भारतीय मानक
Indian Standard

IS 16125 (Part 1) : 2014
ISO/IEC 15938-1 : 2002

सूचना प्रौद्योगिकी — मल्टीमीडिया अंश के वर्णन का इंटरफेस

भाग 1 प्रणालियाँ

Information Technology — Multimedia Content Description Interface

Part 1 Systems

ICS 35.040

भारतीय मानक ब्यूरो
BUREAU OF INDIAN STANDARDS
मानक भवन, 9 बहादुरशाह ज़फर मार्ग, नई दिल्ली–110002
MANAK BHAVAN, 9 BAHADUR SHAH ZAFAR MARG
NEW DELHI-110002
www.bis.org.in www.standardsbis.in

September 2014 **Price ₹ 2390.00**

Contents

Page

Software and Systems Engineering Sectional Committee, LITD 14

NATIONAL FOREWORD

This Indian Standard (Part 1) which is identical with ISO/IEC 15938-1 : 2002 'Information technology — Multimedia content description interface — Part 1: Systems' issued by the International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) jointly was adopted by the Bureau of Indian Standards on the recommendations of the Software and Systems Engineering Sectional Committee and approval of the Electronics and Information Technology Division Council.

This standard is one of the parts of a series of standard on 'Information technology — Multimedia content description interface'. The other part in this series is:

Part 2 Description definition language

Amendments No.1 and 2 issued in the year 2005 and 2006 to the above International Standard has been given at the end of this publication.

The text of ISO/IEC Standard has been approved as suitable for publication as an Indian Standard without deviations. Certain conventions are however not identical to those used in Indian Standards. Attention is particularly drawn to the following:

> Wherever the words 'International Standard' appear referring to this standard, they should be read as 'Indian Standard'.

The technical committee has reviewed the provisions of the following International Standard/Other Publications referred in this adopted standard and has decided that they are acceptable for use in conjunction with this standard:

International Standard/ Other Publication	*Title*
ISO/IEC 10646-1 : 2000	Information technology — Universal Multiple-Octet Coded Character Set (UCS) —Part 1: Architecture and Basic Multilingual Plane
XML	Extensible Markup Language (XML) 1.0 http://www.w3.org/TR/2000/REC-xml-20001006
XML Schema	W3C Recommendation http://www.w3.org/XML/Schema
XML Schema	Part 0 Primer, W3C Recommendation http://www.w3.org/TR/xmlschema-0/
XML Schema	Part 1 Structures, W3C Recommendation http://www.w3.org/TR/xmlschema-1/
XML Schema	Part 2 Datatypes, W3C Recommendation http://www.w3.org/TR/xmlschema-2/
XPath	XML Path Language, W3C Recommendation http://www.w3.org/TR/1999/REC-xpath-19991116
Namespaces in XML	W3C Recommendation http://www.w3.org/TR/1999/REC-xml-names-19990114
RFC 2396	Uniform Resource Identifiers (URI) — Generic Syntax.
IEEE Standard Std 754-1985	Binary Floating-Point Arithmetic

Introduction

This standard, also known as "Multimedia Content Description Interface," provides a standardized set of technologies for describing multimedia content. The standard addresses a broad spectrum of multimedia applications and requirements by providing a metadata system for describing the features of multimedia content.

The following are specified in this standard:

- **Description Schemes (DS)** describe entities or relationships pertaining to multimedia content. Description Schemes specify the structure and semantics of their components, which may be Description Schemes, Descriptors, or datatypes.
- **Descriptors (D)** describe features, attributes, or groups of attributes of multimedia content.
- **Datatypes** are the basic reusable datatypes employed by Description Schemes and Descriptors.
- **Description Definition Language (DDL)** defines Description Schemes, Descriptors, and Datatypes by specifying their syntax, and allows their extension.
- **Systems tools** support delivery of descriptions, multiplexing of descriptions with multimedia content, synchronization, file format, and so forth.

This standard is subdivided into eight parts:

Part 1 – Systems: specifies the tools for preparing descriptions for efficient transport and storage, compressing descriptions, and allowing synchronization between content and descriptions.

Part 2 – Description definition language: specifies the language for defining the standard set of description tools (DSs, Ds, and datatypes) and for defining new description tools.

Part 3 – Visual: specifies the description tools pertaining to visual content.

Part 4 – Audio: specifies the description tools pertaining to audio content.

Part 5 – Multimedia description schemes: specifies the generic description tools pertaining to multimedia including audio and visual content.

Part 6 – Reference software: provides a software implementation of the standard.

Part 7 – Conformance testing: specifies the guidelines and procedures for testing conformance of implementations of the standard.

Part 8 – Extraction and use of MPEG-7 descriptions: provides guidelines and examples of the extraction and use of descriptions.

Indian Standard

INFORMATION TECHNOLOGY — MULTIMEDIA CONTENT DESCRIPTION INTERFACE

PART 1 SYSTEMS

1 Scope

This International Standard defines a Multimedia Content Description Interface, specifying a series of interfaces from system to application level to allow disparate systems to interchange information about multimedia content. It describes the architecture for systems, a language for extensions and specific applications, description tools in the audio and visual domains, as well as tools that are not specific to audio-visual domains.

This part of ISO/IEC 15938 specifies system level functionalities for the communication of multimedia content descriptions. ISO/IEC 15938-1 provides a specification which will:

— enable development of ISO/IEC 15938 receiving sub-systems, called ISO/IEC 15938 Terminal, or Terminal in short, to receive and assemble possibly partitioned and compressed multimedia content descriptions

— provide rules for the preparation of multimedia content descriptions consisting of the tools specified in Parts 3, 4 and 5 of ISO/IEC 15938 for efficient transport and storage.

The decoding process within the ISO/IEC 15938 Terminal is normative. The rules mentioned provide guidance for the preparation and encoding of multimedia content descriptions without leading to a unique encoded representation of such descriptions.

This part of the MPEG-7 Standard is intended to be implemented in conjunction with other parts of the standard. In particular, MPEG-7 Part 1: Systems assumes some knowledge of Part 2: Description Definition Language (DDL) in its normative syntactic definitions of Descriptors and Description Schemes, as well as in the processing of schema and descriptions. The methods for obtaining the descriptions to which the encoding techniques in this part refer are defined in Parts 3, 4, and 5 of ISO/IEC 15938.

MPEG-7 is an extensible standard. The standard method of extending the standard beyond the Description Schemes provided in the standard is to define new ones in the DDL, and to make those DSs as accessible as the instantiated descriptions. Further details are available in Part 2.

2 Normative references

The following normative documents contain provisions which, through reference in this text, constitute provisions of this part of ISO/IEC 15938. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this part of ISO/IEC 15938 are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies. Members of ISO and IEC maintain registers of currently valid International Standards. The Telecommunication Standardization Bureau maintains a list of currently valid ITU-T Recommendations.

- ISO/IEC 10646-1:2000, *Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane*

NOTE The UTF-8 encoding scheme is described in Annex D of ISO/IEC 10646-1:2000.

- XML, *Extensible Markup Language (XML) 1.0,* 6 October 2000 <http://www.w3.org/TR/2000/REC-xml-20001006>
- *XML Schema, W3C Recommendation*, 2 May 2001 <http://www.w3.org/XML/Schema>
- *XML Schema Part 0: Primer,* W3C Recommendation, 2 May 2001 <http://www.w3.org/TR/xmlschema-0/>
- *XML Schema Part 1: Structures*, W3C Recommendation, 2 May 2001 <http://www.w3.org/TR/xmlschema-1/>
- *XML Schema Part 2: Datatypes*, W3C Recommendation, 2 May 2001 <http://www.w3.org/TR/xmlschema-2/>
- XPath, *XML Path Language,* W3C Recommendation, 16 November 1999 <http://www.w3.org/TR/1999/REC-xpath-19991116>
- *Namespaces in XML*, W3C Recommendation, 14 January 1999 <http://www.w3.org/TR/1999/REC-xml-names-19990114>

NOTE These documents are maintained by the W3C (http://www.w3.org).

- RFC 2396, *Uniform Resource Identifiers (URI): Generic Syntax.*
- *IEEE Standard for Binary Floating-Point Arithmetic*, Std 754-1985 Reaffirmed1990, http://standards.ieee.org/reading/ieee/std_public/description/busarch/754-1985_desc.html

3 Terms and definitions

3.1 Conventions

3.1.1 Naming convention

In order to specify data types, Descriptors and Description Schemes, this part of ISO/IEC 15938 uses constructs specified in ISO/IEC 15938-2, such as "element", "attribute", "simpleType" and "complexType". The names associated with these constructs are created on the basis of the following conventions:

If the name is composed of various words, the first letter of each word is capitalized. The rule for the capitalization of the first word depends on the type of construct and is described below.

— ***Element naming:*** the first letter of the first word is capitalized (e.g. *TimePoint* element of *TimeType*).

— ***Attribute naming:*** the first letter of the first word is **not** capitalized (e.g. *timeUnit* attribute of *IncrDurationType*).

— ***complexType naming:*** the first letter of the first word is capitalized, the suffix "Type" is used at the end of the name.

— ***simpleType naming:*** the first letter of the first word is not capitalized, the suffix "Type" may be used at the end of the name.

3.1.2 Documentation convention

3.1.2.1 Textual syntax

The syntax of each XML schema item is specified using the constructs specified in ISO/IEC 15938-2. It is depicted in this document using a specific font and background, as shown in the example below:

```
<complexType name="ExampleType">
   <sequence>
      <element name="Element1" type="string"/>
   </sequence>
   <attribute name="attribute1" type="string" default="attrvalue1"/>
</complexType>
```

Non-normative XML examples are included in separate subclauses. They are depicted in this document using a separate font and background than the normative syntax specifications, as shown in the example below:

```
<Example attribute1="example attribute value">
   <Element1>example element content</Element1>
</Example>
```

3.1.2.2 Binary syntax

3.1.2.2.1 Overview

The binary description stream retrieved by the decoder is specified in Clause 7 and Clause 8. Each data item in the binary description stream is printed in bold type. It is described by its name, its length in bits, and by a mnemonic for its type and order of transmission. The construct “N+” in the length field indicates that the length of the element is an integer multiple of N.

The action caused by a decoded data element in a bitstream depends on the value of the data element and on data elements that have been previously decoded. The following constructs are used to express the conditions when data elements are present:

while (condition) {
data_element
. . .
}
	If the condition is true, then the group of data elements occurs next in the data stream. This repeats until the condition is not true.

do { **data_element** . . . } while (condition)	The data element always occurs at least once. The data element is repeated until the condition is not true.

if (condition) {
data_element
. . .
} else {
data_element
. . .
}
	If the condition is true, then the first group of data elements occurs next in the data stream.
	
	If the condition is not true, then the second group of data elements occurs next in the data stream.

for (i = m; i < n; i++) {
data_element
. . .
}
	The group of data elements occurs (n-m) times. Conditional constructs within the group of data elements may depend on the value of the loop control variable i, which is set to m for the first occurrence, incremented by one for the second occurrence, and so forth.

/* comment */	Explanatory comment that may be deleted entirely without in any way altering the syntax.

This syntax uses the 'C-code' convention that a variable or expression evaluating to a non-zero value is equivalent to a condition that is true and a variable or expression evaluating to a zero value is equivalent to a condition that is false.

Use of function-like constructs in syntax tables

In some syntax tables, function-like constructs are used in order to pass the value of a certain syntax element or decoding parameter down to a further syntax table. In that table, the syntax part is then defined like a function in e.g. C program language, specifying in brackets the type and name of the passed syntax element or decoding parameter, and the returned syntax element type, as shown in the following example:

datatype Function(datatype parameter_name) {	**Number of bits**	**Mnemonic**
if (parameter_name == ...) {		
OtherFunction(parameter_name)		
} else if		
.....		
} else {		
.....		
}		
Return return_value		
}		

Here, the syntax table describing the syntax part called “Function” receives the parameter “parameter_name” which is of datatype “datatype”. The parameter “parameter_name” is used within this syntax part, and it can also be passed further to other syntax parts, in the table above e.g. to the syntax part “OtherFunction”.

The parsing of the binary syntax is expressed in procedural terms. However, it should not be assumed that Clause 7 and 8 implement a complete decoding procedure. In particular, the binary syntax parsing in this specification assumes a correct and error-free binary description stream. Handling of erroneous binary description streams is left to individual implementations.

Syntax elements and data elements are depicted in this document using a specific font such as the following example: `FragmentUpdatePayload`.

boolean

In some syntax tables, the “true” and “false” constructs are used. If present in the stream “true” shall be represented with a single bit of value "1" and "false" shall be represented with a single bit of value "0".

3.1.2.2.2 Arrays

Arrays of data elements are represented according to the C-syntax as described below. It should be noted that each index of an array starts with the value “0”.

data_element[n]	is the n+1th element of an array of data.
data_element[m][n]	is the m+1, n+1th element of a two-dimensional array of data**.**
data_element[l][m][n]	is the l+1, m+1, n+1th element of a three-dimensional array of data.

3.1.2.2.3 Functions

3.1.2.2.3.1 nextByteBoundary()

The function “nextByteBoundary()” reads and consumes bits from the binary description stream until but not including the next byte-aligned position in the binary description stream.

3.1.2.2.4 Reserved values and forbidden values

The terms "reserved" and "forbidden" are used in the description of some values of several code and index tables.

The term "reserved" indicates that the value shall not occur in a binary description stream. It may be used in the future for ISO/IEC defined extensions.

The term "forbidden" indicates a value that shall not occur in a binary description stream.

3.1.2.2.5 Reserved bits and stuffing bits

ReservedBits: a binary syntax element whose length is indicated in the syntax table. The value of each bit of this element shall be “1”. These bits may be used in the future for ISO/IEC defined extensions.

Stuffing bits: bits inserted to align the binary description stream, for example to a byte boundary. The value of each of these bits in the binary description stream shall be “1”.

3.1.2.3 Textual and binary semantics

The semantics of each schema or binary syntax component, is specified using a table format, where each row contains the name and a definition of that schema or binary syntax component:

Name	*Definition*
ExampleType	Specifies an ...
element1	Describes the …
attribute1	Describes the …

3.2 Definitions

3.2.1
access unit
An entity within a description stream that is atomic in time, i.e., to which a composition time can be attached. An access unit is composed of one or more fragment update units.

3.2.2
application
An abstraction of any entity that makes use of the decoded description stream.

3.2.3
binary access unit
An access unit in binary format as specified in Clause 7 and 8.

3.2.4
binary description stream
A concatenation of binary access units as specified in Clause 7 and 8.

3.2.5
binary format description tree
The internal binary decoder model.

3.2.6
byte-aligned
A bit in a binary description stream is byte-aligned if its position is a multiple of 8-bits from the first bit in the binary description stream.

3.2.7
composition time
The point in time when a specific access unit becomes known to the application.

3.2.8
content particle
A particle is a term in the XML Schema grammar for element content, consisting of either an element declaration, a wildcard or a model group, together with occurrence constraints. Refers to ISO/IEC 15938-2.

3.2.9
context mode
Information in the fragment update context specifying how to interpret the subsequent context path information.

3.2.10
context node
The context node is specified by the context path of the current fragment update context. It is the parent of the operand node.

3.2.11
context path
Information that identifies and locates the context node and the operand node in the current description tree.

3.2.12
current context node
The starting node for the context path in case of relative addressing.

3.2.13
current description
The description that is conveyed by the initial description and all access units up to a given composition time.

3.2.14
current description tree
The description tree that represents the current description.

3.2.15
DDL parser
An application that is capable of validating description schemes (content and structure) and descriptor data types against their schema definition.

3.2.16
delivery layer
An abstraction of any underlying transport or storage functionality.

3.2.17
derived type
A type defined by the derivation of an other type.

3.2.18
described time
A point in time or range of time, embedded in the description, that is related to the media described by the description. Note that there is no intrinsic relation between the described time and the composition time of an access unit. This information is e.g. carried by instances of the *MediaTimeType* defined in ISO/IEC 15938-5.

3.2.19
description
Short term for multimedia content description.

3.2.20
description composer
An entity that reconstitutes the current description tree from the fragment update units.

3.2.21
description fragment
A contiguous part of a description attached at a single node. Using the representation model of a description tree, the description fragment is represented by a sub-tree of the description tree.

3.2.22
description stream
The ordered concatenation of either binary or textual access units conveying a single, possibly time-variant, multimedia content description.

3.2.23
description tree
A model that is used throughout this specification in order to represent descriptions. A description tree consists of nodes, which represent elements or attributes of a description. Each node may have zero, one or more child nodes. Simple content are considered as child nodes in Clause 7 of the specification.

3.2.24
effective content particle
The particle of a complexType used for the validation process.

3.2.25
fragment update command
A command within a fragment update unit expressing the type of modification to be applied to the part of the current description tree that is identified by the associated fragment update context.

3.2.26
fragment update component extractor
An entity that de-multiplexes a fragment update unit, resulting in the unit's components: fragment update command, fragment update context, and fragment update payload.

3.2.27
fragment update context
Information in a fragment update unit that specifies on which node in the current description tree the fragment update command shall be executed. Additionally, the fragment update context specifies the data type of the element encoded in the subsequent fragment update payload.

3.2.28
fragment update payload
Information in a fragment update unit that conveys the information which is added to the current description or which replaces a part of the current description.

3.2.29
fragment update payload decoder
The entity that decodes the fragment update payload information of the fragment update.

3.2.30
fragment update unit
Information in an access unit, conveying a description or a portion thereof. Fragment update units provide the means to modify the current description. They are nominally composed of a fragment update command, a fragment update context and a fragment update payload.

3.2.31
fragment update decoder parameters
Configuration parameters conveyed in the `DecoderInit` (see 6.2 and 7.2) that are required to specify the decoding process of the fragment update decoder.

3.2.32
initial description
A description that initialises the current description tree without conveying it to the application (see 5.3). The initial description is part of the `DecoderInit` (see 6.2 and 7.2).

3.2.33
initialisation extractor
An entity that de-multiplexes the `DecoderInit` (see 6.2 and 7.2), resulting in its components initial description, fragment update decoder parameters and schema URI.

3.2.34
multimedia content description
The description of audiovisual data content in multimedia environments using the tools and elements provided by the parts 2, 3, 4 and 5 of ISO/IEC 15938.

3.2.35
operand node
The node in the binary format description tree that is either added, deleted or replaced according to the current fragment update command and fragment update payload. The operand node is always a child node of the context node.

3.2.36
schema
A schema is represented in XML by one or more "schema documents", that is, one or more "<schema>" element information items. A "schema document" contains representations for a collection of schema components, e.g. type definitions and element declarations, which have a common target namespace. A schema document which has one or more "<import>" element information items corresponds to a schema with components with more than one target namespace. Refer also to ISO/IEC 15938-2.

3.2.37
schema resolver
An entity that is capable of resolving the schema identification provided in the `DecoderInit` (see 6.2 and 7.2), and to possibly retrieve the specified schemas.

3.2.38
schema URI
A URI that uniquely identifies a schema.

3.2.39
schema valid
A description that is *schema valid* satisfies the constraints embodied in the Schema to which it should conform.

3.2.40
selector node
The parent node of the topmost node of a description tree. It artificially extends the description tree to allow the addressing of the topmost node.

3.2.41
super type
The parent of a type in its type hierarchy.

3.2.42
systems layer
An abstraction of the tools and processes specified by this part of ISO/IEC 15938.

3.2.43
terminal
The entity that makes use of a coded representation of a multimedia content description.

3.2.44
textual access unit
An access unit in textual format as specified in Clause 6.

3.2.45
textual description stream
A concatenation of textual access units as specified in Clause 6.

3.2.46
topmost node
The node specified by the first element in the description, instantiating one of the global elements declared in the schema.

3.2.47
type hierarchy
The hierarchy of type derivations.

3.3.48
validation
The process of parsing an XML document to determine whether it satisfies the constraints embodied in the Schema to which it should conform.

4 Symbols and abbreviated terms

4.1 Abbreviations

AU	Access Unit
BiM	Binary format for multimedia description streams
D	Descriptor
DDL	Description Definition Language
DL	Delivery Layer
DS	Description Scheme
FU	Fragment Update
FUU	Fragment Update Unit
FSAD	Finite State Automaton Decoder
MPC	Multiple element Position Code
SBC	Schema Branch Code
SPC	Single element Position Code
TBC	Tree Branch Code
TeM	Textual format for multimedia description streams
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UTF	Universal Character Set Transformation Formats
XML	Extensible Markup Language
XPath	XML Path Language

4.2 Mathematical operators

The mathematical operators used to describe this part of ISO/IEC 15938 are similar to those used in the C programming language. However, integer divisions with truncation and rounding are specifically defined. Numbering and counting loops generally begin from zero.

