (with a properties style sheet). The output of the fragmenter is a set of timed AUs.

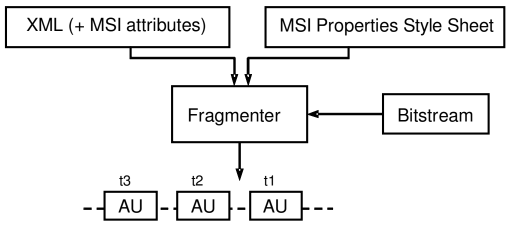


Figure 3: Processing related to media streaming instructions

The fragmenter parses the XML document in a depth-first order. The media streaming instructions properties are computed as specified below. Anchor elements (i.e., elements with the `au` property set to true) are ordered according to the parsing order and so are the corresponding AUs. An anchor element indicates the start of an AU, the extent of which is specified by the `auMode` property.

In the following, the media streaming instructions properties, as listed in Table 1, are specified for:

- Locating AUs in the bitstream.
- Indicating which AUs shall be encoded as random access point.
- Assigning time information (i.e., processing time stamp) to these AUs.

Table 2: Media streaming instructions properties

Name	Possible Values	Inherited	Default Value
auMode	tree, sequential	yes	tree
au	undefined, false, true	no	undefined
auPart	undefined, false, true	no	undefined
rap	undefined, false, true	yes	undefined
timeScale	undefined, an integer value	yes	undefined
dts	undefined, an integer value	no	undefined
cts	undefined, an integer value	no	undefined
dtsDelta	undefined, an integer value	yes	undefined
ctsOffset	undefined, an integer value	yes	undefined
addressUnit	bit, byte	yes	undefined
start	undefined, an integer value	no	undefined
length	undefined, an integer value	no	undefined

The media streaming instructions, as listed in Table 2, are tailored to metadata which can linearly describe a bitstream on an AU granularity, such as BSD, gBSD [Ve04], BFlavor [De06], XFlavor [HE02] or MPEG-7 MDS [Si01]. The start of an AU is indicated by an element with an `au` property set to true. This element is named anchor element. The media streaming instructions indicate the `start` and the `length` of an AU in bits or bytes (depending on the `addressUnit` property). The extent of the AU depends on the value of the `auMode` property of the anchor element as depicted in Figure 4 (the white node indicates an element with the `au` property set to true). In the sequential mode, the AU extends until a new element is found with an `au` property set to false or true. If no element is found with an `au` property set to true or false, the AU extends until the end of the bitstream. In the tree mode, the AU is the bitstream segment described by the XML sub-tree below the element flagged with the `au` property set to true. AU parts are defined in a similar way. The start of a new AU part in an AU is indicated by an `auPart` property set to true and the extent is specified by the `auMode` property. In the sequential mode, the AU part extends until a new element has an `auPart` property set to false or true (in the latter case, a new AU part immediately follows), until the end of the AU, or until the end of the media bitstream. In the tree mode, the AU part is the bitstream segment corresponding to the sub-tree below the element flagged by the `auPart` property. The `auPart` property provides a way for indicating AU parts within an AU in a coding format independent way. In this way, a streaming server that is not aware of the format of the streamed media content may nevertheless meet the requirements of a specific RTP payload format, e.g., special fragmentation rules.

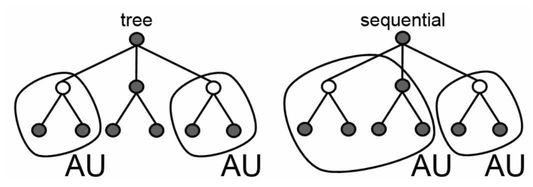


Figure 4: Examples of the different `AUModes`

Other information about AUs is specified by the properties of the anchor element. In particular, the AU is a random access point if the `rap` property of the anchor element is set to true. The `rap` property is inheritable, and it is therefore possible to inherit this property to each AU (i.e., each AU is a RAP) by setting the rap property of the XML root element to true. The time information of the AU (CTS and DTS) is also specified by the properties of the anchor element as explained below. The media streaming instructions use an absolute and a relative mode for specifying time information. In absolute mode, the CTS and DTS of an AU are specified independently from other AUs. In relative mode, the CTS and DTS are calculated relatively to the CTS and DTS of the previous AU. Both modes can be used in the same document. For example, an absolute date can be applied to a given AU, and the CTS and DTS of the following AUs are calculated relatively to this AU. In both modes, CTS and DTS conform to a time scale, i.e. they are specified as a number of ticks. The duration of a tick is given by the time scale which indicates the number if ticks per second, which allows for fine granular timing of AUs. The time scale is specified by the `timeScale` property. The two properties `cts` and `dts` define the CTS and DTS of the AU, expressed as an integer number of ticks. They are not inheritable and may be applied to an anchor element for specifying the CTS and DTS of the corresponding AU. Alternatively, two properties named `dtsDelta` and `ctsOffset` allow calculating the DTS and CTS of the AU relatively to the previous AU. The `dtsDelta` property indicates the time interval in ticks between the current AU and the previous one. The `ctsOffset` property indicates the time interval in ticks between the DTS and the CTS of the current AU. Some media codecs do not require a CTS information. In this case, the `cts` and `ctsOffset` properties are not used and may be undefined.

For each anchor element, the properties of the corresponding AU are then calculated as follows:

```
if isPresent(dts(n)) {  DTS(n) = dts(n); } else {
 if n = 0 {  // i.e., first AU
```

```
   DTS(n) = 0;
  } else { DTS(n) = ((DTS(n-1) + DTS_DELTA(n-1))/TIME_SCALE(n-1)) *  TIME_SCALE(n); }
}
if isPresent(cts(n)) { CTS(n) = cts(n); } else {  CTS(n) = DTS(n) + ctsOffset; }
TIME_SCALE(n) = timeScale(n); DTS_DELTA(n) = dtsDelta(n); RAP(n) = rap(n);
```

Here dts(n), cts(n), timeScale(n), dtsDelta(n), ctsOffset(n), rap(n) represent the media streaming instruction properties of the nth anchor element, and DTS(n), CTS(n), TIME_SCALE(n), DTS_DELTA(n) and RAP(n) represent the properties of the associated n^{th} AU.

4.3 Properties Style Sheet

It is also possible to specify the XML and media streaming instructions properties without adding XML attributes to the original document. This is in particular useful when associated Digital Rights Management (DRM) information forbids editing the original document and/or where the properties are set according to a regular pattern, as this reduces the overhead introduced by the streaming instructions. It also eases the management of multiple media contents (and their related metadata) which are fragmented and timed in the same way. Then, instead of annotating each XML document a single *properties style sheet* can be used. This external document specifies a set of properties which should be set for all elements matching a given pattern. For expressing such patterns, we introduce a new expression language named Lightweight Expression language (LXPath) based on STXPath. STXPath is an expression language developed in the context of STX (Streaming Transformations for XML) [STX04], a transformation language enabling the streamed transformation of an XML document, i.e., without building a tree in memory. The syntax of STXPath is similar to XPath [XPL99], but its semantics differ. Whereas an XPath expression is resolved against the full document, an STXPath expression is resolved against a limited context consisting of the current element, its ancestor's stack and its position within siblings. For example, in XPath, the expression `/node1/node2` returns a sequence containing all `node2` elements, whose parent element is the document element and is named `node1`. In LXPath, on contrary, the same expression returns a sequence containing a single node from this node-set; the one which is an ancestor of the current node. The use of STXPath expressions as matching patterns enables filtering an XML document without loading the full tree into memory, and is suitable for efficient SAX-based architectures. In our approach, we define a limited subset of STXPath required for locating elements in an efficient and simple way.

```
<?xml version="1.0"?>
<schema version="ISO/IEC 21000-7:2004/Amd.2" id="PSS.xsd"
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:ps="urn:mpeg:mpeg21:2003:01-DIA-PSS-NS"
  targetNamespace="urn:mpeg:mpeg21:2003:01-DIA-PSS-NS">
```

```
 <element name="properties"><complexType><sequence>
  <element name="template" minOccurs="0" maxOccurs="unbounded">
   <complexType><sequence>
    <element name="property" minOccurs="0" maxOccurs="unbounded">
     <complexType>
      <attribute name="name" type="QName" use="required"/><attribute name="namespace" type="anyURI"
       use="optional"/><attribute name="value" type="string" use="required"/>
   </complexType></element></sequence>
   <attribute name="match" type="string" use="required"/>
</complexType></element></sequence></complexType></element></schema>
```

Document 1: Properties Style Sheet XML Schema

As shown in Document 1, the properties style sheet consists of a sequence of templates specified by a matching pattern expressed in LXPath and containing a list of properties defined by a qualified name and a value. This properties style sheet and LXPath are designed in a way such that properties can be applied on-the-fly in a SAX-based architecture. While parsing the original document with a SAX parser, each new element is matched against each of the templates, and the corresponding properties are set accordingly.

Table 3: Grammar for LXPath in EBNF notation

MatchPattern	::=	BoolExpr
BoolExpr	::=	Expression ("|" Expression)*
Expression	::=	("/" | "//")? PathStep (("/" | "//") PathStep)*
PathStep	::=	(QName | WildCard) Predicate*
WildCard	::=	"*" | ("*" ":" NCName) | (NCName ":" "*")
Predicate	::=	"[" PredicateExpr "]"
PredicateExpr	::=	OrExpr
OrExpr	::=	AndExpr (("or" | "|") AndExpr)*
AndExpr	::=	ComparisonExpr ("and" ComparisonExpr)*
ComparisonExpr	::=	AdditiveExpr (GeneralComp AdditiveExpr)?
GeneralComp	::=	"=" | "!=" | "<" | "<=" | ">" | ">="
AdditiveExpr	::=	MultiplicativeExpr (("+" | "-") MultiplicativeExpr)*
MultiplicativeExpr	::=	PrimaryExpr (("*" | "div" | "idiv" | "mod") PrimaryExpr)*
PrimaryExpr	::=	AttrExpr | Function | StringLiteral | NumericLiteral
AttrExpr	::=	"@" NCName
Function	::=	"position()"
StringLiteral	::=	""" Char* """
NumericLiteral	::=	IntegerLiteral | DecimalLiteral
IntegerLiteral	::=	("-" | "+")? Digits
DecimalLiteral	::=	("-" | "+")? ("." Digits) | (Digits "." [0-9]*)
Digits	::=	[0-9]+
NCName	::=	[http://www.w3.org/TR/REC-xml-names/#NT-NCName]
QName	::=	[http://www.w3.org/TR/REC-xml-names/#NT-QName]
Char	::=	[http://www.w3.org/TR/REC-xml/#NT-Char]