4.2.1 Arithmetic operators

+	Addition.
-	Subtraction (as a binary operator) or negation (as a unary operator).
++	Increment. i.e. x++ is equivalent to x = x + 1
- -	Decrement. i.e. x-- is equivalent to x = x - 1

*	Multiplication.
^	Power.
sign( )	$sign(x) = \begin{cases} 1 & x \geq 0 \\ -1 & x < 0 \end{cases}$
abs( )	$abs(x) = x \cdot sign(x)$
log2(..)	$\log 2(x) = \log_2(x)$
ceil(..)	$ceil(x) = \begin{cases} \text{int}(x) + 1 & x \geq 0 \\ \text{int}(x) & x < 0 \end{cases}$
int(..)	truncation of the argument to its integer value, e.g. 1.3 is truncated to 1 and –3.7 is truncated to –3.
$\sum_{i=a}^{i<b} f(i)$	the summation of the f(i) with i taking integral values from a up to, but not including b.

4.2.2 Logical operators

||	Logical OR.
&&	Logical AND.
!	Logical NOT.

4.2.3 Relational operators

>	Greater than.
>=	Greater than or equal to.
<	Less than.
<=	Less than or equal to.
==	Equal to.
!=	Not equal to.
max (, ...,)	the maximum value in the argument list.
min (, ... ,)	the minimum value in the argument list.

4.2.4 Assignment

=	Assignment operator.

4.2.5 Character string comparison

Many phases of the fragment encoding rely on a string comparison method. This method is based on the Unicode value of each character in the strings. The following defines the notion of lexicographic ordering:

Two strings are different if they have different characters at some index that is a valid index for both strings, or if their lengths are different, or both.

If they have different characters at one or more index positions, let k be the smallest such index; then the string whose character at position k has the smaller value, as determined by using the < operator, lexicographically precedes the other string.

If there is no index position at which they differ, then the shorter string lexicographically precedes the longer string.

This string comparison is described by each method that is functionally equivalent to the following procedure:

```
compare_strings(string1, string2) {
  len1 = length(string1);
  len2 = length(string2);
  n = min(len1, len2);
  i = 0;
  j = 0;

  while (n-- != 0) {
      c1 = string1[i++];
      c2 = string2[j++];
      if (c1 != c2) {
     return c1 - c2;
      }
  }
  return len1 - len2;
}
```

4.3 Mnemonics

The following mnemonics are defined to describe the different data types used in the coded bitstream.

Name	*Definition*
bslbf	Bit string, left bit first, where "left" is the order in which bit strings are written in this part of ISO/IEC 15938. Bit strings are generally written as a string of 1s and 0s within single quote marks, e.g. '1000 0001'. Blanks within a bit string are for ease of reading and have no significance. For convenience large strings are occasionally written in hexadecimal, in this case conversion to a binary in the conventional manner will yield the value of the bit string. Thus the left most hexadecimal digit is first and in each hexadecimal digit the most significant of the four bits is first.

uimsbf	Unsigned integer, most significant bit first.
vlclbf	Variable length code, left bit first, where "left" refers to the order in which the VLC codes are written. The byte order of multibyte words is most significant byte first.
vluimsbf8	Variable length code unsigned integer, most significant bit first. The size of vluimsbf8 is a multiple of one byte. The first bit (Ext) of each byte specifies if set to 1 that another byte is present for this vluimsbf8 code word. The unsigned integer is encoded by the concatenation of the seven least significant bits of each byte belonging to this vluimsbf8 code word An example for this type is shown in Figure 1.
vluimsbf5	Variable length code unsigned integer, most significant bit first. The first n bits (Ext) which are 1 except of the n-th bit which is 0, indicate that the integer is encoded by n times 4 bits. An example for this type is shown in Figure 2.

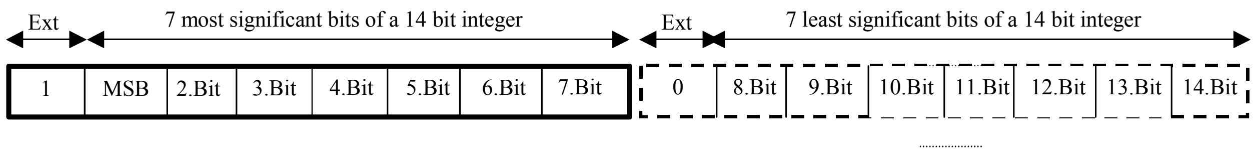

Ext
7 most significant bits of a 14 bit integer
Ext
7 least significant bits of a 14 bit integer
1
MSB
2.Bit
3.Bit
4.Bit
5.Bit
6.Bit
7.Bit
0
8.Bit
9.Bit
10.Bit
11.Bit
12.Bit
13.Bit
14.Bit

Figure 1 — Informative example for the vluimsbf8 data type

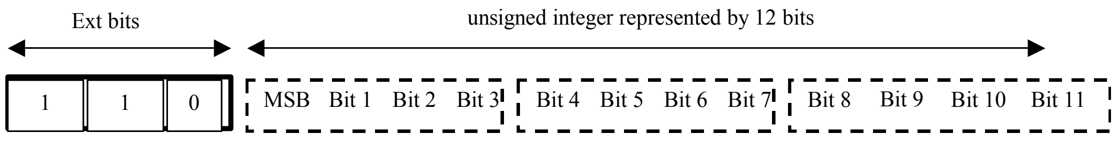

Ext bits
unsigned integer represented by 12 bits
1
1
0
MSB Bit 1 Bit 2 Bit 3
Bit 4 Bit 5 Bit 6 Bit 7
Bit 8 Bit 9 Bit 10 Bit 11

Figure 2 — Informative example for the vluimsbf5 data type

5 System architecture

5.1 Terminal architecture

ISO/IEC 15938 provides the means to represent coded multimedia content descriptions. The entity that makes use of such coded representations of the multimedia content description is generically referred to as the "ISO/IEC 15938 terminal" or just "terminal" in short. This terminal may correspond to a standalone application or be part of an application system.

This and the following three subclauses provide the description of an ISO/IEC 15938 terminal, its components, and their operation. The architecture of such a terminal is depicted in Figure 3. The following subclauses introduce the tools specified in this part of the specification.

In Figure 3, there are three main layers outlined: the application, the normative systems layer, and the delivery layer. ISO/IEC 15938-1 is not concerned with any storage and/or transmission media (whose behaviours and characteristics are abstracted by the delivery layer) or the way the application processes the current description. This specification does make specific assumptions about the delivery layer, and those assumptions are outlined in subclause 5.5.4. The systems layer, which is the subject of this part of ISO/IEC 15938, defines a decoder whose architecture is described here to provide an overview and to establish common terms of reference. A compliant decoder need not implement the constituent parts as visualised in Figure 3, but shall implement the normative decoding process specified in Clauses 6 through 8.

5.2 General characteristics of the decoder

5.2.1 General characteristics of description streams

An ISO/IEC 15938 terminal consumes description streams and outputs a – potentially dynamic – representation of the description called the current description tree. Description streams shall consist of a sequence of one or more individually accessible portions of data named access units. An Access Unit (AU) is the smallest data entity to which "terminal-oriented" (as opposed to "described-media oriented") timing information can be attributed. This timing information is called the "composition" time, meaning the point in time when the resulting current description tree corresponding to a specific access unit becomes known to the application. The timing information shall be carried by the delivery layer (see subclause 5.5.4). The current description tree shall be schema-valid after processing each access unit.

A description consisting of textual access units is termed a textual description stream and is processed by a textual decoder (see subclause 5.2.2 and clause 6). A description stream consisting of binary access units is termed a binary description stream and is processed by a binary decoder (see subclause 5.2.3 and Clauses 7 and 8). A mixture of both formats in a single stream is not permitted. The choice of either binary or textual format for the description stream is application dependent. Any valid ISO/IEC 15938 description, with the exception of those listed in 5.6.4, may be conveyed in either format.

5.2.2 Principles of the textual decoder (informative)

The ISO/IEC 15938-1 method for textual encoding, called TeM, enables the dynamic and/or progressive transmission of descriptions using only text. The original description, in the form of an XML document, is partitioned into fragments (see 5.5.1) that are wrapped in further XML code so that these resulting AUs can be individually transported (e.g. streamed or sent progressively). The decoding process for these AUs does not require any schema knowledge. The resulting current description tree may be byte-equivalent to the original description if desired by the encoder, but it may also exhibit dynamic characteristics such that certain parts of the description are present at the decoder only at chosen times, are never present at all, or appear in a different part of the tree.

5.2.3 Principles of the binary decoder (informative)

Using the ISO/IEC 15938-1 generic method for binary encoding, called BiM, a description (nominally in a textual XML form) can be compressed, partitioned, streamed, and reconstructed at terminal side. The reconstructed XML description will not be byte-equivalent to the original description. Namely, the binary encoding method does not preserve processing instructions, attribute order, comments, or non-significant whitespace. However, the encoding process ensures that XML element order is preserved.

The BiM, in order to gain its compression efficiency, relies on a schema analysis phase. During this phase, internal tables are computed to associate binary code to XML elements, types and attributes. This principle mandates the full knowledge of the same schema by the decoder and the encoder for maximum interoperability.

As with the textual decoder, the resulting current description tree may be topologically equivalent to the original description if desired by the encoder, but it may also exhibit dynamic characteristics such that certain parts of the description are present at the decoder only at chosen times, are never present at all, or appear in a different part of the tree.

5.3 Sequence of events during decoder initialisation

The decoder set-up is signalled by the initialisation extractor receiving a textual or binary `DecoderInit` (specified in 6.2 and 7.2). The signalling of the use of either binary or textual encoding is outside the scope of this specification. However, if the `DecoderInit` is binary, then the following description stream shall consist of binary access units. Similarly, if the `DecoderInit` is textual, then the following description stream shall consist of textual access units. The `DecoderInit` shall be received by the systems layer from the delivery layer. The `DecoderInit` will typically be conveyed by a separate delivery channel compared to the description stream, which is also received from the delivery layer. The component parts of the description stream are discussed in subclause 5.4.

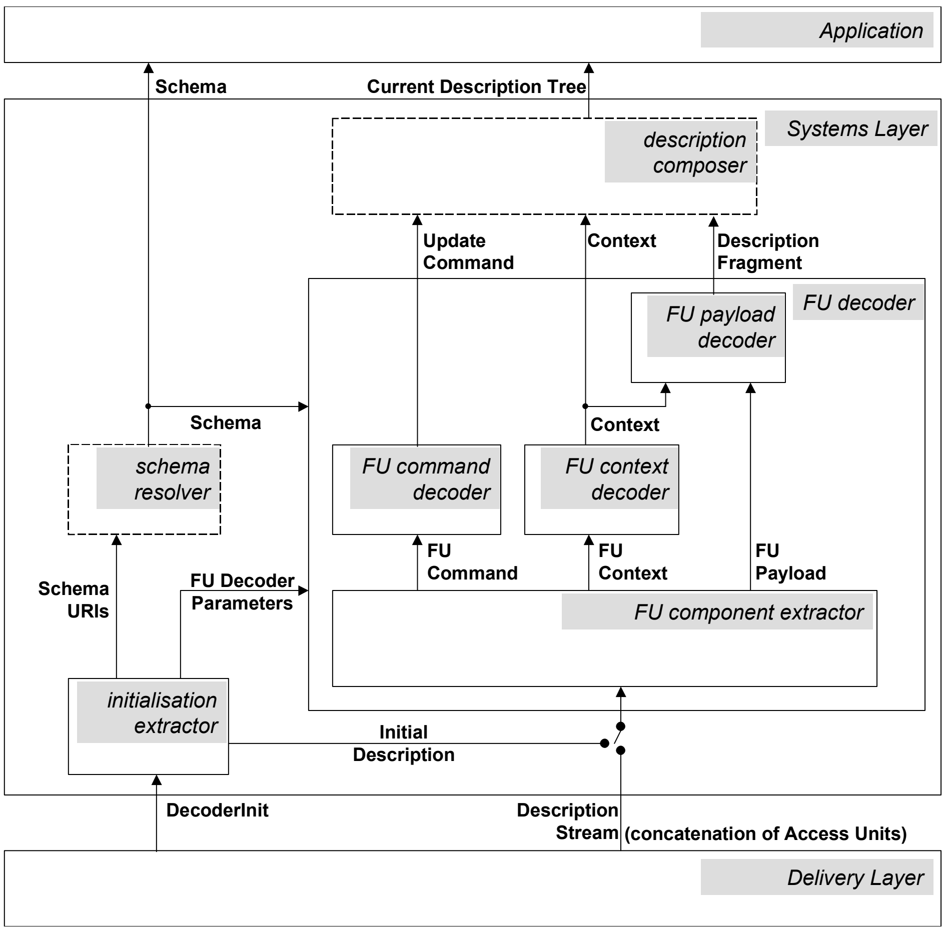


Figure 3 — Terminal Architecture
Dashed boxes in the systems layer are non-normative. FU is an abbreviation for Fragment Update.

The `DecoderInit` contains a list of URIs that identifies schemas, miscellaneous parameters to configure the decoder (FU Decoder Parameters, in Figure 3), and an initial description. There shall be only one `DecoderInit` per description stream. The list of URIs (Schema URIs, in Figure 3) is passed to a schema resolver that associates the URIs with schemas to be passed into the fragment update decoder (see subclauses 6.2 and 7.2). The schema resolver is non-normative and may, for example, retrieve schema documents from a network or refer to pre-stored schemas. The resulting schemas are used by the binary decoder specified in Clauses 7 and 8 and by any textual DDL parser that may be used for schema validation. If a given Schema URI is unknown to the schema resolver, the corresponding data types in a description stream shall be ignored (i.e., “skipped” and not processed).

The initial description has the same general syntax and semantics as an access unit, but with restrictions, as described in subclauses 6.2 and 7.2. The initial description initialises the current description tree without conveying it to the application. The current description tree is then updated by the access units that comprise the description stream. The initial description may be empty, since a schema-valid current description tree for consumption by the application need only be generated after the first access unit is decoded.

5.4 Decoder behaviour

The description stream shall be processed only after the decoder is initialised. The behaviour of the decoder when access units are received before the decoder is initialised is non-normative. Specifically, there is no requirement to buffer such “early AUs.”

An access unit is composed of any number of fragment update units, each of which is extracted in sequence by the fragment update component extractor. Each fragment update unit consists of:

— a fragment update command that specifies the type of update to be executed (i.e., add, replace or delete content or a node, or reset the current description tree);

— a fragment update context that identifies the data type in a given schema document, and points to the location in the current description tree where the fragment update command applies; and

— a fragment update payload conveying the coded description fragment to be added or replaced.

A fragment update extractor splits the fragment update units from the access units and emits the above component parts to the rest of the decoder. The fragment update command decoder generally consists of a simple table lookup for the update command to be passed on to the description composer. The decoded fragment update context information (‘context’ in Figure 3) is passed along to both the description composer and the fragment update payload decoder. The fragment update payload decoder embodies the BiM Payload decoder (Clause 8) or, in the case of the TeM, a DDL parser, which decodes a fragment update payload (aided by context information) to yield a description fragment (see Figure 3).

The corresponding update command and context are processed by the non-normative description composer, which either places the description fragment received from the fragment update payload decoder at the appropriate node of the current description tree at composition time, or sends a reconstruction event containing this information to the application. The actual reconstruction of the current description tree by the description composer is implementation-specific, i.e., the application may direct the description composer to prune or ignore unwanted elements as desired. There is no requirement on the format of this current description tree, e.g. it may remain a binary representation.

5.5 Issues in encoding descriptions

5.5.1 Fragmenting descriptions

A description stream serves to convey a multimedia content description, as available from a (non-normative) sender or encoder, to the receiving terminal, possibly by incremental transmission in multiple access units. Any number of decompositions of the source description may be possible and it is out of scope of this specification to define such decompositions. Figure 4 illustrates an example of a description, consisting of a number of nodes, that is broken into two description fragments.

If multiple description fragments corresponding to a specific node of the description are sent (e.g., a node is replaced) then the previous data within the nodes of the description represented by that description fragment become unavailable to the terminal. Replacing a single node of the description shall effectively overwrite all children of that node.

NOTE If an application wishes to retain such updated node information, it may do so. However, access to such outdated portions of the description is outside the scope of this specification.

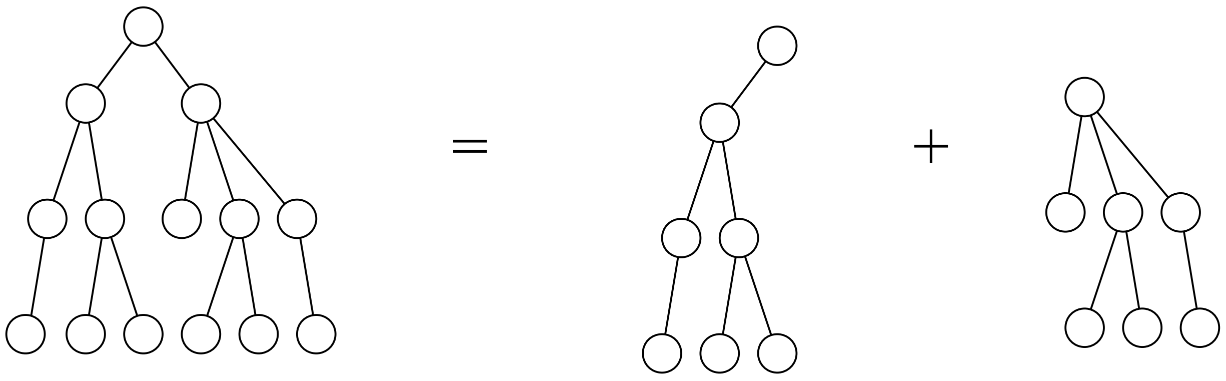


Figure 4 — Decomposition of a description into two description fragments

5.5.2 Deferred nodes and their use

With both the TeM and the BiM, there exists the possibility for the encoder to indicate that a node in the current description tree is "Deferred." A deferred node shall not contain content, but shall have a type associated with it. A deferred node is addressable on the current description tree (there is a fragment update context that unambiguously points to it), but it shall not be passed on to any further processing steps, such as a parser or an application. In other words, a deferred node is a placeholder that is rendered "invisible" to subsequent processing steps.

The typical use of deferred nodes by the encoder is to establish a desired tree topology without sending all nodes of the tree. Nodes to be sent later are marked as "deferred" and are therefore hidden from a parser. Hence, the current description tree minus any deferred nodes must be schema-valid at the end of each access unit. The deferred nodes may then be replaced in any subsequent access unit without changing the tree topology maintained internally in the decoder. However, there is no guarantee that a deferred node will ever be filled by a subsequent fragment update unit within the description stream.

5.5.3 Managing schema version compatibility with ISO/IEC 15938-1

It is very conceivable that a given schema will be updated during its lifetime. Therefore, ISO/IEC 15938-1 provides, with some constraints, interoperability between different versions of ISO/IEC 15938 schema definitions, without the full knowledge of all schema versions being required.

Two different forms of compatibility between different versions of schema are distinguished. In both cases, it is assumed that the updated version of a schema imports the previous version of that schema. Backward compatibility means that a decoder aware of an updated version of a schema is able to decode a description conformant to a previous version of that schema. Forward compatibility means that a decoder only aware of a previous version of a schema is able to partially decode a description conformant to an updated version of that schema.

With both the textual and binary format, backward compatibility is provided by the unique reference of the used schema in the `DecoderInit` using its Schema URI as its namespace identifier.

When using the binary format, forward compatibility is ensured by a specific syntax defined in Clause 7 and 8. Its main principle is to use the namespace of the schema, i.e., the Schema URI, as a unique version identifier. The binary format allows one to keep parts of a description related to different schema in separate chunks of the binary description stream, so that parts related to unknown schema may be skipped by the decoder. In order for this approach to work, an updated schema should not be defined using the ISO/IEC 15938-2 “redefine” construct but should be defined in a new namespace. The Decoder Initialisation identifies schema versions with which compatibility is preserved by listing their Schema URIs. A decoder that knows at least one of the Schema URIs will be able to decode at least part of the binary description stream.

5.5.4 Reference consistency (informative)

The standard itself cannot guarantee reference (link) consistency in all cases. In particular, XPath-style references cannot be guaranteed to point to the correct node, especially when the topology of the tree changes in a dynamic or progressive transmission environment. With ID/IDRef, the system itself cannot guarantee that the ID element will be present, but during the validation phase, all such links are checked, and thus their presence falls under the directive that the current description tree must always be schema-valid. URI and HREF links are typically to external documents, and should be understood not to be under control by the referrer (and therefore not guaranteed).

5.6 Differences between the TeM and BiM

5.6.1 Introduction

BiM and TeM are two similar methods to fragment and convey descriptions as a description stream. While both methods allow one to convey arbitrary descriptions conformant to Parts 3 – 5 of this specification, structural differences in the TeM- and BiM-encoded representation of the description as well as in the decoding process exist.

5.6.2 Use of schema knowledge

The TeM does not require schema knowledge to reconstitute descriptions; hence, the context information identifying the operand node on which the fragment update command is applied is generated with reference to the current description tree as available to the decoder before processing the current fragment update. The TeM operates on an instantiation-based model: one begins with a blank slate (a single selector node) and adds instantiated nodes as they are presented to the terminal. Schema knowledge is, of course, necessary for schema validation to be performed.