The complete grammar of LXPath is shown in Table 3 specified in Extended Backus-Naur Form (EBNF) notation with MatchPattern as entry point. An example for a properties style sheet can be found in Section 5.3.

5 Application: MPEG-21 BSD-based Digital Item Adaptation

MPEG-21 BSD-based adaptation [Ve04] represents a codec-agnostic adaptation approach utilizing XML-based BSDs and exploiting the characteristics of scalable coding formats. It has been adopted as part of the MPEG-21 multimedia framework and is briefly described in the following.

The characteristics of scalable coding formats enable the generation of a degraded version of the original media bitstream by means of simple remove operations followed by minor update operations, e.g., removal of spatial layers and updates of certain header information comprising the horizontal and vertical resolution. A BSD is an XML document which describes a (scalable) bitstream enabling its adaptation in a codec-agnostic way. Only the high-level bitstream structure is described, i.e., how it is organized in terms of packets, headers, or layers. The level of detail of this description depends on the bitstream characteristics and the application requirements. The Adaptation Quality of Service description (AQoS) describes how a media content (segment) needs to be adapted in order to correspond to the various usage environment situations, e.g., how many quality layers need to be dropped to correspond to the currently available network bandwidth.

5.1 State of the Art: Static Adaptation

Figure 5 depicts an adaptation server. The adaptation comprises an adaptation decision taking process resulting in an adaptation decision, which guides the BSD transformation. The transformed BSD then steers the bitstream generation process [VT05][TDV06]. The Adaptation Decision Taking Engine (ADTE) computes an adaptation decision based on the current usage environment description and the AQoS. This adaptation decision is the input to the BSD transformation process which transforms the BSD, e.g., by using standardized XML transformation languages such as XSLT. The bitstream generation process (BSDtoBin) parses the transformed BSD and generates the adapted media bitstream by using the bitstream offsets and parameter values of the remaining BSD elements. Only the bitstream segments described by the remaining BSD elements are copied to the output bitstream whereas all other segments are skipped. The output bitstream (and optionally its XML metadata) is then provided to a media consumer, e.g., an end device or a network node which performs further adaptation steps. Due to the fact that the BSD describes the complete bitstream, any adaptation which is performed always impacts the complete bitstream. No piece-wise adaptation to a dynamically changing usage environment is possible. Further disadvantages when applying this approach to streaming scenarios include:

- High memory requirements due to the need to parse the complete BSD into memory for the adaptation
- High startup delay in streaming scenarios, since any adaptation impacts the complete bitstream
- Slow reaction to dynamically changing usage environment in streaming scenarios, since any adaptation impacts the complete bitstream

5.2 Using Streaming Instructions to Enable Dynamic and Distributed Adaptation

This section describes and illustrates how the streaming instructions described above can be used to extend the static MPEG-21 DIA approach towards dynamic and distributed adaptation scenarios. Figure 6 depicts how we integrated the streaming instructions with the BSD-based adaptation approach in an adaptation server in order to enable dynamic and distributed adaptation. The BSD is provided, together with the XML streaming instructions, to the XML fragmenter. The fragmenter then determines the next PU from the BSD and assigns a time stamp to it, as described in Section 4.1. This PU is then transformed using the XSLT in the same way as a complete BSD would be transformed (as described in Section 5.1). The transformed PU is forwarded to the so-called BSDtoBinAU processor, which combines the functionality of the normative BSDtoBin processor and the media fragmenter. We decided to combine these two processors due to performance reasons. If the BSDtoBin processor cannot be modified, e.g., because it is implemented in hardware, the media fragmenter can be executed independently before the BSDtoBin processor. The BSDtoBinAU processor has the appropriate media AU and its time stamp available, thanks to the media streaming instructions. In the next step the BSDtoBinAU processor adapts the media AU order to correspond to the transformed PU. The transformed PUs, which are still represented in the text domain, are then encoded into AUs using a proper encoding mechanism. This can for example be a mechanism as basic as a general compression program such as WinZip or gzip. Another possibility would be to use XML-aware compression mechanisms such as XMLPPM [HAY06]. Another way to encode the PUs is to use a specific binary codec for XML such as the MPEG-7 Binary XML codec (BiM) [Ni02]. BiM is a schema-aware encoding mechanism which, if properly configured, removes any redundancy which exists between consecutive PUs. The redundancy, resulting from the requirement that PUs need to be able to be processed independently, is removed and only the new information is encoded into AUs (except for when a PU is declared as a RAP). Several studies have been performed on XML compression in the past [CW02][DB05][Su06]. In our own evaluations which also consider streaming support, BiM proved to be the most efficient way to encode

PUs [RTH05].

After encoding the PUs into BiM AUs, the media and BSD AUs are packetized for transport. In this step the timing information provided by media and XML streaming instructions is mapped onto the transport layer (RTP in our case), by including it into the packet header. Both the media and BSD AUs are then streamed into the network, where an adaptation proxy could perform additional adaptation steps or to an end device where the dynamically adapted media is consumed. In this case, the transport of the metadata may be omitted.

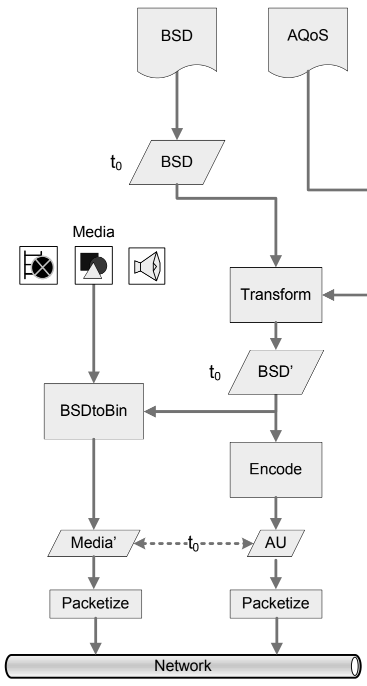


Figure 5: Static BSD-based Adaptation Approach

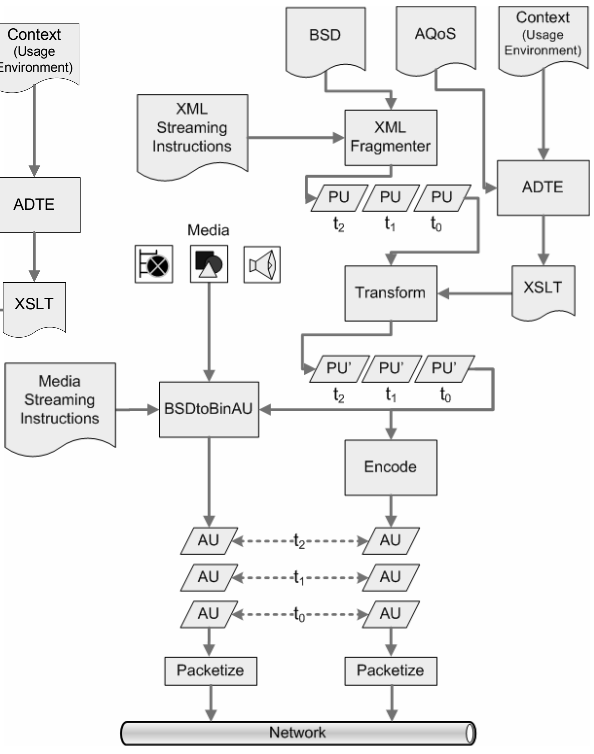


Figure 6: Dynamic BSD-based Adaptation Approach

Other content-related metadata which does not have fragmentation or timing requirements is not streamed but provided using other out-of-band mechanisms, e.g., as attributes in the Session Description Protocol (SDP) [SDP]. The normative behavior of the MPEG-21 DIA mechanisms is not changed by integrating the streaming instructions.

5.3 Example

In this section we provide example code for the mechanisms described above.