The BiM relies upon schema knowledge, i.e., the FU decoder implicitly knows about the existence and position of all potential elements as defined by the schema, no matter whether the corresponding elements have actually been received in the instantiated description. This shared knowledge between encoder and decoder improves compression of the context information and makes the context information independent from the current description tree as available to the decoder. The BiM operates on a schema-based model: all possibilities defined by the schema can be unambiguously addressed using the context information, and as a payload is added, the instantiation of the addressed node is noted. The current description tree is built by the set of all of the instantiated nodes. One non-obvious consequence of this BiM model is that numbering in the internal binary decoder model is “sticky”: once an element is instantiated and thus assigned an address in the internal binary decoder model by its context, the address is unaffected by operations on any other nodes.

5.6.3 Update command semantics

The commands in TeM and BiM are named differently to reflect the fact that the commands operate on different models and have different semantics. The TeM commands have the suffix “node” because the TeM operates (nearly) directly on the current description tree, and thus the removal of a node completely removes it from the tree. The BiM commands have the suffix “content” because the addressing on the current description tree is by indirection, through an internal binary decoder model. Removal of an address, from the point of view of the application, removes the node and its content (sub-elements and attributes) from the current description tree, however the addressed node is still present in the internal decoder model (binary format description tree).

In the TeM, the commands are AddNode, ReplaceNode, and DeleteNode. The AddNode is effectively an “append” command, adding an element among the existing children of the target node. Insertion between two already-received, consecutive children of a node is not possible. One must replace a previously deferred node. By performing a DeleteNode on a node on the current description tree, the addressable indices of its siblings change appropriately.

In the BiM, the commands are AddContent, ReplaceContent and DeleteContent. The AddContent conveys the node data for a node whose path within the description tree is predetermined from the schema evaluation as described in 5.6.2. Hence, internally to the BiM decoder, the paths to (or addresses of) non-empty sibling nodes may be non-contiguous, e.g., the second and fourth occurrence of an element may be present. The "hole" in the numbering is not visible in the current description tree generated by the description composer. Hence, if the third occurrence of said element is added (using AddContent) in a subsequent access unit, it appears to any further processing steps as an "inserted" element in the current description tree, while it simply fills the existing "hole" with respect to the internal numbering of the BiM decoder. Similarly, DeleteContent does delete the node data, but does not change the context path to this node. ReplaceContent replaces node data and does not change the context path to this node either.

For both types of decoders, the "Reset" command reverts the description to the initial description in the `DecoderInit`.

5.6.4 Restrictions on descriptions that may be encoded

The TeM has limited capability to update mixed content models (defined in ISO/IEC 15938-2). Although it allows the replacement of the entire element, or the replacement of child elements, the mixed content itself cannot be addressed or modified.

Wildcards and mixed content models (defined in ISO/IEC 15938-2) are not supported at all by the BiM. Therefore a schema that uses these mechanisms cannot be supported by the binary format.

5.6.5 Navigation

When navigating through a TeM description, at each step the different possible path is given by the element name, an index, and, possibly, a type identifier. The concatenation of that information is expressed (in a reduced form) by XPath.

In BiM, each step down the tree hierarchy is given by a tree branch code (TBC), whose binary coding is derived from the schema. The concatenation of all TBCs constitutes the context path information.

Both mechanisms in TeM and BiM allow for absolute and relative addressing of a node, starting either from the topmost node of the description or a context node known from the previous decoding steps.

5.6.6 Multiple payloads

With the BiM, for compression efficiency, there may be multiple payloads within a single fragment update unit that implicitly operate on subsequent nodes of the same type. This feature does not exist in the TeM.

5.7 Characteristics of the delivery layer

The delivery layer is an abstraction that includes functionalities for the synchronization, framing and multiplexing of description streams with other data streams. Description streams may be delivered independently or together with the described multimedia content. No specific delivery layer is specified or mandated by ISO/IEC 15938.

Provisions for two different modes of delivery are supported by this specification:

— Synchronous delivery – each access unit shall be associated with a unique time that indicates when the description fragment conveyed within this access unit becomes available to the terminal. This point in time is termed "composition time."

— Asynchronous delivery – the point in time when an access unit is conveyed to the terminal is not known to the producer of this description stream nor is it relevant for the usage of the reconstructed description. The composition time is understood to be "best effort," and the order of decoding AUs, if prescribed by the producer of the description, shall be preserved. Note, however, that this in no way precludes time related information ("described time") to be present within the multimedia content description.

A delivery layer (DL) suitable for conveying ISO/IEC 15938 description streams shall have the following properties:

— The DL shall provide a mechanism to communicate a description stream from its producer to the terminal.

— The DL shall provide a mechanism by which at least one entry point to the description stream can be identified. This may correspond to a special case of a random access point, typically at the beginning of the stream.

— For applications requiring random access to description streams, the DL shall provide a suitable random access mechanism.

— The DL shall provide delineation of the access units within the description stream, i.e., AU boundaries shall be preserved end-to-end.

— The DL shall preserve the order of access units on delivery to the terminal, if the producer of the description stream has established such an order.

— The DL shall provide either error-free access units to the terminal or an indication that an error occurred.

— The DL shall provide a means to deliver the `DecoderInit` information (see subclauses 6.2 and 7.2) to the terminal before any access unit decoding occurs and signal the coding format (textual/binary) of said information.

— The DL shall provide signalling of the association of a description stream to one or more media streams.

— In synchronous delivery mode, the DL shall provide time stamping of access units, with the time stamps corresponding to the composition time (see section on synchronous delivery earlier in this subclause) of the respective access unit.

— If an application requires access units to be of equal or restricted lengths, it shall be the responsibility of the DL to provide that functionality transparently to the systems layer.

Companion requirements exist in order to establish the link between the multimedia content description and the described content itself. These requirements, however, may apply to the delivery layer of the description stream or to the delivery layer of the described content streams, depending on the application context:

— The DL for the description stream or the described content shall provide the mapping information between the content references within the description stream and the described streams.

— The DL for the description stream or the described content shall provide the mapping information between the described time and the time of the described content.

6 Textual Format - TeM

6.1 Overview

The following subclauses specify the syntax elements and associated semantics of the textual format for ISO/IEC 15938 descriptions, abbreviated TeM. The textual `DecoderInit` (6.2), the textual `AccessUnit` (6.3), and the textual fragment update unit (6.4), with its constituent parts: the textual fragment update command (6.5), textual fragment update context (6.6) and textual fragment update payload (6.7).

The following schema wrapper shall be applied to the syntax defined in Clause 6.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
       xmlns:mpeg7s="urn:mpeg:mpeg7:systems:2001"
       targetNamespace="urn:mpeg:mpeg7:systems:2001"
       elementFormDefault="qualified"
       attributeFormDefault="unqualified">

       <!-- here clause 6 schema definition -->

</schema>
```

6.2 Textual DecoderInit

6.2.1 Syntax

```
<!-- ################################################ -->
<!-- Definition of DecoderInit        -->
<!-- ################################################ -->

<complexType name="mpeg7s:DecoderInitType">
   <sequence>
      <element name="SchemaReference" type="mpeg7s:SchemaReferenceType"
                maxOccurs="unbounded"/>
      <element name="InitialDescription" type="mpeg7s:AccessUnitType"
                minOccurs=”0”/>
   </sequence>
   <attribute name="systemsProfileLevelIndication" type="decimal"
                use="required"/>
</complexType>

<complexType name="SchemaReferenceType">
   <attribute name="name" type="anyURI" use="required"/>
   <attribute name="locationHint" type="anyURI" use="optional"/>
</complexType>
```

6.2.2 Semantics

The `DecoderInit` is used to initialise configuration parameters required for the decoding of the textual access units.

Name	*Definition*
SchemaReference	A list of references to the schemas used by the description. Each reference consists of a schema URI and a location hint URI. The URI identifies the namespace associated with the schema, as specified in ISO/IEC 15938-2. The

	locationHint is a valid and up-to-date URI that is guaranteed to provide open access to the schema. It shall be present in every case except when it is precisely identical to the corresponding URI. NOTE A terminal need not use the locationHint to locate the schema and may access the schema definition associated with the schema URI using other implementation-defined means. In fact, some terminals may not be capable of retrieving or parsing new schemas.
InitialDescription	This optional element conveys portions of a description using the same syntax and semantics as an access unit (see 6.3). The following restrictions on InitialDescription, compared to a regular fragment update unit apply: — For all fragment update units within the InitialDescription, FUCommand shall take the value "addNode". — The FUContext of the first fragment update unit within the InitialDescription shall have the value of "/" referring to the instance document's root element. The FUPayload(s) shall contain the instance data for the initial description. This instance data provides an initial state of the current description tree. Decoding the InitialDescription plus any subsequent AU or AUs shall lead, after composition, to a schema-valid current description tree that may be passed to the application.
systemsProfileLevelIndication	Used to indicate which Systems profile and level is being used, as defined in Table 1. This table, defined in Clause 7, provides the code associated with each defined Systems profile and level

NOTE That the system layer is not required to output a current description tree after decoding the initial description and, therefore, the decoded instance data need not result in a schema-valid description.

6.3 Textual Access Unit

6.3.1 Syntax

```
<!-- ################################################ -->
<!--  Definition of AccessUnitType              -->
<!-- ################################################ -->

<complexType name="AccessUnitType">
   <sequence>
      <element name="FragmentUpdateUnit" type="mpeg7s:FragmentUpdateUnitType"
               maxOccurs="unbounded"/>
   </sequence>
</complexType>
```

6.3.2 Semantics

An AU is composed of a sequence of fragment update unit(s). Multiple fragment update units in an access unit are ordered and shall be processed by the terminal such that the result of applying the commands is equivalent to having executed them sequentially by the description composer in the order specified within the access unit. The current description tree resulting after composition must be schema valid after all fragment update units have been processed, but intermediate results need not be schema valid.

Name	*Definition*
FragmentUpdateUnit	See 6.4.

6.4 Textual Fragment Update Unit

6.4.1 Syntax

```
<!-- ################################################ -->
<!--  Definition of FragmentUpdateUnitType      -->
<!-- ################################################ -->

<complexType name="FragmentUpdateUnitType">
  <sequence>
    <element name="FUCommand" type="mpeg7s:FragmentUpdateCommandType"/>
    <element name="FUContext" type="mpeg7s:FragmentUpdateContextType"
         minOccurs="0"/>
    <element name="FUPayload" type="mpeg7s:FragmentUpdatePayloadType"
         minOccurs="0" />
  </sequence>
</complexType>
```

6.4.2 Semantics

A fragment update unit is the container for a fragment update and consists of a fragment update command, an optional fragment update context and optional fragment update payload.

Name	*Definition*
FUCommand	Specifies the type of update command to be executed (see 6.5).
FUContext	Establishes the context node for updating the description. The `FUContext` shall not be present when the "reset" command is specified, but shall be present in every other case.
FUPayload	Provides the update value for "addNode" and "replaceNode" commands. The `FUPayload` shall not be present when either a "reset" or "deleteNode" command is used, but shall be present in every other case.

6.5 Textual Fragment Update Command

6.5.1 Syntax

```
<!-- ################################################ -->
<!--  Definition of FragmentUpdateCommandType          -->
<!-- ################################################ -->

<simpleType name="FragmentUpdateCommandType">
   <union>
```

```
        <simpleType>
            <restriction base="string">
                <enumeration value="addNode"/>
                <enumeration value="deleteNode"/>
                <enumeration value="replaceNode"/>
                <enumeration value="reset"/>
            </restriction>
        </simpleType>
        <simpleType>
            <restriction base="string"/>
        </simpleType>
    </union>
</simpleType>
```

6.5.2 Semantics

Name	*Definition*
addNode	Adds the node conveyed within the `FUPayload` to the context node as the last child of the context node.
replaceNode	Replaces the context node by the node(s) conveyed within the `FUPayload`.
deleteNode	Deletes the context node and the nodes that are children of the context node. The parent of the context node becomes the new context node. A fragment update unit with a deleteNode command shall not contain a `FUPayload` element.
reset	Resets the current description tree to the initial description specified in `DecoderInit`. If the initial description is not schema valid on its own, the AU containing the "reset" command shall also contain at least one other fragment update unit such that the description is schema valid when the decoding of the access unit containing the “reset” command is complete. Such an AU (i.e. containing a reset, possibly followed by one or more addNode commands) would normally be marked as a sync point. A fragment update unit with a reset command shall neither contain a `FUContext` nor a `FUPayload` element.

NOTE Deleting a node that has siblings of the same name implicitly causes the position index numbers, as specified by the XPath, of all following sibling nodes of the same name to decrease by one.

6.6 Textual Fragment Update Context

6.6.1 Syntax

```
<!-- ##################################################   -->
<!-- Definition of FragmentUpdateContextType            -->
<!-- ##################################################   -->

<simpleType name="FUContextType">
    <restriction base="string">
        <pattern value =

"/?((\.|(\.\.)|(((\i\c*:)?\i\c*)(\[\d+\])?))(/((\.)|(\.\.)|(((\i\c*:)?\i\c*)(\[\d+\])
?)))*(/@(\i\c*:)?\i\c*)?)|(@(\i\c*:)?\i\c*)"

        />
```

```
   </restriction>
</simpleType>
```

6.6.2 Semantics

The `FUContextType` specifies the navigation path to the context node using the subset of the XPath language. The regular expression specified in the pattern facet specifies this subset. For clarity purpose, this subset is also presented in Annex B. The fragment update context shall be constructed based on the current description tree as composed prior to decoding the current fragment update unit. A relative XPath is interpreted as starting from the 'current context node'. The current context node in TeM is the *parent* node of the context node in the previous `FragmentUpdateUnit`, except in the case of 'AddNode,' where the current context node is the context node from the previous `FragmentUpdateUnit`.

Name	*Definition*
FUContextType	Specifies the navigation path to the context node. The representation is based on the subset of XPath expressions. Inside the XPath expression, qualified elements and attributes shall be used when an element or attribute's name is qualified.

6.6.3 Examples

In the following, examples of the instances of the FUContext datatype are shown:

```
<FUContext>/mpeg7:MPEG7[1]/mpeg7:ContentDescription[1]/mpeg7:AudioVisualContent[1]/mp
eg7:Video[1]</FUContext>

<FUContext>../mpeg7:Video[2]</FUContext>

<FUContext>/mpeg7:MPEG7[1]/@mpeg7:type</FUContext>
```

6.7 Textual Fragment Update Payload

6.7.1 Syntax

```
<!-- ################################################ -->
<!--  Definition of FragmentUpdatePayloadType         -->
<!-- ################################################ -->

<complexType name="FragmentUpdatePayloadType">
     <sequence>
        <any processContents="skip" minOccurs="0"/>
     </sequence>

     <attribute name="hasDeferredNodes" type="boolean"
                 use="required" default="false"/>
     <anyAttribute namespace="##other" processContents="skip" use="optional"/>
</complexType>
```

6.7.2 Semantics

Specifies an update value for a fragment update command.

Name	*Definition*
hasDeferredNodes	if this attribute is true it signals that all elements of the fragment that have empty content and no attributes shall be interpreted as deferred nodes. Such deferred nodes shall be removed from the current description tree before handing it over to the application or before performing schema validation.

NOTE The processContents directive indicates that the fragment payload itself shall not be subject to schema validation when validating individual AUs. This is required since the fragment payload is related to the schema of the transported description rather than the schema for the TeM, identified by urn:mpeg:mpeg7:systems:2001.

7 Binary format - BiM

7.1 Overview

The following subclauses specify the syntax elements and associated semantics of the binary format for ISO/IEC 15938 descriptions, abbreviated BiM. The binary DecoderInit (7.2), the binary access unit (7.3), the binary fragment update unit (7.4) and its constituent parts, the binary fragment update command (7.5) and binary fragment update context (7.6) are covered by Clause 7. The specification of the binary fragment update payload follows in Clause 8.

7.2 Binary DecoderInit

7.2.1 Overview

The binary `DecoderInit` specified in this subclause is used to configure parameters required for the decoding of the binary access units. There is only one `DecoderInit` associated with one description stream.

Main components of the `DecoderInit` are an indication of the profile and level of the associated description stream, a list of schema URIs and optimised type codecs associated to certain data types as well as the initial description.

An optimised type codec specifies an encoding for a data type not using the generic mechanism specified in Clause 8 but optimised with full knowledge of the properties of that data type. Some optimised type codecs are specified in Part 3 of this specification.

7.2.2 Syntax

DecoderInit () {	**Number of bits**	**Mnemonic**
SystemsProfileLevelIndication	8+	vluimsbf8
UnitSizeCode	3	bslbf
ReservedBits	5	
NumberOfSchemas	8+	vluimsbf8
for (k=0; k< NumberOfSchemas; k++) {		
SchemaURI_Length[k]	8+	vluimsbf8
SchemaURI[k]	8* SchemaURI_Length[k]	bslbf
LocationHint_Length[k]	8+	vluimsbf8
LocationHint[k]	8* LocationHint_Length[k]	bslbf
NumberOfTypeCodecs[k]	8+	vluimsbf8
for (i=0; i< NumberOfTypeCodecs[k]; i++) {		
TypeCodecURI_Length[k][i]	8+	vluimsbf8
TypeCodecURI[k][i]	8* TypeCodecURI _Length[k][i]	bslbf
NumberOfTypes[k][i]	8+	vluimsbf8
for (j=0; j< NumberOfTypes[k][i]; j++) {		
TypeIdentificationCode[k][i][j]	8+	vluimsbf8
}		
}		
}		
InitialDescription_Length	8+	vluimsbf8
InitialDescription()		
}		

7.2.3 Semantics

Name	*Definition*
SystemsProfileLevelIndication	Indicates the profile and level as defined in ISO/IEC 15938-1 to which the description stream conforms. Table 1 lists the indices and the corresponding profile and level.
UnitSizeCode	This is a coded representation of UnitSize, as specified in Table 2. UnitSize is used for the decoding of the binary fragment update payload as specified in Clause 8.
NumberOfSchemas	Indicates the number of schemas on which the description stream is based. A zero-value is forbidden.
SchemaURI_Length[k]	Indicates the size in bytes of the `SchemaURI[k]`. A value of zero is forbidden.
SchemaURI[k]	This is the UTF-8 representation of the URI to unambiguously reference one of the schemas that are needed for the decoder to decode the description stream. The `SchemaURI` identifies the schema that declares this `SchemaURI` as being its targetNamespace. The identified schema is the one composed of all schema components defined in its targetNamespace and all schema components imported from other namespaces. To support forward compatibility, multiple `SchemaURI`s are also used to identify imported schemas. Decoders that are aware of any of these schemas will be able to process at least the corresponding parts of the description. The `SchemaID` (see 7.6.3 and 8.4.4) as well as `SchemaIDOfSubstitution` (see 8.4.3) refer to the entries with the corresponding indices in this `SchemaURI` list. The `SchemaURI[0]` shall be assigned to the schema which imports all the namespaces that are identified by a `SchemaURI[k]` with an index k > 1. Thus `SchemaURI[0]` identifies the targetNamespace of the description. NOTE In order to maximize forward compatibility, it is recommended to list the SchemaURI for as many imported namespaces as practical.
LocationHint_Length[k]	Indicates the size in bytes of the `LocationHint[k]` syntax element.
LocationHint[k]	This is the UTF-8 representation of the URI referencing the location of the schema with index k. The `LocationHint[k]` shall be present except if the corresponding `SchemaURI[k]` already provides the location reference. In that case it may be omitted by setting the corresponding `LocationHint_Length[k]` to the value "0".
NumberOfTypeCodecs[k]	Indicates the number of optimised data type codecs that are subsequently associated with data types contained in the schema referred to by the index k.
TypeCodecURI[k][i]	This is the UTF-8 representation of a URI referencing an optimised binary data type codec. This codec shall be used for all data types listed subsequently.
NumberOfTypes[k][i]	Indicates the number of data types which shall be coded with the optimised data type codec referenced by `TypeCodecURI[k][i]`.
TypeIdentificationCode[k][i][j]	Selects one data type from the set of all data types contained in the schema with index k. This data type shall be coded with the optimised data type codec referenced by `TypeCodecURI[k][i]` for all instantiations of this data type in the

	description stream. The syntax and semantics of `TypeIdentificationCode[k][i][j]` is the same as of the type identification code defined in subclause 7.6.5.4 except that here it is represented using vluimsbf8. The `TypeIdentificationCode[k][i][j]` assumes the "anyType" as base type. There shall not be more than one data type codec associated to the same data type. NOTE In order to maximise forward compatibility the value of the index k should refer to the targetNamespace in which the data type is defined.
InitialDescription_Length	Indicates the size in bytes of the `InitialDescription`.
InitialDescription()	This conveys portions of a description using the same syntax and semantics as an access unit (see 7.3). The `InitialDescription` provides an initial state for the binary format description tree (see 7.4) before decoding any subsequent access units. The following restrictions on `InitialDescription`, compared to a regular access unit apply: — For all fragment update units within the `InitialDescription` the fragment update command shall take the value "AddContent". — The first fragment update unit within the initial description shall use an "absolute addressing mode", i.e. `CommandModeCode` equal to '001' or '110'. The `InitialDescription` may be empty, indicated by setting `InitialDescription_Length` to zero. NOTE That the system layer is not required to produce an output after decoding the `InitialDescription` and, therefore, the decoded instance data need not result in a schema-valid description. However, decoding the `InitialDescription` plus any subsequent AU or AUs shall lead, after composition, to a schema-valid current description tree that may be passed to the application.

Table 1 — Index Table for SystemsProfileLevelIndication

Index	*Systems Profile and Level*
0	no profile specified
1 – 127	Reserved for ISO Use

Table 2 — Code Table for UnitSize

Unit Size Code	*Unit Size*
000	default
001	1
010	8
011	16
100	32
101	64
110	128
111	reserved

7.3 Binary Access Unit

7.3.1 Overview

A binary access unit is composed of one or more binary fragment update units that represent one or more description fragments. Therefore, an access unit may convey updates for several distinct parts of a description simultaneously. Syntax and semantics of a fragment update unit are described in subclause 7.4.

7.3.2 Syntax

AccessUnit () {	**Number of bits**	**Mnemonic**
NumberOfFUU	8+	vluimsbf8
for (i=0; i< NumberOfFUU ; i++) {		
FragmentUpdateUnit()		
}		
}		

7.3.3 Semantics

Name	*Definition*
NumberOfFUU	Indicates the number of fragment update units in this access unit.

7.4 Binary Fragment Update Unit

7.4.1 Overview

For the specification of the syntax and semantics of a binary fragment update unit it is recalled that descriptions are hierarchically defined, and therefore, they can be interpreted as a description tree. The elements and attributes in the description tree can generically be also referred to as "nodes".

The topmost node is the node corresponding to the first element in the description. It instantiates one of the global elements declared in the schema. The selector node is defined to be the parent node of the topmost node artificially extending the hierarchy at the top. Figure 5 shows an example description tree.

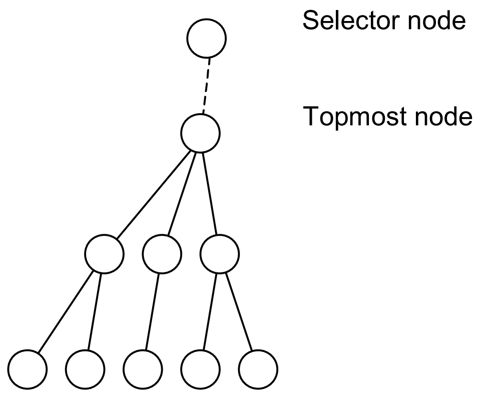


Figure 5 — Example for the tree representation of a description

Two different notions of description trees are used in this Clause: the "current description tree" (see Clause 5) and the "binary format description tree".

The binary format description tree is used for addressing the nodes. The addressing relies upon schema knowledge, i.e. the shared knowledge of encoder and decoder about the existence and position of potential/allowed elements within the schema. The address information specifies a node within the binary format description tree built from all these possible - and not necessarily instantiated - elements as defined in the schema. Moreover, each node has a specific and fixed address which allows an unambiguous identification not depending on the current description as present at the decoder. Note that it is possible to address nodes which do not correspond to an instantiated element. Nodes corresponding to instantiated elements are called "instantiated nodes". Deferred nodes shall be considered as instantiated nodes.

The current description tree is defined, immediately after the decoding of any AU, as the set of instantiated nodes in the binary format description tree.

Each binary fragment update unit consists of 3 sections:

— the fragment update command defining which kind of operation shall be performed on the binary format description tree, i.e. if a description fragment shall be added, replaced or deleted or if the complete binary format description tree shall be reset;

— the fragment update context signals on which node in the binary format description tree the fragment update command shall be executed. Fragment update context is present unless the fragment update command is "reset";

— the fragment update payload containing a description fragment. fragment update payload is present unless the fragment update command is "DeleteContent" or "Reset". A special mode called "MultiplePayloadMode" also allows there to be multiple instances of fragment update payloads of the same type within one fragment update unit.

Additionally, each fragment update unit carries the information about its length in bytes, which allows a decoder to skip. This mechanism may be used in case the decoder does not know the corresponding schema required to decode this fragment update unit.

7.4.2 Syntax

FragmentUpdateUnit () {	**Number of bits**	**Mnemonic**
FUU_Length	**8+**	vluimsbf8
FragmentUpdateCommand	**4**	bslbf

if (FragmentUpdateCommand != '0100') {
        / '0100' corresponds to "Reset" */*		

FragmentUpdateContext()		

if (FragmentUpdateCommand != '0011') {
            / '0011' corresponds to "DeleteContent" */*		

for (i=0;i<NumberOfFragmentPayloads;i++) {		
FragmentUpdatePayload(startType)		
}		
}		
}		
nextByteBoundary()		
}		

7.4.3 Semantics

Name	*Definition*
FUU_Length	Indicates the length in bytes of the remainder of this fragment update unit (excluding the `FUU_Length` syntax element).
FragmentUpdateCommand	Signals the operation that shall be executed on the binary format description tree as specified in 7.5.
FragmentUpdateContext()	See 7.6.
startType	This internal variable indicates the effective data type of the first element that is conveyed in the fragment update payload. The `startType` variable is of type SchemaComponent as specified in 8.3.1. Its value is derived from the Operand_TBC in the `FragmentUpdateContext` as specified in 7.6.
NumberOfFragmentPayloads	The value of this internal variable is derived from `FragmentUpdateContext` as specified in 7.6.
FragmentUpdatePayload()	See 8.3.1.

7.5 Binary Fragment Update Command

The `FragmentUpdateCommand` code word specifies the command that shall be executed on the binary format description tree. Table 3 defines the code words and the semantics of the fragment update command.

Table 3 — Code Table of fragment update commands

Code Word	Command Name	Specification
0000	---	Reserved
0001	AddContent	Add the description fragment contained in the fragment update payload at the node indicated by the operand node (see 7.6). The operand node shall not be an instantiated node but it turns into an instantiated node after processing this fragment update unit. Additionally, all nodes which are part of the context path specified in the fragment update context turn into instantiated nodes after processing this fragment update unit if these had not been instantiated nodes before. NOTE In the current description tree this is equivalent to either appending or inserting the corresponding nodes.
0010	ReplaceContent	Replace the description fragment at the node indicated by the operand node with description fragment contained in the fragment update payload. The operand node shall be an instantiated node.

		This command is equivalent to the command sequence of “DeleteContent” and “AddContent”. NOTE In the current description tree this is equivalent to replacing the corresponding node.
0011	DeleteContent	Delete the description fragment at the node that is indicated by the operand node. The respective node and all its child nodes are reverted to “not instantiated”. NOTE In the current description tree this is equivalent to deleting the corresponding node.
0100	Reset	Reset the complete binary format description tree, i.e. revert all nodes in the binary format description tree to “not instantiated”. and decode the `InitialDescription` conveyed in the `DecoderInit`. NOTE This is equivalent to deleting the complete current description tree.
0101 – 1111	---	Reserved

7.6 Binary Fragment Update Context

7.6.1 Overview

The fragment update context specifies on which node of the binary format description tree the fragment update command shall be executed. This node is called the “operand node”. Additionally, the fragment update context specifies the data type of the node encoded in the subsequent fragment update payload(s).

The operand node is addressed by building a path (called "Context Path") through the binary format description tree. The context path consists of a sequence of local navigation information called Tree Branch Code (TBC). A TBC represents tree branch information from a node to a child node on the path to the operand node (see Figure 6).

A set of TBCs associated to the same complexType is called a TBC table. A TBC table is composed of one TBC for each possible child node of the complexType. Child nodes are defined as the attribute nodes of the complex type as well as, either the contained element nodes or a dedicated node representing a simple content. For the selector node there is a special TBC table containing TBCs corresponding to the global elements defined in the schema. Other TBCs are added to the TBC tables for specific purposes as described below. The algorithm for generating the TBC tables is described in 7.6.5.

A TBC is composed of four parts:

1) a Schema Branch Code (SBC) by which one node among the possible child nodes is selected (see 7.6.5.2),

2) a Substitution Code which is used if the element declaration addressed by the SBC is a reference to an element which is the head of a substitution group (see 7.6.5.3),

3) a Path Type Code which is used if the type of the element identified by the SBC and the Substitution Code is the base type of other named derived types (see 7.6.5.4), and

4) a Position Code which is used if multiple occurrences of the element addressed by the SBC are possible (see 7.6.5.5).

The TBCs for the selector node have no Substitution Code and no Position Code.

NOTE In the syntax definition of the context path this concatenation of TBCs is partly reordered (i.e. the Position Codes from all TBCs are shifted to the end of the context path as described further below).

Two types of Context Path exist. In both cases the Context Path is built from concatenated TBCs and leads to the operand node. In case of an absolute Context Path, the Context Path starts from the selector node. In case of a relative Context Path, the Context Path starts from a "current context node". The current context node in BiM is defined by the previous `FragmentUpdateUnit` as specified in the following paragraphs. Figure 6 shows the principle of absolute and relative addressing.

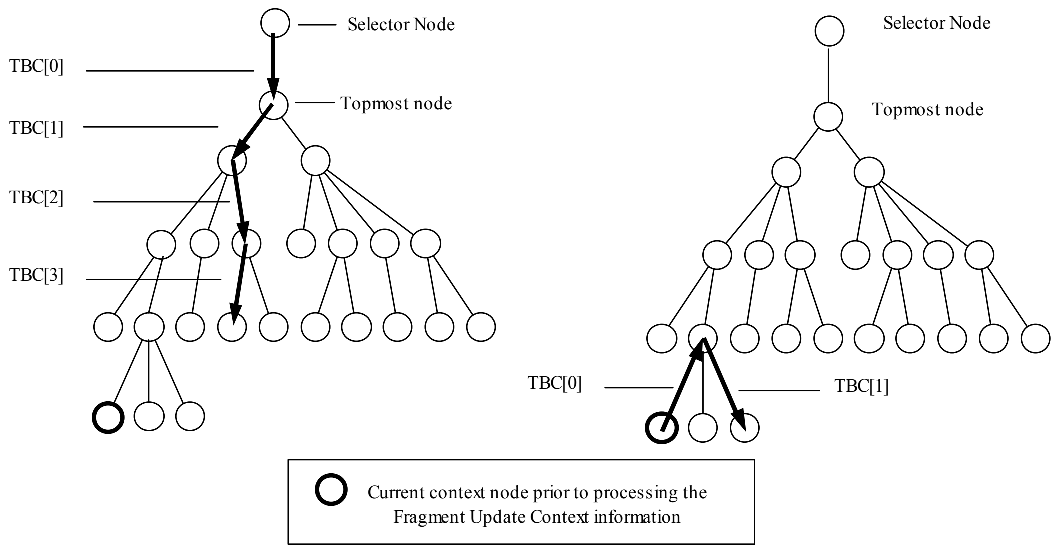


Figure 6 — Absolute (left) and relative context path example

A `ContextModeCode` (7.6.4) allows selecting between absolute and relative addressing modes. Additionally, the `ContextModeCode` may signal the instantiation of multiple fragment update payloads of the same type within a single fragment update unit as specified in 7.6.4 and 7.6.5.6.

There are two different TBC tables associated to each complexType: The ContextTBC table contains only references to the child elements of complexType and additionally one code word to signal the termination of the path (Path Termination Code). The ContextTBC table contains also one TBC to refer to the parent node. It allows relative navigation within the binary format description tree and move upwards to the parent node. The OperandTBC table additionally contains also the references to the attributes and either to the elements of simpleType or to a simple content, but does not contain the Path Termination Code nor the reference to the parent node. Furthermore, in the OperandTBC table one TBC is reserved for User Data Extension as defined in section 7.6.5.2. Example TBC tables are shown in Table 4 and Table 5.

The Context Path coding is done as follows: For all but the last TBC in the Context Path the ContextTBC tables shall be used while, for the last TBC in the Context Path the OperandTBC tables shall be used. The path termination code shall be used to signal that the immediately following TBC is the OperandTBC which is the last TBC in the Context Path. The following definitions apply:

— The "context node" is defined as the node specified by the Context Path except the final TBC and the path termination code. The context node becomes the "current context node" for the Context Path of the subsequent fragment update unit.

— The "operand node" is defined to be the node addressed by the final TBC (from the OperandTBC table). This is the node on which the fragment update command is executed.

This is illustrated in Figure 7.

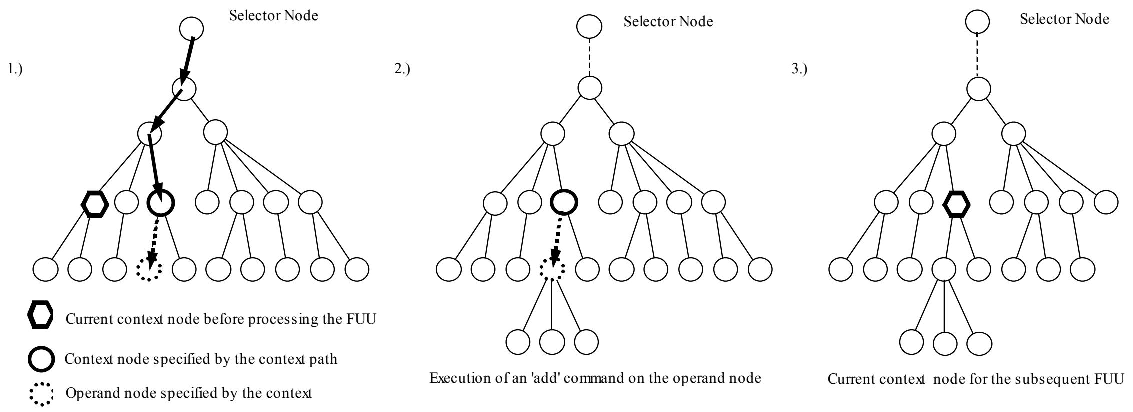

1.)
2.)
3.)
Selector Node
Selector Node
Selector Node
Current context node before processing the FUU
Context node specified by the context path
Operand node specified by the context
Execution of an 'add' command on the operand node
Current context node for the subsequent FUU

Figure 7 — Example of Context node and operand node during the execution of a fragment update unit

The ContextTBC and OperandTBC tables are generated automatically from the schema as specified in 7.6.5 and, hence, do not appear in this specification. Table 4 and Table 5 show examples of a ContextTBC table and of a OperandTBC table for a complexType with 8 children (6 elements and 2 attributes) where 4 elements are of complexType.

Table 4 — Example of a Context TBC Table

Tree Branch Code				Tree Branch
SBC_ Context	*Substitution Code*	*Type Code*	*Position Code*	
000	--	--	--	Reference to parent
001	[Subst. Code]	[Type Code]	[Pos.Code]	Reference to first child of complexType
010	[Subst. Code]	[Type Code]	[Pos.Code]	Reference to second child of complexType
011	[Subst. Code]	[Type Code]	[Pos.Code]	Reference to third child of complexType
100	[Subst. Code]	[Type Code]	[Pos.Code]	Reference to fourth child of complexType
101 - 110	--	--	--	Forbidden
111	--	--	--	Path Termination Code

Table 5 — Example of a Operand TBC Table

Tree Branch Code				Tree Branch
SBC_ Operand	*Substitution Code*	*Type Code*	*Position Code*	
0000	--	--	[Pos.Code]	User Data Extension Code
0001	[Subst. Code]	[Type Code]	[Pos.Code]	Reference to first child

0010	[Subst. Code]	[Type Code]	[Pos.Code]	Reference to second child (element)
0011	[Subst. Code]	[Type Code]	[Pos.Code]	Reference to third child (element)
0100	[Subst. Code]	[Type Code]	[Pos.Code]	Reference to fourth child (element)
0101	[Subst. Code]	[Type Code]	[Pos.Code]	Reference to fifth child (element)
0110	[Subst. Code]	[Type Code]	[Pos.Code]	Reference to sixth child (element)
0111	--	--	--	Reference to seventh child (attribute)
1000	--	--	--	Reference to eighth child (attribute)
1001 -1111	--	--	--	Forbidden

As every ContextTBC table contains a code word for the reference to the parent node, it is also possible to move upwards in the binary format description tree when using a relative addressing mode.

In order to support efficient searching and filtering the description stream is ordered in a way that first all instances of Schema Branch Codes including their corresponding Substitution Code and Type Code are present and only then all Position Codes of the context path follow as shown in Figure 8.

SBCs, Substitution Codes & Type Codes	**Position Codes**

SBC_Context 1 SubstitutionCode 1 Type Code 1	SBC_Context 2 SubstitutionCode 2 Type Code 2	[...]	SBC_Context n-1 SubstitutionCode n-1 Type Code n	SBC_Context - (Path Termination Code)	SBC_Operand n SubstitutionCode n Type Code n	Pos Code 1	Pos Code 2	[...]	Pos Code n-1	Pos Code n

Figure 8 — Example of the structure of a Context Path

7.6.2 Syntax

FragmentUpdateContext () {	**Number of bits**	**Mnemonic**
SchemaID	ceil(log2(`NumberOfSchemas`))	uimsbf
ContextModeCode	3	bslbf
ContextPath()		
}		

ContextPath () {	**Number of bits**	**Mnemonic**
TBC_Counter = 0		
NumberOfFragmentPayloads = 1		
do {		
if ((ContextModeCode == '001' || ContextModeCode == '011') && TBC_Counter ==0) { */* absolute addressing mode and first TBC of the context path */*		
SBC_Context_Selector	ceil(log2(number of global elements +1))	bslbf
PathTypeCode()		
}		
else {		
SBC_Context	ceil(log2(number of child elements of complexType + 2))	bslbf
SubstitutionCode()		
PathTypeCode()		
}		
TBC_Counter ++		
} while ((SBC_Context_Selector != "Path Termination Code") && (SBC_Context != "Path Termination Code"))		
if (SBC_Context_Selector == "Path Termination Code")){		
SBC_Operand_Selector	ceil(log2(number of global elements))	bslbf
PathTypeCode()		
}		
else {		
SBC_Operand	ceil(log2(number of child elements + number of attributes + has_simpleContent + 1))	bslbf
SubstitutionCode()		
PathTypeCode()		
}		
TBC_Counter ++		
for (i=0; i < TBC_Counter; i++) {		
PositionCode()		
}		
if ((ContextModeCode == '011') || (ContextModeCode == '100')) { */* multiple fragment update payload mode*/*		
do {		
IncrementalPositionCode	ceil(log2(NumberOfMultiOccurren	bslbf

	ceLayer+2))	
if (IncrementalPositionCode != “Skip_Indication“) {		
NumberOfFragmentPayloads++		
}		
else {		
IncrementalPositionCode * /* indicating the skipped position */*	ceil (log2(`NumberOfMultiOccurrenceLayer`+2))	bslbf
}		
} while (IncrementalPositionCode != “IncrementalPositionCodeTermination”)		
NumberOfFragmentPayloads-- */* there is no fragment update payload corresponding to the IncrementalPositionCodeTermination */*		
}		
}		

7.6.3 Semantics

Name	*Definition*
SchemaID	Identifies the schema (from the list of `schemaURI`s transmitted in the `DecoderInit`) which is used as basis for the fragment update context coding. The `SchemaID` code word is built by sequentially addressing the list of `SchemaURI` contained in the `DecoderInit`. The length of this field is determined by: “ceil(log2(`NumberOfSchemas`))”. The value of this code word is the same as the variable "k" in the definition of the `SchemaURI[k]` syntax element as specified in 7.2.3. The `SchemaID` syntax element is also used for the decoding of the fragment update payload as described in subclause 8.4.4. If the `ContextModeCode` selects a relative addressing mode then the `SchemaID` shall have the same value as in the previous fragment update unit.
ContextModeCode	Signals the addressing mode for the Context Path as specified in 7.6.4.
ContextPath()	See 7.6.5.

Name	*Definition*
TBC_Counter	This internal variable represents the number of TBCs in the context path.
SBC_Context_Selector	Selects one global element of the schema referenced by `SchemaID` using the ContextTBC table as specified in 7.6.5.2.3.
PathTypeCode()	See 7.6.5.4.

SBC_Context	Selects one child node using the ContextTBC table as specified in 7.6.5.2.2.
SubstitutionCode()	See 7.6.5.3.
SBC_Operand_Selector	Selects one global element of the schema referenced by `SchemaID` using the table OperandTBC table as specified in 7.6.5.2.3.
SBC_Operand	Selects one child node using the OperandTBC table as specified in 7.6.5.2.2.
PositionCode()	See 7.6.5.5.
NumberOfFragmentPayloads	This internal variable represents the number of fragment update payload syntax elements present in this fragment update unit as specified in 7.6.5.6.
NumberOfMultiOccurrenceLayer	This internal variable represents the number of TBCs in the context path for which a Position Code is present. Its use is specified in 7.6.5.6.
IncrementalPositionCode	See 7.6.5.6.

7.6.4 Context Mode

The context mode specifies the addressing mode for the context path. The code word for the context mode selection has a fixed bit length of 3 bits and its semantics are specified in Table 6.

Table 6 — Code Table of Context Mode

Code	Context Mode
000	Reserved
001	Navigate in “Absolute addressing mode” from the selector node to the node specified by the Context Path.
010	Navigate in “Relative addressing mode” from the context node set by the previous fragment update unit to the node specified by the Context Path in
011	Navigate in “Absolute addressing mode” from the selector node to the nodes specified by the Context Path and use the mechanism for multiple payload as specified in 7.6.5.6.
100	Navigate in “Relative addressing mode” from the context node set by the previous fragment update unit to the nodes specified by the Context Path and use the mechanism for multiple payload as specified in 7.6.5.6.
101-111	Reserved

The following restriction applies on the usage of these Context Modes:

— The first fragment update unit of the first access unit of a description stream shall use an absolute addressing mode (i.e. code ‘001’ or ‘011’). In addition, the first fragment update unit of the initial description shall use an absolute addressing mode.

7.6.5 Context Path

7.6.5.1 Overview

The Context Path specifies on which node in the binary fragment description tree the fragment update command shall be executed and specifies the data type of the operand node. This data type of the operand node is required for the decoding of the fragment update payload and is internally conveyed in the variable `startType`. A Context Path is composed of a sequence of Tree Branch Codes (TBC). Each TBC is composed of a Schema Branch Code, a Substitution Code, a Path Type Code and a Position Code. The following subclauses describe the syntax elements that are used to build the TBCs.

7.6.5.2 Schema Branch Codes

7.6.5.2.1 Overview

A Schema Branch Code (SBC) is used to select a node as branch for the navigation in the binary format description tree. The SBCs in the ContextTBC table and in the OperandTBC table differ (as described in subclause 7.6.1 and shown in Table 5). The SBCs are assigned as specified in 7.6.5.2.2. For the special case of the selector node the SBCs are assigned as specified in 7.6.5.2.3.

7.6.5.2.2 SBC_Context and SBC_Operand

— The length of the Schema Branch Code words is derived from the schema and it is determined by the number of different child nodes of the complexType as follows:

  — For the table for ContextTBCs: ceil(log2(number of child elements of complexType + 2)).

  — For the table for OperandTBCs: ceil(log2(number of child elements + number of attributes + has_simpleContent + 1)), where the variable “has_simpleContent” takes the value 1 if the complexType has simple content and the value 0 otherwise.

— In the table for ContextTBCs the all-zero Schema Branch code is always assigned to the reference to the parent node. This SBC shall only be used if the Context Mode Code selects a relative addressing mode.

— In the table for ContextTBCs the all-one Schema Branch Code is always used for the Path Terminating Code

— In the table for OperandTBCs the all-zero SBC is always assigned to the User Data Extension Code. This can be used to insert any user data. A decoder not capable to decode the user data shall skip the user data and continue decoding from the subsequent fragment update unit.

NOTE User data is defined by users for their specific applications. It may in principle be used for extensions of schemas. However, it is recommended to use the mechanisms provided by ISO/IEC 15938-2 for such extensions.

— All other Schema Branch Codes are assigned to the children nodes of the complexType. The children are defined as the attributes of the complex type as well as, either the contained elements or a dedicated child representing a simple content. If there are two or more element declarations with the same name in the complexType definition then each shall be assigned a different SBC.

— A referenced attribute or referenced model group is not considered as a child. Instead the attributes of the referenced attribute group and the content of the referenced group are considered as children.

- If data types are derived then the SBCs for all children of the base data type are assigned first. In the case of derivation by restriction the SBCs of the base data type are kept. Following this rule the children of the base data type have the same SBCs also in the derived data type.

- In the table for ContextTBCs the SBCs are assigned only to child elements that are of complexType while in the table for OperandTBCs the SBCs are assigned to all child elements and attributes and to the simple content.

- The SBCs for child elements and simple content are assigned first, the SBCs for attributes are assigned last. The attributes are ordered lexicographically for the assignment of the Schema Branch Codes.

- In order to unambiguously assign SBCs to the child elements, the element declarations are ordered by the following rules applied in the following order:

  - if a “choice” group is declared within a “choice” group then the inner “choice” group is deleted and its content is added to the content of the outer “choice” group. This rule is applied until there are no more choice groups contained in other choice group.

  - element declarations and “sequence” group declarations declared within a “choice” or an “all” group are ordered lexicographically with respect to their signature as defined in 8.5.2.2.4.

  - element declarations and model group declarations in “sequence” groups are not reordered.

  - if a group is declared within another group then the inner group is replaced at the respective position in the outer group by its content. This rule is applied until there are no more groups contained in other groups.

  After this ordering the SBCs are assigned sequentially to the elements order in the remaining group.

7.6.5.2.3 SBC_Context_Selector and SBC_Operand_Selector

For the special case of the selector node the following rules apply:

- The length in bits of these SBCs is determined by the number of global elements declared in the schema referred by the `SchemaID` as follows:

  - SBC_Context_Selector: ceil(log2(number of global elements + 1)).

  - SBC_Operand_Selector: ceil(log2(number of global elements)).

- The SBCs are assigned sequentially to the global elements defined in the schema referred by the `SchemaID`. Lexicographical ordering is performed before the assignment.

- No codes are assigned for a reference to the parent node nor for the User Data Extension Code. The Path Termination Code, however, is present in the ContextTBC table.

7.6.5.3 Substitution Codes

7.6.5.3.1 Overview

In case a TBC represents a reference to an elements that is the head of a substitution group there is an additional code for addressing that substitution group. This code is called `SubstitutionSelect`. It identifies the selected element in the set of all elements members of this substitution groups. The presence of a substitution and consequently the presence of the `SubstitutionSelect` code word is signalled by the `SubstitutionFlag`.

7.6.5.3.2 Syntax

SubstitutionCode () {	**Number of bits**	**Mnemonic**
if (substitution_possible == 1) {		
SubstitutionFlag	1	bslbf
if (SubstitutionFlag == 1) {		
SubstitutionSelect	ceil(log2(number_of_possible_substitutes))	bslbf
}		
}		
}		

7.6.5.3.3 Semantics

Name	*Definition*
substitution_possible	This is in internal flag which is derived from schema evaluation as specified in 7.6.5.3.1 indicating whether an element is a head element of a substitution group. `substitution_possible` is always false for the following TBCs: “Path Termination Code”, “User Data Extension Code”, “Reference to Parent”.
SubstitutionFlag	Signals whether a substitution is present for the element (`SubstitutionFlag`=1).
SubstitutionSelect	This code is used as address within a substitution group where each element is assigned a `SubstitutionSelect` code. The `SubstitutionSelect` codes referring to the elements are assigned sequentially starting from zero after lexicographical ordering of the element using their expanded names as defined in subclause 8.2. The length of this field is determined by “ceil(log2(number_of_possible_substitutes))”.

7.6.5.4 Type Code in the Context Path (Path_Type_Code)

7.6.5.4.1 Overview

The `PathTypeCode` is used within the Context Path to indicate the element type in case of a type cast using the xsi:type attribute. This type is called the effective type.

The `PathTypeCode` is only present if a type cast can occur for the element, i.e. if in the schema referenced by the `SchemaID` there is at least one named type derived from the respective element type. The presence of a type cast (i.e. the presence of the xsi:type attribute in the description) is signalled by the `TypeCodeFlag`. This flag is also present in the case of an abstract type definition. If a type cast is signalled then a `TypeIdentificationCode` is present which selects the effective type from the set of possible types.

7.6.5.4.2 Syntax

PathTypeCode () {	**Number of bits**	**Mnemonic**
if (type_cast_possible == 1) {		
TypeCodeFlag	1	bslbf
if ((TypeCodeFlag == 1) {		
TypeIdentificationCode	ceil(log2(number of derived types))	bslbf
}		
}		
}		

7.6.5.4.3 Semantics

Name	*Definition*
type_cast_possible	This internal flag which is derived from schema evaluation as specified in 7.6.5.4.1 indicates whether a type can be subject to type casting. `type_cast_possible` is always false for the following TBCs: “Path Termination Code”, “User Data Extension Code”, “Reference to Parent”.
TypeCodeFlag	This flag indicates whether a type cast is present or not.
TypeIdentificationCode	Identifies a type by a code word. The Type Identification Code is generated for a given type (simpleType or complexType) from the set of all derived types (itself being not included) including abstract types defined in the schema referenced by SchemaID. The Type Identification Codes are assigned in a depth-first manner with respect to the hierarchy of types which forms a tree as shown in an example in Figure 9. For types which are siblings within the type hierarchy the code words are assigned in a lexicographical order based on their expanded names. The Type Identification Code identifies the derived type which is used for the type cast. The length of the code word for the Type Identification Code is equal to “ceil(log2(number of derived types in the schema))”.

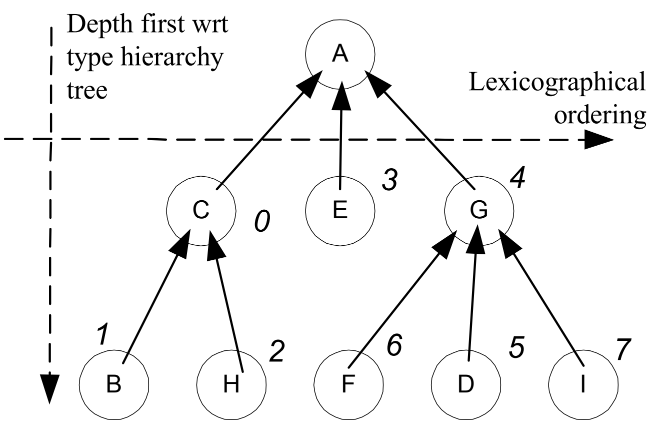


Figure 9 — Example for the Type Identification Code assignment for the types derived from A

7.6.5.5 Position Codes

7.6.5.5.1 Overview

Within a TBC a Position Code shall uniquely identify the position of a node among its sibling nodes in the binary description tree. It is present only if multiple occurrences are possible for the element referenced by the SBC or for any model group declared in the corresponding complexType definition. Position Codes are distinguished in Multiple element Position Codes (MPC) and Single element Position Codes (SPC) for efficiency reasons. The presence of the Position Code and the decision whether SPC or MPC are used is determined by the complexType definition.

There is no Position Code present in the TBC if the SBC is equal to the “Path Termination Code” or to the “Reference to Parent”. Additionally, in the TBCs for the selector node there is no Position Code.

NOTE Since the Context Path consists of several TBCs (each of which has either no Position Code, a SPC or a MPC) it is possible to have SPCs and MPCs within the same Context Path.

7.6.5.5.2 Single Element Position Codes

A SPC is used, if a Position Code is present according to 7.6.5.5.1 and if the corresponding complexType does not contain model groups with maxOccurs > 1. The SPC is only present if the SBC addresses an element with maxOccurs > 1. The SPC indicates the position of the node among the nodes addressed by the same SBC.

The position is represented as integer value. The length in bits of the SPC is the equal to “ceil(log2(maxOccurs of the element addressed by the SBC))”. If this length exceeds 4 bits then vluimsbf5 is used for coding the SPC.

7.6.5.5.3 Multiple Element Position Codes

In case of complexTypes with complex content which contain model groups with maxOccurs > 1, the positions of all nodes representing child elements declared in this complexType are encoded using the MPC. The position of an element relative to its sibling nodes is defined by its index among all children nodes that represent elements. Positions are the same for ContextTBCs and OperandTBCs, i.e. elements of simpleType are also counted in the MPC for a ContextTBC.

The length in bits of the MPC is determined by the following method which uses the ‘max occurs’ property of the effective content particles of the type definition.

The maximum number of elements that a particle can instantiate is called MPA. It is computed according to the following rules:

— **For a sequence particle**

if an index ‘i' exists such that MPA_i = 'unbounded' or $m_{sequence}$ = 'unbounded'

$MPA_{sequence}$ = 'unbounded'

else

$$MPA_{sequence} = m_{sequence} * \sum_{1}^{nb\_of\_children} MPA_i$$

where

“MPA_i” is equal to the maximum number of elements that the ith children particle of the sequence can instantiate

“$m_{sequence}$” is equal to the 'max occurs' property of the sequence particle

— **For a choice particle**