Document 2 shows an MPEG-21 DIA BSD which includes media and XML streaming instructions in order to enable dynamic processing of the BSD and the described media. In this example, each top-level `gBSDUnit` describes an AU of the MPEG-4 Scalable Video Codec [SMW06], including its start and length in bit (as indicated by the `addressUnit` attribute). As can be seen, the BSD already provides attributes for `addressUnit`, `start` and `length`. The fragmenter therefore uses the values in these attributes rather than duplicating them in the corresponding streaming instructions attributes. Within an AU, each `gBSDUnit` describes a single layer of the SVC stream. The layer is identified by the marker attribute value, which for the first layer of the second AU states that it is the first FGS layer of the first spatial layer which belongs to the first temporal layer (“`T0:S0:F0`”).

```
<dia:DIA xmlns:xmlsi="urn:mpeg:mpeg21:200x:01-SI"  xmlns:msi="urn:mpeg:mpeg21:200x:01-MSI"
 xmlns:dia="urn:mpeg:mpeg21:2003:01-DIA-NS" xmlns="urn:mpeg:mpeg21:2003:01-DIA-gBSD-NS"
 xmlns:bs1="urn:mpeg:mpeg21:2003:01-DIA-BSDL1-NS" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <dia:Description msi:timeScale="1000" msi:auMode="tree" xmlsi:timeScale="1000"
  xmlsi:puMode="ancestorsDescendants" xsi:type="gBSDType" addressUnit="bit" addressMode="Absolute"
  bs1:bitstreamURI="cdi_qcif_125_PARLIERsvc_201.raw">
  <gBSDUnit start="0" length="0" msi:dts="0" msi:cts="1280" msi:au="true" xmlsi:anchorElement="true"
   xmlsi:absTimeInt="0" msi :rap="true" xmlsi :rap="true">
   <gBSDUnit start="0" length="128" marker="T0:S0:F0"/>
   <!-- ... and so on ... -->
   <gBSDUnit start="22288" length="72" marker="T0:S0:F0"/>
  </gBSDUnit>
  <gBSDUnit start="22360" length="0" msi:dts="80" msi:cts="1360" msi:au="true" xmlsi:anchorElement="true"
   xmlsi:absTimeInt="80" msi :rap="true" xmlsi :rap="true">
   <gBSDUnit start="22360" length="1560" marker="T0:S0:F0"/>
   <gBSDUnit start="23920" length="6304" marker="T0:S0:F1"/>
   <gBSDUnit start="30224" length="10784" marker="T0:S0:F2"/>
   <gBSDUnit start="41008" length="72" marker="T1:S0:F0"/>
   <gBSDUnit start="41080" length="1920" marker="T1:S0:F0"/>
   <gBSDUnit start="43000" length="552" marker="T1:S0:F1"/>
   <gBSDUnit start="43552" length="3048" marker="T1:S0:F2"/>
   <!-- ... and so on ... -->
  </gBSDUnit>
  <gBSDUnit start="62992" length="0" msi:dts="1360" msi:cts="2640" msi:au="true"
   xmlsi:anchorElement="true" msi :rap="true" xmlsi :rap="true" xmlsi:absTimeInt="1360">
   <gBSDUnit start="62992" length="4456" marker="T0:S0:F0"/>
   <!-- ... and so on ... -->
  </gBSDUnit>
  <gBSDUnit start="107616" length="0" msi:dts="2640" msi:cts="3920" msi:au="true"
   xmlsi:anchorElement="true" xmlsi:absTimeInt="2640" msi :rap="true" xmlsi :rap="true">
   <gBSDUnit start="107616" length="1200" marker="T0:S0:F0"/>
   <!-- ... and so on ... -->
  </gBSDUnit>
  <!-- ... and so on ... -->
 </dia:Description></dia:DIA>
```

Document 2: Example of gBSD with streaming instructions

The streaming instructions are in bold. After declaring the namespaces which belong to the streaming instructions, the `timeScale`, `auMode` and `puMode` are specified in the `Description` element. The inheritance of these properties makes sure that they are valid for all `gBSDUnits` which are children of the `Description` element. In this application, the `ancestorsDescendants puMode` is used, which specifies that any PU consists of the element containing the `anchorElement` attribute and all its ancestors and descendants. The first resulting PU, when applying this fragmentation rule, can be seen in Document 4. Investigation of these documents shows that each document describes only a small part (in this case an AU) of the media bitstream. However, as we used the `ancestorsDescendants puMode`, the documents correspond to the requirement that a PU has to be well-formed and needs to be able to be consumed as such by a terminal. This allows us to use normative DIA mechanisms without the need to change them. These PUs are then provided to the BSDtoBinAU processor (which is a combination of the normative BSDtoBin processor and our media fragmenter), which extracts the AUs, as specified by the media streaming instructions and adapts them, as specified by MPEG-21 DIA.

Alternatively, the properties style sheet provided in Document 3 provides the streaming instructions externally, without changing the gBSD itself. As specified in Section 4.3 the properties style sheet consists of a sequence of templates specified by a matching pattern expressed in LXPath and containing a list of properties defined by a qualified name and a value. This properties style sheet sets the same attributes as shown in the example in Document 2.

```
<properties xmlns="urn:mpeg:mpeg21:200x:01-PS"  xmlns:msi="urn:mpeg:mpeg21:2003:01-DIA-MSI-NS"
 xmlns:xmlsi=" urn:mpeg:mpeg21:2003:01-DIA-XSI-NS" xmlns:dia="urn:mpeg:mpeg21:2003:01-DIA-NS"
 xmlns:gBSD="urn:mpeg:mpeg21:2003:01-DIA-gBSD-NS">
  <template match="/dia:DIA/dia:Description">
   <property name="msi:timeScale" value="1000"/><property name="msi:auMode" value="tree"/>
   <property name="xmlsi:timeScale" value="1000"/><property name="xmlsi:puMode"
    value="ancestorsDescendants"/>
  </template>
  <template match="/dia:DIA/dia:Description/gBSD:gBSDUnit">
   <property name="xmlsi:anchorElement" value="true"/><property name="xmlsi:rap" value="true"/>
   <property name="msi:au" value="true"/><property name="msi:rap" value="true"/>
  </template>
  <template match="/dia:DIA/dia:Description/gBSD:gBSDUnit[0]">
   <property name="msi:dts" value="0"/><property name="msi:cts" value="1280"/>
   <property name="xmlsi:absTimeInt" value="0"/>
  </template>
  <template match="/dia:DIA/dia:Description/gBSD:gBSDUnit[1]">
   <property name="msi:dts" value="80"/><property name="msi:cts" value="1360"/>
  <property name="xmlsi:absTimeInt" value="80"/>
 </template>
 <!-- … and so on … -->
</properties>
```