```
        if an index ‘i' exists such that $MPA_i$ = 'unbounded' or $m_{choice}$ = 'unbounded'
            $MPA_{choice}$ = 'unbounded'
        else
            $MPA_{choice} = m_{choice} * \max(MPA_i)$
        where
            “$MPA_i$” is equal to the maximum number of elements that the ith children particle of the choice can instantiate
            “$m_{choice}$” is equal to the 'max occurs' attribute of the choice particle
```

— **For an all particle**

```
    if $m_{all}$= 'unbounded'
        $MPA_{all}$ = 'unbounded'
    else
        $MPA_{all} = m_{all} * \max(MPA_i)$
        where
            “$MPA_i$” is equal to the maximum number of elements that the ith children particle of the all can instantiate
            “$m_{all}$” is equal to the 'max occurs' property of the all particle
```

— **For an element declaration particle**

```
    $MPA_{element} = m_{element}$
    where
        “$m_{element}$” is equal to the 'max occurs' property of the element declaration particle
```

Combining these rules, the maximum number of elements that can be present in an instance of the complexType is equal to the MPA of its effective content particle. The MPC is decoded according to the following rules:

- if (MPA <= 65535)
  - the MPC is coded as a uimsbf field of “ceil(log2(MPA))” bits
- if (M > 65535) or (M = 'unbounded')
  - the MPC is coded as a vluimsbf5.

7.6.5.5.4 Implicit Assignment of Position

If an instantiated element was conveyed as part of a fragment update payload then the corresponding node has not been explicitly assigned a position in the binary format description tree. In this case, the following implicit positions are assigned to each added node for which a position code is expected in the TBC addressing this node:

— in the case a MPC is expected: a position is assigned incrementally (starting from zero) to the added elements.

— in the case a SPC is expected: a position is assigned incrementally (starting from zero) to the added elements corresponding to the same SBC.

7.6.5.6 Multiple Payload Mode

A fragment update unit can contain multiple fragment update payloads of the same type if the context paths of those fragment update payloads are identical except for their position codes. The position codes for the first fragment update payload are coded in the same way as in the case of a single payload, while the position codes for the other fragment update payloads within this fragment update unit are indicated in the context path by "Incremental Position Codes" as shown in Figure 10.

SBCs, Substitution Codes & Type Codes										Position Codes									
															Incremental				
SBC_Context 1	Substitution Code 1	SBC_COntext 2	Substitution Code 2	..	SBC_Context n-1	Substitution Code n-1	SBC_Context (Path Termination Code)	SBC_Operand n	Substitution Code n	Pos Code 1	Pos Code 2	..	Pos Code n-1	Pos Code n	Increment 1	Increment 2	..	Increment m	Termination

Figure 10 — Example of the structure of a Context Path (multiple fragment update payloads)

The length in bits of each Incremental Position Code is equal to "ceil(log2(NumberOfMultiOccurrenceLayer+2))", where `NumberOfMultiOccurrenceLayer` denotes the number of TBCs in the Context Path for which a Position Code is present. A "multiple-occurrence node" is defined as a node which is addressed in the context path by a TBC that has a position code. An example is shown in Figure 11.

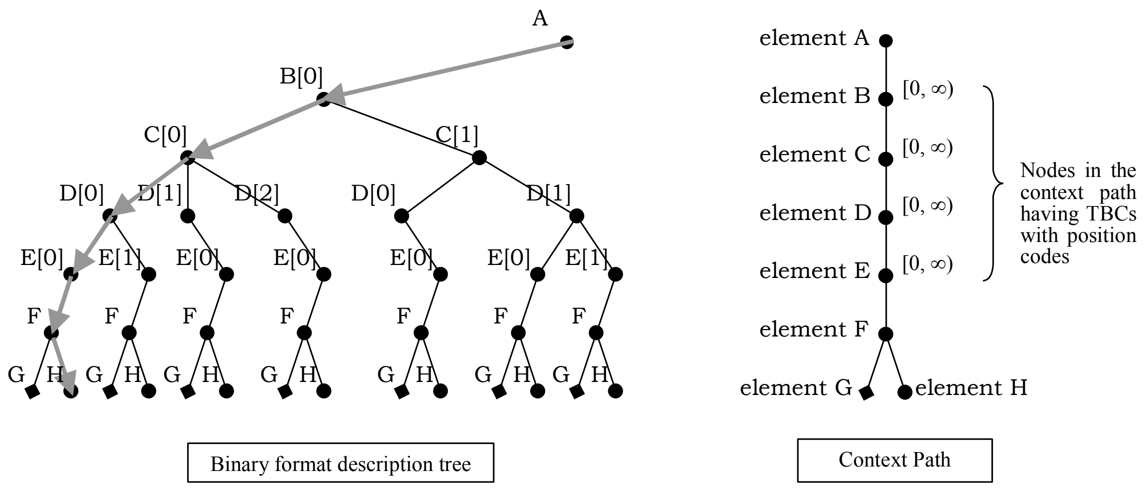


Figure 11 — Example for multiple-occurrence nodes in a context path

The set of multiple-occurrence nodes is indexed beginning from the starting node of the context path. An Incremental Position Code indicates the index of the multiple-occurrence node in the context path for which the position code shall be incremented by 1. The position code for all multiple-occurrence nodes with a higher index is set to "0".

In order to skip positions for which no fragment update payload is present in this fragment update unit a specific incremental position code called "Skip_Indication" is used. This signals that the position specified by the subsequent incremental position code has no payload. In order to indicate that no further incremental position code is present, a specific incremental position code called "IncrementalPositionCodeTermination" is used.

The codes for the indices of the multiple-occurrence nodes are assigned as follows:

— the "all zeros" code word is reserved for "Skip_Indication"

— the code words are then assigned sequentially to the indices of the multiple-occurrence nodes

— the "all ones" code word is reserved for "IncrementalPositionCodeTermination"

Example

An example for the multiple fragment update payload mode is given below:

Table 7 shows the case that the `NumberOfMultiOccurrenceLayer` is equal to 4 (i.e. shown by 3 bits). The multiple-occurrence nodes are indexed by 0 to 3.

Table 7 — Example for the assignment of incremental position codes

Code	Position Codes
000	"Skip_Indication" Indicates that the next position is skipped, i.e. there is no payload corresponding to the position indicated by the subsequent Incremental Position Code.
001	Increment the Position Code of the multiple-occurrence node with index 0. Set the Position Code of the multiple-occurrence nodes with indices > 0 to "0".
010	Increment the Position Code of the multiple-occurrence node with index 1. Set the Position Code of the multiple-occurrence nodes with indices > 1 to "0".
011	Increment the Position Code of the multiple-occurrence node with index 2. Set the Position Code of the multiple-occurrence nodes with indices > 2 to "0".
100	Increment the Position Code of the multiple-occurrence node with index 3. Set the Position Code of the multiple-occurrence nodes with indices > 3 to "0".
101-110	Forbidden.

111	"IncrementalPositionCodeTermination" Indicates to terminate the increments of Position Codes, i.e. the preceding Incremental Position Code indicates the last position for which a fragment update payload is present in this fragment update unit.

Examples of updating the position codes by incremental position codes are shown in Figure 12, in which "Incr Pos Code" denotes the incremental position code and "Pos Codes" denote the position codes before/after the updating; The left side of an arrow is before updating and the right side is after updating.

The code '100' denotes that the multiple-occurrence node with index 3 is updated as shown in Figure 12 (a). The code '011' denotes that the multiple-occurrence node with index 2 is updated as shown in Figure 12 (b), in which the position code of the multiple-occurrence node with index 3 is set to "0". The code '010' denotes that the multiple-occurrence node with index 1 is updated shown as Figure 12 (c), in which multiple-occurrence node with indices 2 and 3 are set to "0". The code '111' indicates the termination of the incremental position codes. When the code '000' is received, the position obtained by the next incremental position code is indicated as skipped meaning that there is no payload corresponding that position.

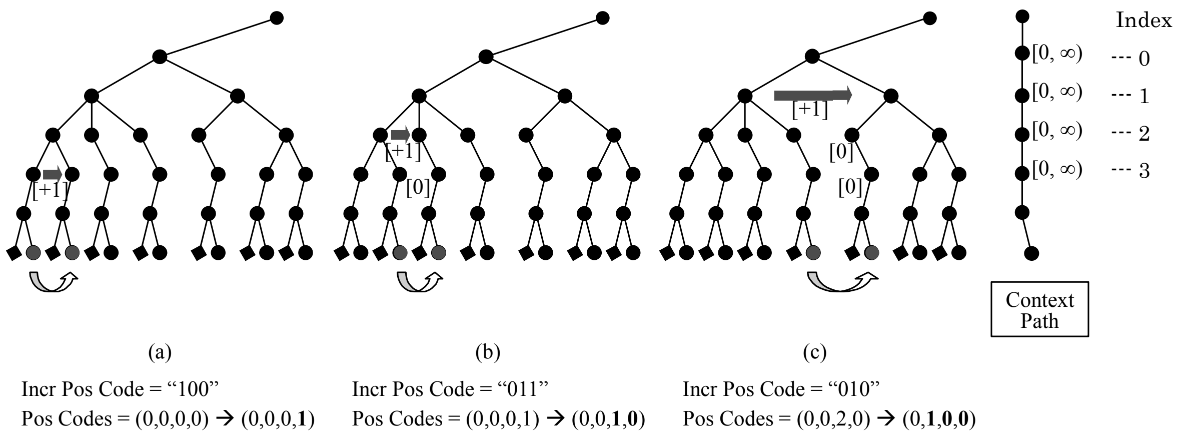


Figure 12 — Indicated Positions using Incremental Position Codes

8 Binary Fragment Update Payload

8.1 Overview

The binary fragment update payload syntax (`FUPayload`) is specified in subclause 8.3. It is composed of flags which define some decoding modes and a payload content which is either an `element` or a simple value (`simpleType`). The syntax of a binary `element` is specified in subclause 8.4. The element content (attributes, complex content or simple content) is decoded by the decoding processes specified in subclause 8.5. In particular, a complexType with complex content is decoded by a Finite State Automaton Decoder (short FSAD). FSADs are generated from the complex types definitions in the schema. Their main objective is to model a decoding process which uses the schema knowledge to efficiently compress structural information (element nesting, element and attribute names). They trigger the decoding of their children elements which in return can use FSADs to decode their content. As a consequence, the payload decoder manages a stack of FSADs each one modeling the decoding of an element with complex content. The leaves of the binary format description tree are decoded by dedicated decoders associated to simple types.

8.2 Definitions

The syntax and semantics of some decoding steps rely on "SchemaComponent" variables. They represent a schema component as defined in XMLSchema – Part 1, Chapter 3.15.2.

The following methods accept "SchemaComponent" parameters.

Name	*Definition*
boolean isSimpleType(SchemaComponent theType)	Returns "true" if the SchemaComponent object "theType" is a simpleType (XMLSchema – Part 2, Chapter 4.1.1).
boolean restrictedType(SchemaComponent baseType, SchemaComponent extendedType)	Returns "true" if the type “extendedType” is a restriction of the type “baseType” i.e. if the two types are separated in the type hierarchy only by derivations by restriction (see XMLSchema – Part 1, Chapter 2.2.1.1). In other cases, it returns "false".
boolean hasSimpleContent(SchemaComponent theType)	Returns “true” if “theType” is a complex type and has SimpleContent.
SchemaComponent getSimpleContentType(SchemaComponent theType)	Returns the simple type associated to the simple content of the type “theType” i.e. the simple type corresponding to the ‘content type’ property of the type "theType" (see XMLSchema – Part 1, Chapter 3.4.1).
boolean hasNamedSubtypes(SchemaComponent theType)	Returns true if the type "theType" has named derived types, i.e. anonymous derived types are not considered.
SchemaComponent getDerivedType(SchemaComponent theType, integer derivedTypeCode)	Returns the SchemaComponent associated to the derived type of the type "theType" whose type code is "derivedTypeCode" as specified in subclause 7.6.5.4.

SchemaComponent expanded name

In order to unambiguously identifies a named schema component we define its “expanded name”:

An schema component expanded name is a character string composed of the namespace URI of the component, followed by ':', followed by the name of the component.

8.3 Fragment Update Payload syntax and semantics

8.3.1 FragmentUpdatePayload

8.3.1.1 Syntax

FragmentUpdatePayload (SchemaComponent startType) {	**Number of bits**	**Mnemonic**
if (isSimpleType(startType)) {		
SimpleType(startType)		
} else {		
DecodingModes()		
Element(“hot“, startType)		
}		
}		

8.3.1.2 Semantics

The `FragmentUpdatePayload` syntax element is the main wrapper of the binary fragment update payload. It is composed of either a `SimpleType` or a `DecodingMode` and an `Element`.

Name	*Definition*
startType	The type of the element to decode. This type is transmitted to the `FragmentUpdatePayload` by the `FragmentUpdateContext` (See 7.6).
SimpleType()	See 8.4.7.
DecodingModes()	See 8.3.2.
Element()	See 8.4.1. The “hot” value and its semantics are defined in 8.4.1.

8.3.2 Decoding Modes

8.3.2.1 Syntax

DecodingModes () {	**Number of bits**	**Mnemonic**
lengthCodingMode	2	bslbf
hasDeferredNodes	1	bslbf
hasTypeCasting	1	bslbf
ReservedBits	4	bslbf
}		

8.3.2.2 Semantics

A BiM fragment payload starts with a 8-bit header which initialises some decoder modes.

Name	*Definition*
lengthCodingMode	A code which specifies if elements length coding is present in a mandatory or optional mode or if it is not present at all according to Table 8.
hasDeferredNodes	A flag which specifies if the `FragmentUpdatePayload` contains deferred nodes. This 1-bit flag can have the following values: — 0 : `hasDeferredNodes` is equal to false, — 1 : `hasDeferredNodes` is equal to true.
hasTypeCasting	A flag which specifies if in this fragment update payload one or more elements explicitly assert their type using the attribute `xsi:type`. This 1-bit flag can have the following values: — 0 – `hasTypeCasting` is equal to false, — 1 - `hasTypeCasting` is equal to true.
ReservedBits	Reserved for future extensions.

Table 8 — lengthCodingMode definition

Code Word	Skipping mode
00	Length not coded
01	Length optionally coded
10	Length always coded
11	reserved

8.4 Element syntax and semantics

8.4.1 Element

8.4.1.1 Syntax

Element (Enumeration SchemaModeStatus, SchemaComponent theType) {	**Number of bits**	**Mnemonic**
if (NumberOfSchemas >1) {		
if (SchemaModeStatus == "hot") {		
SchemaModeUpdate	1-3	vlclbf
}		
if (ElementContentDecodingMode == "mono"){		
Mono-VersionElementContent(ChildrenSchemaMode, theType)		
} else {		
Multiple-VersionElementContent(ChildrenSchemaMode, theType)		
}		
} else {		

Mono-VersionElementContent(“mono”, theType)		
}		
}		

8.4.1.2 Semantics

Name	*Definition*
NumberOfSchemas	The number of schema conveyed in the `DecoderInit` as specified in 7.2.
SchemaModeStatus	The status of the schema mode value associated to the currently decoded element. This enumerated variable can have the following values: — “*hot*” - the schema used for the decoding of the element content might change (see 8.4.3) — “*frozen*” - the schema used for the decoding of the element content shall remain the same as the schema used for the element itself.
SchemaModeUpdate	A variable which determines if the decoding of the element content is done with the same schema as the element itself. The content of the element is coded in "Mono-Version mode" or in "Multiple-Version mode". The `SchemaModeUpdate` code word indicates the following, according to Table 9: — "*mono_not_frozen*" - The decoding of the element is performed using the mono-version decoding mode as specified in subclause 8.4.2. The schema used for the decoding process of the element content is the same as the one used for the element itself. — "*multi_not_frozen*" - The decoding of the element is performed using multiple-version decoding mode as specified in subclause 8.4.3. The schemas used for the decoding of the element content are specified by the `SchemaID` field of each `ElementContentChunk` as specified in 8.4.4. — "*mono_frozen*" - The decoding of the element is performed using the mono-version decoding mode as specified in subclause 8.4.2. The schema used for the decoding process of the element content is the same as the one used for the element itself. The SchemaModeStatus of its children element is set to “frozen”. — "*multi_children_frozen*" - The decoding of the element is performed using the multiple-version decoding mode as specified in subclause 8.4.3. Each one of its children elements is decoded using the mono-version decoding mode as specified in subclause 8.4.2. The schemas used for the decoding process are specified by the `SchemaID` of each `ElementContentChunk` as specified in 8.4.4.
ElementContentDecodingMode	An enumerated variable which determines in which mode is the element decoding performed. It can have the following values: — *“mono”* - The decoding of the element is performed using the mono-version decoding mode as specified in subclause 8.4.2. — *“multi”* - The decoding of the element is performed using the multiple-version decoding mode as specified in subclause 8.4.3.

	Its value is deduced from the `SchemaModeStatus` and the `SchemaModeUpdate` according to rules defined in Table 10.
ChildrenSchemaMode	The schema mode associated to the element content. Its value is deduced from the `SchemaModeStatus` and the `SchemaModeUpdate` according to rules defined in Table 10.
Mono-VersionElementContent()	See 8.4.2.
Multiple-VersionElementContent()	See 8.4.3.

Table 9 — Schema Mode Update

Code Word	Schema Mode Update
0	mono_not_frozen
10	multi_not_frozen
110	mono_frozen
111	multi_children_frozen

Table 10 — ChildrenSchemaMode and ElementContentDecodingMode values

SchemaMode Update	SchemaMode Status	Children SchemaMode	ElementContent DecodingMode
mono_not_frozen	hot	hot	mono
multi_not_frozen	hot	hot	multi
mono_frozen	hot	frozen	mono
multi_children_frozen	hot	frozen	multi
-	frozen	frozen	mono

8.4.2 Mono-version element content

8.4.2.1 Syntax

Mono-VersionElementContent (Enumeration ChildrenSchemaMode, SchemaComponent theType) {	Number of bits	Mnemonic
if (lengthCodingMode == "Length optionally coded") {		
LengthFlag	1	bslbf
if (LengthFlag == 1) {		
TheLength	5+	vluimsbf5
}		
}		
else if (lengthCodingMode == "Length always coded") {		
TheLength	5+	vluimsbf5
}		

If (!PayloadTopLevelElement()) {		
SubstitutionCode()		
effectiveType = PayloadTypeCode(theType, false)		
} else {		
effectiveType=theType		
}		
if (effectiveType != “deferred” && effectiveType != ”nil”){		
if (useOptimisedDecoder(effectiveType)) {		
optimisedDecoder(effectiveType)		
} else {		
Attributes(effectiveType)		
Content(ChildrenSchemaMode, effectiveType)		
}		
}		
}		

8.4.2.2 Semantics

Name	*Definition*
LengthFlag	This flag specifies whether the length of this Mono-VersionElementContent is coded.
TheLength	The length in bits of the remainder of this Mono-VersionElementContent, excluding the Length function.
ChildrenSchemaMode	The `SchemaModeStatus` to be propagated to the children elements.
theType	The default element type i.e. the type associated to this element in the schema or the type passed by the context path if the element is the first element being decoded in the `FUPayload`.
PayloadTopLevelElement ()	Returns “true” if the element being decoded is the first one of the payload. In this case there is no need to decode the substitution code and the type code since they have already been decoded by the `FUContextPath`.
SubstitutionCode ()	Indicates the substitution information as specified in 7.6.5.3.
PayloadTypeCode ()	Indicates the type information as specified in subclause 8.4.5.
effectiveType	The effective type of the element. `effectiveType` shall be equal to the value of the `xsi:type` attribute, in case of a type cast, or else `effectiveType` shall be the default type.
useOptimisedDecoder()	Returns "true" if the type `effectiveType` is associated to an optimised type decoder as conveyed in the `DecoderInit` (refer to subclause 7.2).
optimisedDecoder ()	Triggers the optimised type decoder associated to the type `effectiveType` as conveyed in the `DecoderInit` (refer to subclause 7.2).

Attributes()	Decodes element attributes as specified in subclause 8.5.3.
Content()	Decodes element content as specified in subclause 8.4.6.

8.4.3 Multiple-version element content

8.4.3.1 Overview

In this case, the element is coded in several version-consistent bitstream chunks i.e. `ElementContentChunks`. All elements in an `ElementContentChunk` are decoded using a single schema. A schema identifier is present before each `ElementContentChunk`. These identifiers are generated on the basis of URIs conveyed in the `DecoderInit` (see 7.2). A `Length` is present when the element is coded in several `ElementContentChunks`, allowing the decoder to skip `ElementContentChunks` related to unknown schema.

NOTE The decoder keeps track of a `SchemaModeStatus`. It is used to improve coding efficiency. The decoder can "freeze" the schema needed to decode the description. In this case no overhead is induced by the multiple-version element coding for the elements contained in the element being decoded, i.e. the entire sub-tree.