Document 3: Example of Properties Style Sheet

```
<dia:DIA < !-- … NS declarations ommited to save space … --> >
 <dia:Description msi:timeScale="1000" msi:auMode="tree" xmlsi:timescale="1000"
  xmlsi:puMode="ancestorsDescendants" xsi:type="gBSDType" addressUnit="bit" addressMode="Absolute"
  bs1:bitstreamURI="cdi_qcif_125_PARLIERsvc_201.raw">
  <gBSDUnit start="0" length="0" msi:dts="0" msi:cts="1280" msi:au="true" xmlsi:anchorElement="true"
   xmlsi:absTimeInt="0">
   <gBSDUnit start="0" length="128" marker="T0:S0:F0"/>
   <!-- ... and so on ... --></gBSDUnit></dia:Description></dia:DIA>
```

Document 4: First PU resulting from processing the gBSD in Document 2

6 Measurements

In order to validate our work, the system described in Section 5.2 was implemented in C++, together with the streaming instructions processors, i.e., the media and XML fragmenter. The libxml XMLTextReader interface[3] (an XML Pull Parser) was used for accessing the XML information. The aim of the measurements is to evaluate if our prototype implementation of a dynamic MPEG-21 adaptation node can be utilized in a real-time streaming scenario. To this end we first measure the performance of the streaming instructions processors and then we evaluate the CPU load and memory utilization of the complete adaptation node (depicted in Figure 8). All tests were performed on a Dell Optiplex GX620 desktop with an Intel Pentium D 2.8 GHz processor and 1024 MB of RAM using Windows XP SP2 as an operating system. Time measurements were performed using the ANSI-C clock method.

Table 4: Characteristics of Test Data

	MPEG-4 BSAC	EZBC	MPEG-4 SVC
Media Size	12511 KB	450536 KB	538816 KB
Average AU Size	0,22 KB	197,86 KB	18,59 KB
BSD Size	196265 KB	144939 KB	123189 KB
Average PU Size	4,02 KB	63,80 KB	4,90 KB
Number of [A|P]Us	56100	2277	28980
Resolution	N/A	QCIF	QCIF
Frame Rate	21	12,5	12,5
Length in Minutes	44,52	48,58	193,2

Table 4 provides an overview of the test data. Media and the corresponding BSDs for three different media codecs were selected. MPEG-4 BSAC [Pu99] is a scalable audio codec, EZBC [HW00] is a scalable video codec based on wavelets and MPEG-4 SVC [SMW06] is a scalable video codec based on conventional block transforms which is currently being

[3] libxml; http://xmlsoft.org

standardized in MPEG. The considerable size differences between the SVC and the EZBC content (both media and metadata) are due to the fact that the EZBC was encoded with 6 spatial layers and the SVC was encoded with only a single spatial layer. For our tests, the BSD is provided in the uncompressed domain and we consider that each PU describes exactly one AU. We used streaming instructions embedded into the BSD to specify the fragmentation mechanism for our measurements. All tests have been repeated 10 times in order to get accurate results.

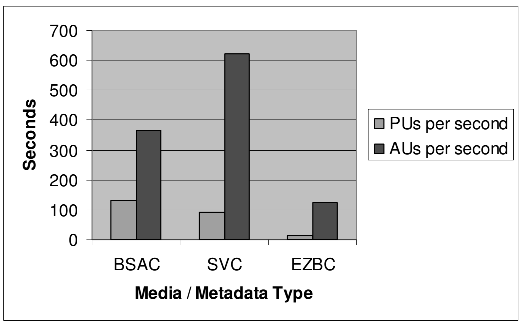
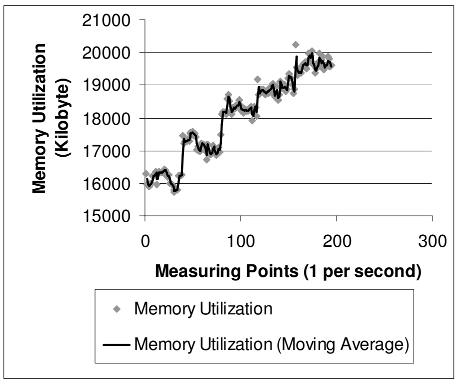


Figure 7: Streaming instructions: performance

For the XML fragmenter the measurements cover: 1) Access to the BSD from the file system, 2) Parsing the BSD using the libxml XMLTextReader, 3) Compose PUs, 4) Assign timing information to the PUs and 5) Encapsulate PUs and their timing into RTP packets. For the media fragmenter the measurements cover: 1) Access to the BSD from the file system, 2) Access to the media from the file system, 3) Parse the BSD using the libxml XMLTextReader, 4) Extract AUs, 5) Assign timing information to the AUs and 6) Output AUs and their timing to a file. Figure 7 shows the performance of the media and XML fragmenters. Considering that each EZBC AU describes 16 temporal layers (i.e., frames) and that each SVC AU describes 5 temporal layers, we can conclude that our prototype implementation offers good real-time performance.