8.4.3.2 Multiple version encoding of an element (informative)

Each XML element is associated to a type which defines its content model. Derived types are defined by restriction or extension of existing types. When managing different versions of a schema, a version 2 type might extend a version 1 type as shown in Figure 13. In this case, a multiple-version coding can be used to provide a forward compatible coding of this element. For example, the type T2.6 can be coded in two `ElementContentChunk`s. The first `ElementContentChunk` could encode those parts of T2.6 which were derived from T1.4 (see Figure 13). Encoding would be done exactly as if it were type T1.4.. The second `ElementContentChunk` then encodes the difference between types T1.4 and T2.6. A “Schema-1-decoder” will be able to decode the first part of the element content and skip the second part using the Length information.

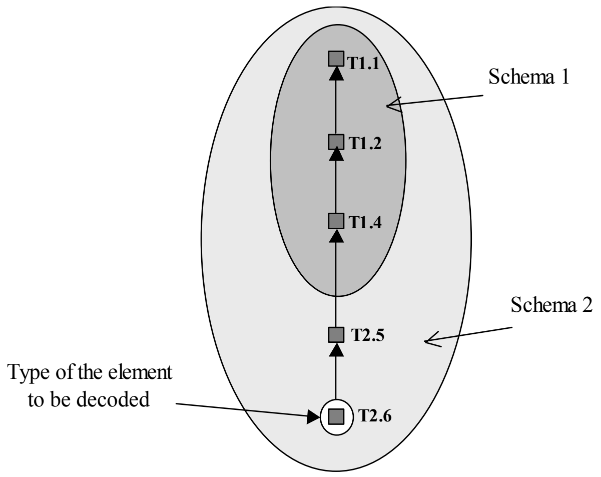


Figure 13 — Example of a type hierarchy defined across versions

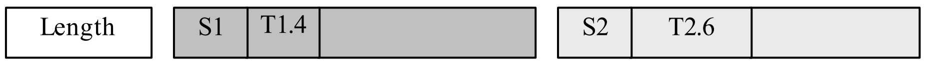


Figure 14 — Example of a forward compatible encoding

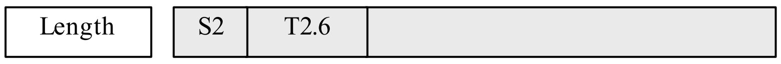


Figure 15 — Example of a non forward compatible encoding

8.4.3.3 Syntax

Multiple-VersionElementContent (Enumeration ChildrenSchemaMode, SchemaComponent defaultType) {	**Number of bits**	**Mnemonic**
Length	5+	vluimsbf5
SubstitutionFlag	1	bslbf
if (substitutionFlag) {		
SchemaIDOfSubstitution	ceil(log2(`NumberOfSchemas`))	bslbf
SubstituteElementCode	5+	vluimsbf5
}		
nextType = defaultType		
do {		
nextType = ElementContentChunk (ChildrenSchemaMode, nextType)		
} while (!endOfElement())		
}		

8.4.3.4 Semantics

Name	*Definition*
Length	The length in bits of the remainder of this Multiple-VersionElementContent, excluding the `Length` field.
SubstitutionFlag	A flag which specifies whether the element is substituted by another element which is a member of its substitution group (see XMLSchema – Part 1, Chapter 2.2.2.2).
SchemaIDOfSubstitution	The version identifier which refers to the schema where the substitute element is defined. Its value is the index of the URI in the `SchemaURI` array defined in 7.2.
SubstituteElementCode	The code of the substitute element. The code is computed following the rules defined in 7.6.5.3 using the schema identified by `SchemaIDOfSubstitution`. `SubstituteElementCode` shall be ignored if the schema identified by `SchemaIDOfSubstitution` is unknown to the decoder.
ElementContentChunk ()	A version-consistent chunk of the element related to a single schema as specified in subclause 8.4.4.
EndOfElement ()	Returns “true” if the content of the element is decoded completely, i.e. the number of decoded bits is identical to the number of bits coded in the Length field.

8.4.4 ElementContentChunk

8.4.4.1 Syntax

SchemaComponent ElementContentChunk (Enumeration ChildrenSchemaMode, SchemaComponent currentType) {	**Number of bits**	**Mnemonic**
SchemaID	ceil(log2(`NumberOfSchemas`))	bslbf
nextType = PayloadTypeCode(currentType, true)		
If(firstElementContentChunk()) {		
Attributes(nextType)		
Content(ChildrenSchemaMode, nextType)		
} else if (!restrictedType(currentType, nextType)) {		
AttributesDelta(currentType, nextType)		
ContentDelta(ChildrenSchemaMode, currentType, nextType)		
}		
return nextType		
}		

8.4.4.2 Semantics

An `ElementContentChunk` defines the decoding of one schema-consistent part of a multiple version encoded element.

Name	*Definition*
SchemaID	Identifies the schema which is needed to decode this `ElementContentChunk`. Its value is the index of the URI in the `SchemaURI` array defined in 7.2.
PayloadTypeCode ()	Decodes type information as specified in subclause 8.4.5. The set of types among which the type codes are assigned is the set of all types derived from the current type defined in the schema identified by `SchemaID`. The payload type code is progressively refined as the decoding of the element progresses. The last decoded type code defines the `xsi:type` attribute of the resulting current description tree.
firstElementContentChunk ()	Returns true if the `ElementContentChunk` being decoded is the first one of the `Multiple-VersionElementContent`.
Attributes ()	Decodes element attributes as specified in subclause 8.5.3 using the element type `nextType`.
Content ()	Decodes element content as specified in subclause 8.4.6 using the element type `nextType`.
AttributesDelta ()	Decodes the attributes added to the `currentType` by the derived `nextType`. If more than one type separates the two types in the type hierarchy, all the attributes added are gathered. The decoding process of these added attributes is done according to rules defined in subclause 8.5.3.

ContentDelta ()	Decodes the part of the complex content added to the `currentType` type by the derived `nextType`. If more than one type separates the two types in the type hierarchy, all the extensions are gathered. The decoding process is done according to rules defined in subclause 8.4.6.

8.4.5 PayloadTypeCode

8.4.5.1 Syntax

SchemaComponent PayloadTypeCode(SchemaComponent defaultType, boolean multi) {	**Number of bits**	**Mnemonic**
if (multi) {		
PayloadTypeIdentificationCode	ceil(log2(number of possible subtypes of defaultType + sizeIncrease))	bslbf
effectiveType = getDerivedType(defaultType, PayloadTypeIdentificationCode)		
}else if ((hasTypeCasting && hasNamedSubtypes(defaultType)) || hasDeferredNodes || elementNillable()) {		
PayloadTypeCastFlag	1	bslbf
if (PayloadTypeCastFlag == 1) {		
PayloadTypeIdentificationCode	ceil(log2(number of possible subtypes of defaultType + sizeIncrease))	bslbf
effectiveType = getDerivedType(defaultType, PayloadTypeIdentificationCode)		
}		
} else {		
effectiveType = defaultType		
}		
return effectiveType		
}		

8.4.5.2 Semantics

The Payload Type Code is used within the BiM Payload to indicate that a type cast occurred using the xsi:type attribute. It is also used to indicate if the element being decoded is a deferred element or a nil element.

Name	*Definition*
multi	A Boolean indicating whether the element is decoded in multiple version or mono version mode.

PayloadTypeIdentificationCode	The Type Identification Code is generated for a specific type (simpleType or complexType) from the set of all named derived types (itself being not included) of the default type in the current schema as specified in 7.6.5.4. The set of possible derived types is extended by the following rules: — If the element is not nillable and deferred elements are allowed, a "deferred" type is inserted at the first position in the set of all derived types i.e. its code is equal to 0, all other type codes are increased by 1 due to the `sizeIncrease` value. — If the element is nillable and deferred elements are not allowed, a "nil" type is inserted at the first position in the set of all derived types i.e. its code is equal to 0, all other type codes are increased by 1 due to the `sizeIncrease` value. — If the element is nillable and deferred elements are allowed, a "deferred" type is inserted at the first position in the set of all derived types i.e. its code is equal to 0 and a "nil" type is added at the first position in the set of all derived types i.e. its code is equal to 1, all other type codes are increased by 2 due to the `sizeIncrease` value.
PayloadTypeCastFlag	Indicates if a `PayloadTypeIdentificationCode` is defined.
sizeIncrease	Handles the increase in size of the set of possible subtypes due to the presence of "nil" and "deferred" types. Its value is set by the following rules: — If the element is not nillable and deferred elements are allowed, the `sizeIncrease` field is set to 1. — If the element is nillable and deferred elements are not allowed, the `sizeIncrease` field is set to 1. — If the element is nillable and deferred elements are allowed, the `sizeIncrease` field is set to 2.
elementNillable()	Returns true if the element being decoded is nillable i.e. its "nillable" property is equal to true (see XML Schema – Part 1, Chapter 3.3.1).
effectiveType	A SchemaComponent object representing the derived type of the type to be decoded.

8.4.6 Content

8.4.6.1 Syntax

Content(Enumeration ChildrenSchemaMode, SchemaComponent theType) {	Number of bits	Mnemonic
if (hasSimpleContent(theType)) {		
SimpleType(getSimpleContentType(theType))		
} else {		
ComplexContent(ChildrenSchemaMode, theType)		
}		
}		

8.4.6.2 Semantics

Name	*Definition*
SimpleType	See 8.4.7.
ComplexContent	Refers to the decoding process specified in subclause 8.5.2.

8.4.7 SimpleType

8.4.7.1 Syntax

SimpleType(SchemaComponent theType) {	**Number of bits**	**Mnemonic**
if (useOptimisedDecoder(theType)) {		
optimisedDecoder(theType)		
} else {		
defaultDecoder(theType)		
}		
}		

8.4.7.2 Semantics

Name	*Definition*
useOptimisedDecoder()	Returns "true" if the type `effectiveType` is associated to an optimised type decoder as conveyed in the `DecoderInit` (refer to subclause 7.2).
optimisedDecoder ()	Triggers the optimised type decoder associated to the type `effectiveType` as conveyed in the `DecoderInit` (refer to subclause 7.2).
defaultDecoder ()	Triggers the default decoder associated to the type "theType" as specified in subclause 8.5.4.1.

8.5 Element Content decoding process

8.5.1 Overview

The element content decoder relies on schema analysis. The schema analysis generates a "finite state automaton decoder" that model the decoding of a complex content. The use and construction of finite state automaton decoders is defined in subclause 8.5.2.

NOTE In this subclause, the automata-based approach replaces the usual C-like tables to specify syntax. The automata-based method is generic and its goal is to dynamically emulate such syntax tables rather than to statically define them. This specification does not mandate the decoder to be effectively implemented using automata.

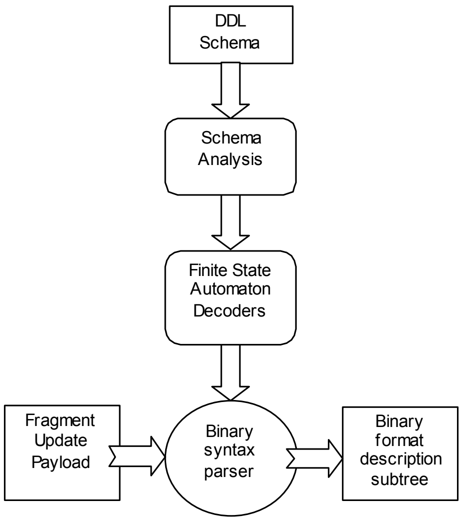


Figure 16 — The complex content decoding process

8.5.2 Complex content decoding process

8.5.2.1 Finite state automaton decoders

The decoding of every complex content is modeled by a finite state automaton decoder. A finite state automaton decoder is composed of "states" and "transitions". A transition is a unidirectional link between two states. A state is a receptacle for a token. There is only one token used during the decoding process. The location of the token indicates the current state of the automaton. The token can navigate from the current state to another state only through transitions. For each finite state automaton decoder there is one "start state" and one "final state". When a finite state automaton decoder is triggered by the `Content` syntax element defined in subclause 8.4.6 or the `ContentDelta` syntax element defined in subclause 8.4.4, the token is placed in the "start state". When the token reaches the "final state" the decoding of the complex content is finished.

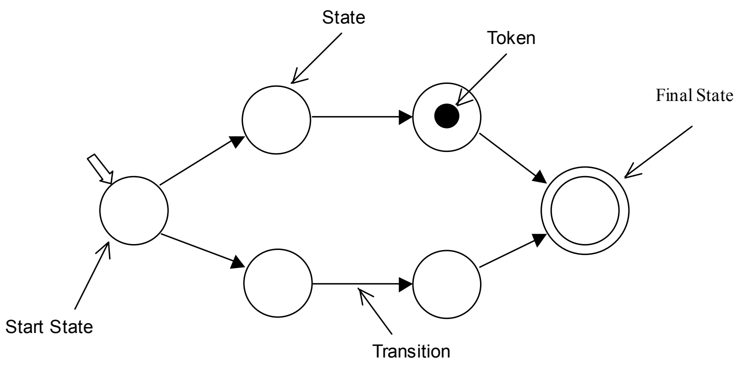


Figure 17 — Example of an automaton

A transition is "crossed" when a token moves from one state to another state through it. A state is "activated" when a token enters it. A behavior is associated to some transitions or states. This behavior is triggered when the transition is crossed or when the state is activated. There are different types of state and transition:

— *Element transitions*: Element transitions, when crossed, specifies to the decoder which element is present in the description.

- *Type states*: Type states, when activated, trigger type decoders.
- *Loop transitions:* Loop transitions are used to model the decoding of one or more element or group of elements. There are three different types of “Loop transitions”: the "loop start transition", the "loop end transition" and the "loop continue transition". These three loop transitions are always used together in an automaton.
  - *Loop start transitions*: Loop start transitions are crossed when there are many occurrences of some elements or groups of elements to be decoded.
  - *Loop continue transitions*: Loop continue transitions are crossed when there is at least one more element or group of elements to be decoded.
  - *Loop end transitions*: Loop end transitions are crossed when there are no more elements or group of elements to be decoded.
- *Code transitions*: Code transitions are associated to a binary code and a signature. Code transitions are crossed when their associated binary code is read from the binary description stream. Their binary code is deduced from their signature.
  - *Shunt transitions*: Shunt transitions are a special kind of code transitions. Their binary code value is always equal to 0.
- *Simple transitions and simple states*: simple transitions and simple states have no specific behavior, they are used to structure the automaton.

The construction of finite state automaton decoders is specified in subclause 8.5.2.2. The decoding process using finite state automaton decoders is specified in subclause 8.5.2.3. The behaviors of the above mentioned states and transitions are specified in subclause 8.5.2.4.

8.5.2.2 Finite state automaton decoder construction

8.5.2.2.1 Overview

This subclause specifies the process which constructs a finite state automaton decoder from the complex content of a complex type. The construction process is composed of 4 phases that are detailed in the subsequent subclauses.

- Phase 1 - Type content realization - This phase flattens complex type derivation. It realizes group references, element references.
- Phase 2 – Generation of the type syntax tree - This phase produces a syntax tree for the type’s complex content. This syntax tree is transformed in order to improve compression ratio.
- Phase 3 - Normalization of the type syntax tree - This phase normalizes the complex content’s syntax tree i.e. it associates a unique signature to every node of the syntax tree. These signatures are used in the following phase to generate binary codes used during the decoding process.
- Phase 4 - Finite State Automaton Decoder generation - This final phase produces the finite state automaton decoder used to decode the type’s complex content.

8.5.2.2.2 Phase 1 – Type content realization

During this phase, the type definition is analyzed in order to produce a realized type definition. A realized type is a “compiled” version of the type definition:

- The “effective content particle” of the type is the particle (see XML Schema – Part 1, Chapter 2.2.3.2) of the content type property generated for the type. It is specified in (see XML Schema – Part 1, Chapter 3.4.2). It is generated given the two following rules:
  - If the type is derived by extension from another type, the effective content particle of the type is appended, within a sequence group, to the effective content particle of its super type,
  - If the type is derived by restriction from another type, the effective content particle of the type is equals to its content,
- The “reference-free effective content particle” of the type is equal to its “effective content particle” where every element reference and group reference is replaced by its referenced definition,
- Each element and type name of the “reference-free effective content particle” is expanded i.e. their name is replaced by their expanded name (as defined in subclause 8.2)

8.5.2.2.3 Phase 2 - Syntax tree generation

8.5.2.2.3.1 Syntax tree definition

A syntax tree associated to the complex type is generated based on the “reference-free effective content particle” generated in phase 1. The syntax tree associated to the type is composed of different syntax tree nodes: element declaration nodes, group nodes and occurrence nodes. Element declaration nodes associate an element name to its type. They are leaves of the syntax tree and are derived from element declaration particles. Group nodes define a composition group (sequence, choice or all) and are derived from group particles. A group node contains only occurrence nodes. Occurrence nodes are derived from the ‘min occurs’ and ‘max occurs’ property of the particle and contain group nodes or element declaration nodes.

The syntax tree is reduced to improve the compression efficiency of the binary format by the transformations defined in subclauses 8.5.2.2.3.2, 8.5.2.2.3.3 and 8.5.2.2.3.4. These transformations simplify the content definition in a non destructive way i.e. the level of validation is not decreased by these transformations. In the following figures, occurrence nodes are represented by "[minOccurs, maxOccurs]", group nodes by the group names "sequence", "choice" or "all" and element declaration nodes by the element name followed by its associated type between brackets e.g. “anElementName {theElementType}”.

8.5.2.2.3.2 Group simplification

This rule applies to every group that contains only one syntax tree node (element or other group) whose minOccurs is equal to 0 or 1. In that case, the group is replaced by its content. Occurrences associated to the group nodes are multiplied as shown in Figure 18.

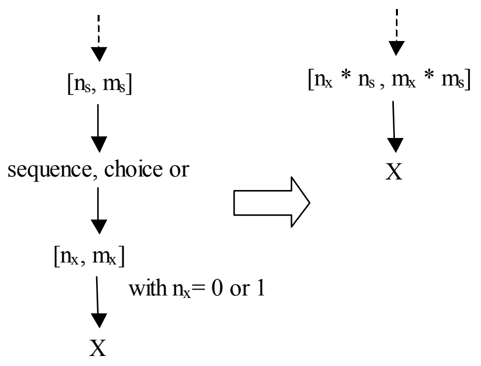


Figure 18 — Group simplification rule

8.5.2.2.3.3 Empty choice simplification

This rule applies to a choice when it contains at least one item (group or element) whose minOccurs equals 0. The minOccurs associated to the contained item is replaced by 1 and the minOccurs associated to the choice by 0.

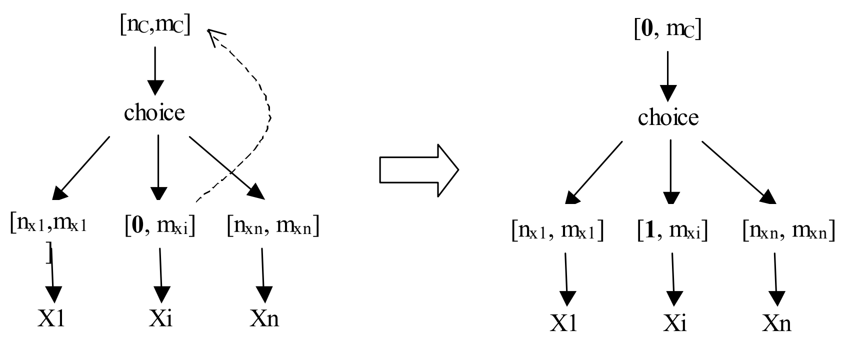


Figure 19 — Empty choice simplification rule

8.5.2.2.3.4 Choice Simplification

This rule applies when a choice contains another choice whose occurrence equals to 1. The child nodes of the inner choice are inserted in the outer choice.

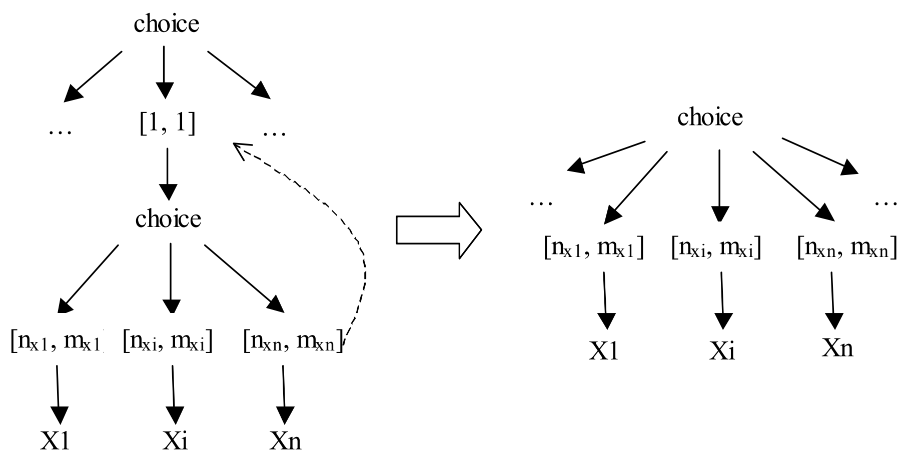


Figure 20 — Choice simplification rule

8.5.2.2.4 Phase 3 - Syntax tree normalization

Syntax tree normalization gives a unique name to every element declaration node and group node of the syntax tree. It allows to order the sibling nodes and assign a binary code to them. This code is used during the automata construction phase.

A signature is generated for every node of the syntax tree by the following rules:

— A group node signature is equal to concatenation of the character ':', the group key word (sequence, choice, all) and the "children signature" in that order. The "children signature" is defined by the concatenation of the signatures of the child nodes of the group node separated by the "white space" character. A "white space" character separates the group key word and the first child signature. In case of a "choice" or a "all", the children signatures are alphabetically sorted and then appended. In case of a "sequence", the children signatures are appended in the order of their definition in the schema,

— An occurrence node signature is equal to the signature of its child,

— Element declaration node signatures are equal to the expanded name of the element.

Example

Given the following complexType defined in the "http://www.mpeg7.org/example" namespace :