Consequently we measured the performance of the complete adaptation server, as depicted in Figure 8. These measurements cover: 1) PU composition and AU extraction as measured above – except for step 5 (no file output), 2) BSD-based adaptation to each PU / AU (i.e., a) Compute an adaptation decision using the AQoS and the UED(s), b) Transform the BSD PU according to the adaptation decision, c) Scale (i.e., adapt) the media AU (e.g., discard enhancement layers) according to the transformed BSD PU, d) Update start and length information of the BSD PU according to the scaled media AU), 3) Packetize the media AU and the BSD PU into RTP packets and populate the RTP header with media and XML streaming instructions properties (e.g., timing, random access) and 4) Stream the packets into the network.

We measured the memory utilization and CPU load of our adaptation server. To this end, we access a single content (consisting of a media stream and a BSD), which is fragmented according to the streaming instructions, adapted, packetized and streamed to the player on the end device. We then access another content, and so on, until there are ten streams (five media streams and five BSD streams) being processed and delivered concurrently. Figure 8 shows the results of these tests for the SVC content. There is a single content being processed for the first 40 seconds. Then there are two contents until second 80. From second 80 to 120 we see three contents being processed concurrently. After second 120 there are four contents being processed and finally (after second 160) there are five contents being processed in parallel. At this number we had to finish our measurements, because the PC (a separate node) running the players could not support more instances of the player. With 10 concurrent streams, the memory utilization is at 20MB and the CPU load is at around 6%. As can be seen from the measurements, the adaptation server would have supported several more content streams (or contents with a higher bitrate).



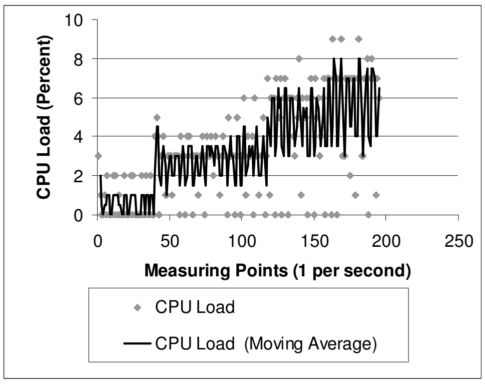


Figure 8: MPEG-21 based Dynamic DIA Adaptation of 1 to 5 QCIF SVC Streams: Memory Utilization and CPU Load

7 Summary and Future Work

In this paper, we addressed the problem of processing large metadata descriptions in streaming scenarios. To this end we introduced streaming instructions for fragmenting content-related metadata, associating the media segments and metadata fragments with each other, and streaming and processing them in a synchronized manner. The streaming instructions extend an XML metadata document by providing additional attributes to describe the fragmentation and timing of media data and XML metadata such as to enable their synchronized delivery and processing. In addition, a style sheet approach provides the opportunity to dynamically set such streaming properties without actually modifying the metadata themselves. We evaluated the implemented mechanisms both as “stand-alone” processors and integrated in a specific application scenario. We showed the usefulness of our work by implementing an adaptation node which uses our mechanisms to extend the static DIA approach to dynamic and distributed usage scenarios.

The streaming instructions have been proposed for inclusion in the MPEG-21 multimedia framework and are currently being considered as an amendment [DDA06].

Future work will include further evaluation of the streaming instructions for different types of metadata. This may lead to new PU modes and/or streaming instruction properties. The current synchronization mechanism relies on time stamps, implying a one-to-one relationship between media and metadata AUs. This is not optimal, since metadata AUs are usually much smaller than media AUs and the protocol overhead becomes considerable. A more flexible synchronization mechanism will be investigated. Research on the robustness of metadata channels is a logical next step, since enabling a single metadata AU to describe multiple media AUs makes the loss of a metadata AU a much more serious issue than in a one-to-one relationship.

References

[Fo06] P. Fox, D. McGuinness, R. Raskin, K. Sinha, "Semantically-Enabled Scientific Data Integration", Geoinformatics 2006, May, 2006

[AKS03] S. Atarashi, J. Kishigami, S. Sugimoto, "Metadata and new challenges", Symposium on Applications and the Internet Workshop, January, 2003

[Ve04] A. Vetro, "MPEG-21 Digital Item Adaptation: Enabling Universal Multimedia Access", IEEE Multimedia, pp. 84-87, January, 2004

[De06] D. Van Deursen, W. De Neve, D. De Schrijver, R. Van de Walle, "BFlavor: an optimized XML-based framework for multimedia content customization", 25th PCS, Beijing, April, 2006

[HE02] D. Hong, A. Eleftheriadis, "XFlavor: Bridging Bits and Objects in Media Representation", Proceedings, IEEE Int’l Conference on Multimedia and Expo (ICME), Lausanne, 2002

[BS06] W. Bailer, P. Schallauer, "The Detailed Audiovisual Profile: Enabling Interoperability between MPEG-7 Based Systems", Proceedings, 12th International Multi-Media Modeling Conference (MMM 2006), Beijing, January, 2006

[Si01] T. Sikora, "The MPEG-7 Visual Standard for Content Description – An Overview", IEEE Trans. on CSVT, vol. 11, no. 6, pp. 696-702, June, 2001

[BMF] ISO/IEC 14496-12:2005 Part 12: ISO Base Media File Format

[DPC05] K. Diepold, F. Pereira, W. Chang, "MPEG-A: Multimedia Application Formats", IEEE Multimedia, pp. 34-41, October, 2005

[WCL03] E. Y.C. Wong, A. T.S. Chan, H. Leong, "Semantic-based Approach to Streaming XML Contents Xstream", 27th Annual International Computer Software and Applications

Conference (COMPSAC 2003), Dallas, November, 2003

[Ni02] U. Niedermeier et al., "An MPEG-7 tool for compression and streaming of XML data", IEEE International Conference on Multimedia and Expo (ICME 02), Lausanne, August, 2002

[PPP04] S. Pfeiffer, C. Parker, A. Pang, "The Continuous Media Markup Language (CMML)", Internet Draft, IETF, March, 2004

[PPP05] S. Pfeiffer, C. Parker, A. Pang, "The Annodex exchange format for time-continuous bitstreams", Internet Draft, IETF, March, 2005

[Ru01] L. Rutledge, "SMIL 2.0: XML for Web Multimedia", IEEE Internet Computing, pp. 78-84, September, 2001

[Si03] F. Simeoni, D. Lievens, R. Connor, P. Manghi, "Language Bindings to XML", IEEE Internet Computing, pp. 19-27, January, 2003

[Qu03] A. Quint, "Scalable Vector Graphics", IEEE Multimedia, pp. 99-102, July, 2003

[XIS04] XML Information Set (Second Edition), W3C Recommendation, February, 2004

[VT05] A. Vetro, C. Timmerer, "Digital Item Adaptation: Overview of Standardization and Research Activities", IEEE Transactions on Multimedia, vol. 7, no. 3, pp. 418-426, June, 2005

[TDV06] C. Timmerer, S. Devillers, A. Vetro, "Digital Item Adaptation - Coding Format Independence, in: Ian Burnett, Fernando Pereira, Rik Van de Walle, Rob Koenen (eds.), The MPEG-21 Book", John Wiley and Sons Ltd, pp. 282-331, 2006

[STX04] Streaming Transformations for XML (STX), Version 1.0, Working Draft 1, July, 2004, http://stx.sourceforge.net/documents/

[XPL99] XML Path Language (XPath), Version 1.0, W3C Recommendation, November, 1999

[HAY06] S. Harrusi, A. Averbuch, A. Yehudai, "XML Syntax Conscious Compression", Data Compression Conference, pp. 402-411, March, 2006

[CW02] M. Cokus, D. Winkowski, "XML Sizing and Compression Study For Military Wireless Data", XML Conference & Exposition 2002, December, 2002

[DB05] S. J. Davis, I. S. Burnett, "Efficient Delivery within the MPEG-21 Framework", First International Conference on Automated Production of Cross Media Content for Multi-Channel Distribution, pp. 205-208, November, 2005

[Su06] R. D. Sutter, S. Lerouge, P. D. Neve, C. Timmerer, H. Hellwagner, R. V. d. Walle, "Comparison of XML serializations: cost benefit vs. complexity", ACM Multimedia Systems, vol. 12, no. 1, pp. 1-15, August, 2006

[RTH05] M. Ransburg, C. Timmerer, H. Hellwagner, “Transport Mechanisms for Metadata-driven Distributed Multimedia Adaptation”, First International Conference on Multimedia Access Networks (MSAN 2005), July, 2005

[SDP] SDP: Session Description Protocol, RFC2327

[Pu99] H. Purnhagen, "An Overview of MPEG-4 Audio Version 2", AES 17th International Conference on High-Quality Audio Coding, Firenze, Sep. 1999

[HW00] S.-T. Hsiang, J. W. Woods, "Embedded image coding using zeroblocks of subband/wavelet coefficients and context modelling", MPEG-4 Workshop and Exhibition at ISCAS 2000, Geneva, May, 2000

[DDA06] ISO/IEC 21000-7:2004/FPDAmd 2: Dynamic and Distributed Adaptation, 2006

[SMW06] H. Schwarz, D. Marpe, T. Wiegand, "Overview of the Scalable H.264/MPEG4-AVC Extension", International Conference on Image Processing (ICIP 2006), October, 2006