```
<complexType name="CoordinateMapping">
   <sequence maxOccurs="unbounded">
      <element name="pixel" type="IntegerVectorType"/>
      <choice>
         <element name="coordPoint" type="FloatVectorType"/>
         <element name="srcpixel" type="IntegerVectorType"/>
      </choice>
   </sequence>
   <element name="mappingFunct" type="mappingFunct"
            minOccurs="0" maxOccurs="unbounded"/>
</complexType>
```

The corresponding syntax tree is :

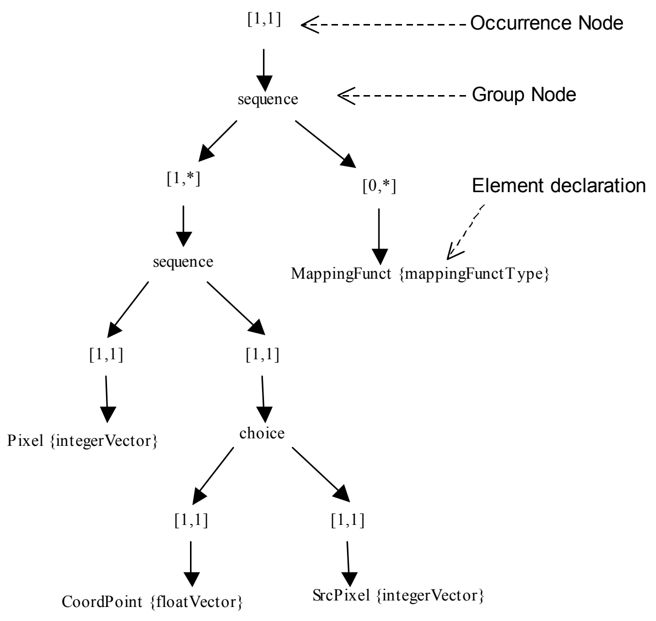


Figure 21 — Example - The syntax tree of coordinate mapping complexType

In this example:

— The signature of the choice is:

:choice http://www.mpeg7.org/example:CoordPoint http://www.mpeg7.org/example:Srcpixel

— The signature of the lower sequence is

:sequence http://www.mpeg7.org/example:Pixel :choice http://www.mpeg7.org/example:CoorPoint http://www.mpeg7.org/example:Srcpixel

— The signature of the upper sequence is

:sequence :sequence http://www.mpeg7.org/example:Pixel :choice
http://www.mpeg7.org/example:CoorPoint http://www.mpeg7.org/example:Srcpixel
http://www.mpeg7.org/example:MappingFunct

8.5.2.2.5 Phase 4 - Finite state automaton generation

8.5.2.2.5.1 Main Automaton Construction Procedure

A complex content automaton is recursively defined by the following rules. These rules are applied starting from the leaf nodes of the syntax tree up to the root node of the syntax tree:

- Every node of the content model syntax tree produces an automaton, short “node automaton”,
- The complex content automaton of the complex type to decode is the node automaton produced by the root node of its syntax tree,
- Every node automaton is generated by the merging of its child automata. The nature of the merging is dependent of the nature of the node as specified in 8.5.2.2.5.2,
- At the end of the process, automata are realized in order to associate binary codes to the "code transitions" (refers to subclause 8.5.2.1).

8.5.2.2.5.2 Phase 4.a - Automata construction

Element declaration node automaton

An automaton for an element declaration node is composed of two states, a start state and a final state, and a transition between them. It is used to specify the “element name” / “type” association declared in the complex type definition. The transition is an "element transition" to which the element name of the element declaration node is associated. The target state of the transition is a "type state" to which the element type of the element declaration node is associated.

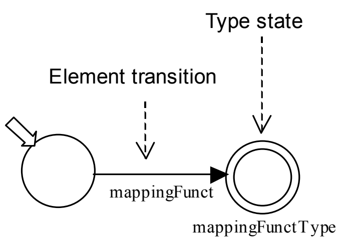


Figure 22 — Example of an element declaration node automaton

Occurrence node automaton

An occurrence node automaton is generated by adding loop transitions and states to its child node automaton. The transformation applied to the occurrence node child automaton depends on the minOccurs and maxOccurs values of the occurrence node:

- case a: if minOccurs = 1, maxOccurs = 1
  - no change to the child node automaton.
- case b: if minOccurs = 0, maxOccurs = 1

- two states are added to the child node automaton : a new start state and a new final state,
- a "Shunt transition" is added between the new start state and the new final state,
- a "Code transition" is added between the new start state and the old one, its signature is equal to the signature of the child node of this occurrence node,
- a simple transition is added between the old final state and the new one.

— case c: if maxOccurs > 1

  - two states are added to the child node automaton : a new start state and a new final state.
  - An intermediate simple state is added to the child node automaton. A "Code transition" is added between the new start state and the intermediate state. The signature of this "code transition" is equal to the signature of the child node of the occurrence node. A "Loop start transition" is added between the intermediate state and the old start state,
  - a "Loop end transition" is added between the old final state and the new one,
  - a "Loop continue transition" is added between the old final state and the old start state.

- case c-2: if minOccurs = 0

  - a "Shunt transition" is added between the new start state and the new final state.

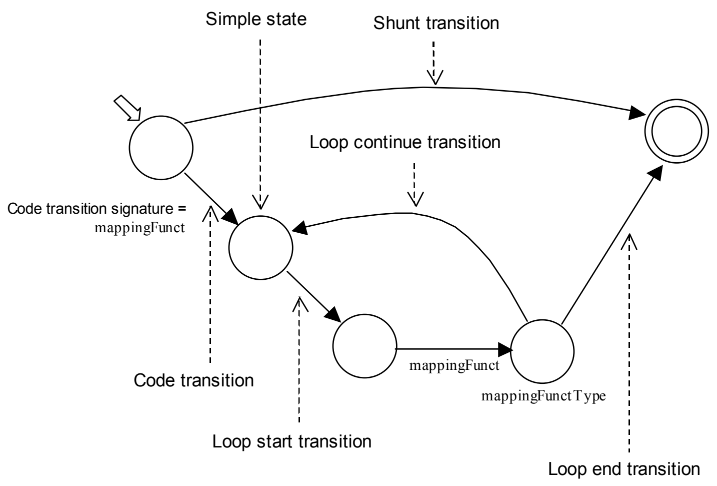


Figure 23 — Example of an occurrence node automaton

Choice node automaton

A choice automaton is built by the parallel merging of its child automata:

- Two new states are created : a new start state and a new final state,
- Code transitions are added between the new start state and every start state of its child nodes automata. The signatures of these code transitions are equal to the signature of their corresponding child node,
- Simple transitions are added between every final state of its children and its new final state.

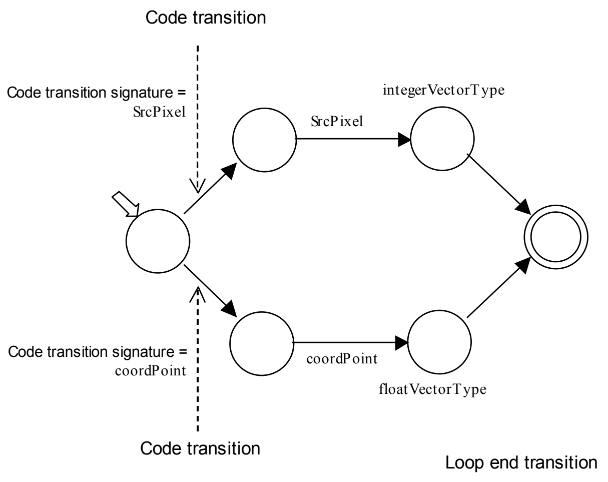


Figure 24 — Example of a choice node automaton

Sequence node automaton

A sequence node automaton is constructed by merging its children node automata. The merging is done in the order of the children nodes in the syntax tree (identical to the order of the sequence in the schema). A simple transition is added between the final state of a child node automaton and the start state of its following child node automaton in the sequence. The start state of the resulting automaton is the start state of the first child node automaton of the sequence. The final state of the resulting automaton is the final state of the last child node automaton of the sequence.

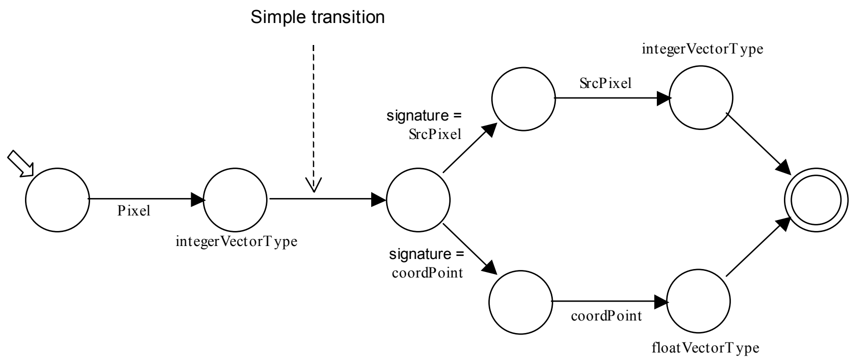


Figure 25 — Example of a sequence node automaton

All node automaton

The all node automaton is recursively constructed and forms a tree of choices. Every level of this tree is used to choose an element among those that are still possible.

A recursive method “allConstructionProcedure” is defined that receives two parameters as input: an ordered list of syntax tree nodes (noted `AllNodes`) and a state of an automaton (noted `PreviousFinalS`) and that returns a list of automaton states:

```
AllConstructionProcedure(AllNodes, PreviousFinalS) {
  Let "FinalStates" be an empty list of automaton states
  Let "ListStates" be an empty list of automaton states
  For each syntax tree node "X" of the list "AllNodes" {
    Generate "XA", the automaton of "X" using the rules defined in 8.5.2.2.5
    Let "StartXA" be the start state of the "XA" automaton and "FinalXA" the final
     state of the "XA" automaton
    Add a code transition between "PreviousFinalS" and "StartXA". The signature of
     this code transition is equal to the signature of "X".
    Let "RemainingNodes" be the copy of the "AllNodes" list in which "X" has been
     removed
    If the list "RemainingNodes" is empty {
      Return a new list of automaton states containing only "FinalXA"
    }
    "ListStates" = allConstructionProcedure(RemainingNodes, FinalXA)
    Add each automaton state of "ListStates" to "FinalStates"
  }
  Return "FinalStates"
}
```

An all node automaton is constructed by the following rules:

— Two new states are created : a new start state "NSS" and a new final state "NFS",

— Execute the "allConstructionProcedure" with all the list "AllChildNodes" (containing every child nodes of the all node) and the NSS:

  — `ListOfFinalStates = allConstructionProcedure(AllChildNode, NSS)`

— Create a simple transition between each states of the returned list and the new final state.

8.5.2.2.5.3 Phase 4.b - Code realization

This final phase transforms the "Code transition" signatures into binary codes. The binary code of a "code transition" is equal to its position in the alphabetically ordered list of "code transition" signatures starting from the same state. If there exists a "shunt transition", this "shunt transition" is always the first transition in this list i.e. its binary code value is always equal to 0. The length (in bits) of the binary codes associated to code transitions starting from the same state are equal to "ceil(log2(number of code transitions))".

8.5.2.3 Decoding a complex content using a finite state automaton decoder

The decoding of a complex content is done by the propagation of a token through the corresponding FSAD. Its propagation is guided by the binary description stream. When the token faces different possible paths, it consumes some bits from the description stream to identify the "code transition" which will guide it to the next state. The number of bits to read is equal to "ceil(log2(number of transitions starting from the state))".

When the token enters the final state of an automaton, the decoding of the corresponding type is finished and the reconstructed description sub-tree is returned to the decoder that triggered the FSAD.

8.5.2.4 Behavior of states and transitions

8.5.2.4.1 Code transition behavior

Code transitions are used to guide the token through the automata. A binary code is associated to each "code transition". When this binary code is read in the binary description stream, the token crosses the transition to reach the target state of the transition.

8.5.2.4.2 Type state behavior

When activated, a type state triggers the decoding of a contained element. The `Element` decoding method (see 8.4.1) is called with the `ChildrenSchemaMode` flag as the first parameter and the type associated to this type state as the second parameter.

8.5.2.4.3 Loop transitions behavior

Loop transitions (Loop start transition, loop end transition and loop continue transition) are used to model the decoding of one or more elements or groups of elements. Their behaviour is dependent on the UnitSize value conveyed in the `DecoderInit` (see 7.2). There are two possible cases for this:

1- The UnitSize is set to "default" according to Table 2,

In this case, when a "Loop start transition" is crossed, the decoder reads an integer (`NumberOfOccurrences`) from the stream which represents the number of times the associated "Loop continue transition" will be crossed. The `NumberOfOccurrences` field is decoded with respect to the minOccurs and maxOccurs values of the occurrence node from which the loop transitions have been generated:

— if maxOccurs - minOccurs > 65535 or maxOccurs = unbounded:

`NumberOfOccurrences` is decoded using vluimsbf5.

`NumberOfOccurrences` = `NumberOfOccurrences` + minOccurs

— if maxOccurs - minOccurs <= 65535:

`NumberOfOccurrences` is decoded with "ceil(log2(maxOccurs-minOccurs+1))" bits using uimsbf

`NumberOfOccurrences` = `NumberOfOccurrences` + minOccurs

2- The UnitSize is different from "default" according to Table 2,

In this case, an extra-bit called "continuation code" is utilized to signal the loop continuation. This bit directly precedes a set of repeated elements or groups of elements. Then,

— if the "continuation code" is equal to '1', it signals that the elements or groups of elements is repeated UnitSize more times i.e. that the loop continuation transition is crossed UnitSize more times,

— if the "continuation code" is equal to '0' and UnitSize is equal to ‘1’ then there is no more elements or groups of elements following i.e. the loop end transition is crossed,

— if the "continuation code" is equal to '0' and ‘UnitSize’ is greater than ‘1’, it is immediately followed by a uimsbf which signals the number of similar elements or groups of elements following i.e. the number of times the loop continuation transition is crossed:

— The length of this bit-field is computed the following way:

Bit-field length = ceil(log2(K))

Where

K = UnitSize, if maxOccurs is unbounded,

or

K = min(n, (maxOccurs-minOccurs) - num * n),

where,

num = Number of continuation bits with the value 1

n = UnitSize.

Example of loop transitions

The following figure illustrates the case where every pattern is preceded by a continuation code

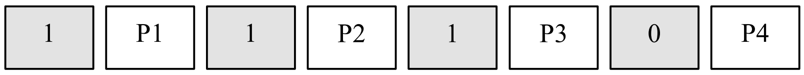


Figure 26 — Example of occurrence coding - UnitSize = 1

The following figure illustrates the nature of the continuation code, in case of 8 patterns when UnitSize = 4 and maxOccurs is unbounded:

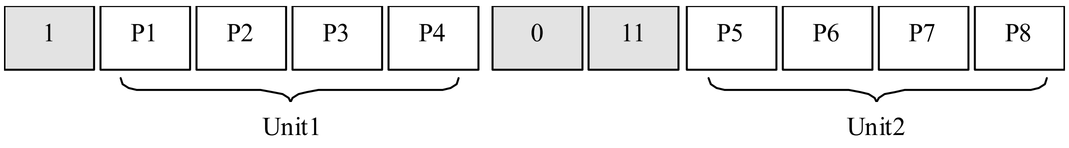


Figure 27 — Example of occurrence coding - the continuation code

As a final example, assume that maxOccurs of a particular element is bounded to 230, and the unit size is 32. If the instance document contains all the 230 elements, then 7 units of completely filled blocks will be conveyed, which amounts to 224 elements. After that a partial block follows. Since only 6 elements are left, the continuation code for the length of this partial block consumes 3 bits, rather than 5 bits.

8.5.3 Attributes decoding process

The attributes of an element are decoded with the following rules:

— All the attributes definitions are collected from the super types of the complex type,

— All the attribute definitions defined as FIXED in the schema are suppressed,

— All the attribute definitions are lexicographically ordered using their expanded name.

A sequence automaton is generated and used to decode the attributes. As a result, attributes are decoded by set of consecutive patterns. These patterns are composed of two components:

— an attribute flag, which defines the presence or the absence of an optional attribute,

— an attribute value decoded as defined in subclause 8.4.7.

The 1-bit attribute flag is only present for optional attributes. It is equal to 0 if the attribute is not coded or 1 if the attribute is coded. It is not present for mandatory attributes.

8.5.4 Simple types decoding process

8.5.4.1 Primitive simple types

Decoding of the following DDL primitive datatypes occurs as specified below:

- boolean is coded by one bit, 1 = true, 0 = false
- float is coded as a IEEE 754 floating-point "single precision"
- double is coded as a IEEE 754 floating-point "double precision"
- hexBinary is coded as a binary stream, preceded by its size in bits coded using vluimsbf5
- base64Binary is coded as a binary stream, preceded by its size in bits coded using vluimsbf5

The following DDL primitive datatypes are coded as a UTF-8 string preceded by its size in bytes coded as vluimsbf5:

- decimal
- string
- duration
- dateTime
- time
- date
- gYearMonth
- gYear
- gMonthDay
- gDay
- gMonth
- anyURI
- QName

8.5.4.2 Specific simple types decoders

8.5.4.2.1 Integers

Restricted integers are decoded using a specific datatype decoder. This decoder uses the “minExclusive”, “maxExclusive”, “minInclusive” and “maxInclusive” facets of the simpleType to deduce the coding length of the integer :

- If the simple type is restricted both in its minimum and maximum value, the decoding process is the following:
    - if (minInclusive is defined) then min = minInclusive

    - if (minExclusive is defined) then min = minExclusive+1
    - if (maxInclusive is defined) then max = maxInclusive
    - if (maxExclusive is defined) then max = maxExclusive-1
    - if (max-min <= 65535) then
      - value is decoded with ceil(log2(max-min+1)) bits as a uimsbf
      - value = value + min
    - if (max-min > 65535) then
      - value is decoded using vluimsbf5
      - value = value + min
- else the value is decoded by :
  - one bit to indicate the sign (0 : positive, 1 : negative), followed by
  - abs(value) is decoded using vluimsbf5

8.5.4.2.2 Enumeration

Every enumerated simple type is decoded using an lexicographically sorted dictionary of all the possible enumeration values. The decoder reads an integer, using uimsbf coding whose bit size is equal to "ceil(log2(number of possible values))". The decoded value is the entry corresponding to this integer in the lexicographically sorted dictionary.

8.5.4.2.3 Lists

A list simple type is decoded in two steps.

The first step decodes the length of the list (in number of items) using the following rules:

- if maxLength - minLength <= 65535
  - `length` is decoded with "ceil(log2(maxLength-minLength+1))" bits using uimsbf,
  - `length` = `length` – minLength.
- if maxLength - minLength > 65535 or maxLength is not constrained
  - `length` is coded using vluimsbf5,
  - `length` = `length` – minLength.

The second step decodes each item of the list using its simple type decoder as specified in 8.4.7.

8.5.4.2.4 Union

An union simple type is decoded in two steps. The first step decodes a type identification code. The second decodes the value itself. The type identification code is assigned among the set of the union member types (See XML Schema – Part 2, Chapter 2.5.1.3). The codes are assigned in the order of the union member types definition. The coding length is equal to "ceil(log2(number of union member types))". The `SimpleType` syntax element (see

8.4.7) is then used for the decoding. The type identified by the type code is used as the "SchemaComponent" parameter.

8.5.4.2.5 basicTimePointType

In the ISO/IEC 15938-2 the basicTimePointType is specified by the following regular expression:

$$\underbrace{(\underbrace{\backslash -?}_{1.1}\backslash d+\underbrace{(\backslash -\backslash d\{2\}\underbrace{(\backslash -\backslash d\{2\})?}_{1.2.1})?}_{1.2})?}_{1}\underbrace{(T\backslash d\{2\}\underbrace{(:\backslash d\{2\}\underbrace{(:\backslash d\{2\}\underbrace{(:\backslash d+\underbrace{(\backslash .\backslash d\{2\})?}_{2.1.1.1.1})?}_{2.1.1.1})?}_{2.1.1})?}_{2.1})?}_{2}\underbrace{(F\backslash d+)?}_{3}\underbrace{((\backslash -|\backslash +)\backslash d\{2\}:\backslash d\{2\})?}_{4}$$

Figure 28 — Regular expression of the basicTimePointType datatype

Each optional section of the regular expression is specified in Figure 28 by a number. In the binary syntax definition (see syntax table below) a flag which is named according to these numbers signals which of these optional sections is present in an instantiation of the basicTimePointType. In the second part of that table the coding of the values of each section is specified. The values are named according to the numbers of the section.

basicTimePointType() {		
flag1	1	bslbf
if (flag1){		
flag1.1	1	bslbf
flag1.2	1	bslbf
if (flag1.2){		
flag1.2.1	1	bslbf
}		
}		
flag2	1	bslbf
if (Flag2){		
flag2.1	1	bslbf
if (flag2.1){		
flag2.1.1	1	bslbf
if (flag2.1.1) {		
flag2.1.1.1	1	bslbf
if (flag2.1.1.1) {		
flag2.1.1.1.1	1	bslbf
}		
}		
}		
}		
flag3	1	bslbf
flag4	1	bslbf
if (flag1){		
\d+	5+	vluimsbf5
if (flag1.2)		

Value1.2, \d{2}, value range 1-12	4	uimsbf
if (flag1.2.1){		
Value1.2.1\d{2}, value range 1-31	5	uimsbf
}		
}		
}		
if (flag2) {		
Value2, \d{2}, value range 0-23	5	uimsbf
if (flag2.1) {		
Value2.1, \d{2}, value range 0-59	6	uimsbf
if (flag2.1.1) {		
Value2.1.1, \d{2}, value range 0-59	6	uimsbf
}		
}		
}		
if (flag3) {		
Value3, \d+	5+	vluimsbf5
}		
if (flag2) {		
if (flag2.1) {		
if (flag2.1.1) {		
if (flag2.1.1.1) {		
Value2.1.1.1, \d+	ceil(log2(Value3))	uimsbf
if (flag2.1.1.1.1) {		
Value2.1.1.1.1, \d{2}, value range 0-99	7	uimsbf
}		
}		
}		
}		
}		
if (flag4)		
{		
Value4 (Hours), '+' | '-'\d{2}, value range –15-+16	5	bslbf
Value4 (Minutes), \d{2}, value range 0-45, increment 15	2	bslbf
}		
}		

Derived simple types such as timePointType based on basicTimePointType which further restrict the regular expression by omitting optional sections are encoded using the same syntax. But the omitted sections are signalled by setting the flag of the omitted section to “0”.

8.5.4.2.6 basicDurationType

In the ISO/IEC 15938-2 the basicDurationType is specified by the following regular expression:

$$\underbrace{\backslash\text{-}}_{1}?P\underbrace{(\backslash d+D)?}_{2}\underbrace{(T\underbrace{(\backslash d+H)?}_{3.1}\underbrace{(\backslash d+M)?}_{3.2}\underbrace{(\backslash d+S)?}_{3.3}\underbrace{(\backslash d+N)?}_{3.4}\underbrace{(\backslash d\{2\}f)?)?}_{3.5}}_{3}\underbrace{(\backslash d+F)?}_{4}\underbrace{((\backslash\text{-}\,|\,\backslash+)\,\backslash d\{2\}:\backslash d\{2\}Z)?}_{5}$$

Figure 29 — Regular expression of the basicDurationType datatype

Each optional section of the regular expression is specified in Figure 29 by a number. In the binary syntax definition (see syntax table below) a flag which is named according to these numbers signals which of these optional sections is present in an instantiation of the basicDurationType. In the second part of that Table the coding of the values of each section is specified. The values are named according to the numbers of the section.

basicDurationType() {		
flag1	1	bslbf
flag2	1	bslbf
flag3	1	bslbf
if (Flag3){		
flag3.1	1	bslbf
flag3.2	1	bslbf
flag3.3	1	bslbf
flag3.4	1	bslbf
flag3.5	1	bslbf
}		
flag4	1	bslbf
flag5	1	bslbf
if (flag2){		
Value2, \d+	5+	vluimsbf5
}		
if (flag3) {		
if(flag3.1) {		
if(flag2){		
Value3.1, \d+, value range 0-23	5	uimsbf
}		
else {		
Value3.1, \d+	5+	vluimsbf5
}		
}		
if(flag3.2) {		
if(flag3.1){		
Value3.2, \d+, value range 0-59	6	uimsbf
}		
else {		
Value3.2, \d+	5+	vluimsbf5
}		
}		
if (flag3.3) {		

if(flag3.2){		
Value3.3, \d+, value range 0-59	6	uimsbf
}		
else {		
Value3.3, \d+	5+	vluimsbf5
}		
}		
}		
if (flag4) {		
Value4, \d+	5+	vluimsbf5
}		
if (flag3) {		
if (flag3.4) {		
if(flag3.3){		
Value3.4, \d+	ceil(log2(Value4))	uimsbf
}		
else {		
Value3.4, \d+	5+	vluimsbf5
}		
}		
if (flag3.5) {		
Value3.5, \d{2}, value range 0-99	7	uimsbf
}		
}		
if (flag5)		
{		
Value5 (Hours), '+' | '-'\d{2}, value range –31-+32	6	bslbf
Value5 (Minutes), \d{2}, value range 0-45, increment 15	2	bslbf
}		
}		

Derived simple types such as durationType based on basicDurationType which further restrict the regular expression by omitting optional sections are encoded using the same syntax. But the omitted sections are signalled by setting the flag of the omitted section to "0".

Annex A
(informative)

Informative examples

A.1 Flexible transmission of descriptions

One of the main advantages of the BiM structure is to offer a high level of flexibility through the use of navigation commands with absolute and relative paths while using an optimized encoding of sub-trees. The instance tree is thus first divided into sub-trees with a flexible granularity. The encoder has the freedom to define points in the instance structure that will be located globally (or relatively for sake of compactness) in order to allow fast access to this given nodes and possible re-synchronisation. This allows the BiM stream to be scalable in the sense that the sub-trees can be transmitted or stored in any order, easing the access to the most important information in a given application context. 3 use case scenarios are described below as different examples to illustrate the need for different levels of fast access granularity.

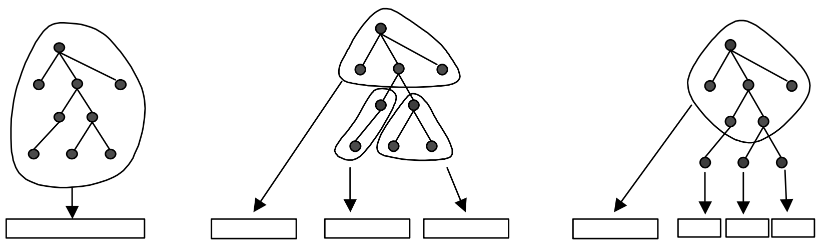


Figure A.1 — Different streaming strategies

A.2 Example of the construction of a finite state automaton decoder

The following subclause presents an example of finite state automaton decoder generation.

A.2.1 Definition of the CoordinateMappingType

Given the following complexTypes defined in the "http://www.mpeg7.org/example" namespace :

```
<complexType name="pixelCPointType">
 <sequence maxOccurs="unbounded">
  <element name="pixel" type="IntegerVectorType"/>
  <choice>
   <element name="coordPoint" type="FloatVectorType"/>
   <element name="srcpixel" type="IntegerVectorType"/>
  </choice>
 </sequence>
</complexType>

<complexType name="CoordinateMappingType">
 <extension base="pixelCPointType">
  <element name="mappingFunct" type="mappingFunct"
                   minOccurs="0" maxOccurs="unbounded" />
   </extension>
</complexType>
```

A.2.2 Effective content particle of type CoordinateMappingType

The effective content particle is an abstract object, which is represented in this example in an XML schema manner:

```
<sequence minOccurs="1" maxOccurs="1">

 <sequence minOccurs="1" maxOccurs="unbounded">
   <element name="pixel" type="IntegerVectorType"/>

   <choice minOccurs="1" maxOccurs="1">
    <element name="coordPoint" type="FloatVectorType" minOccurs="1" maxOccurs="1"/>
    <element name="srcpixel" type="IntegerVectorType" minOccurs="1" maxOccurs="1"/>
   </choice>
 </sequence>

 <element name="mappingFunct" type="mappingFunct"
                                           minOccurs="0" maxOccurs="unbounded"/>
</sequence>
```

A.2.3 CoordinateMappingType syntax tree after type realization

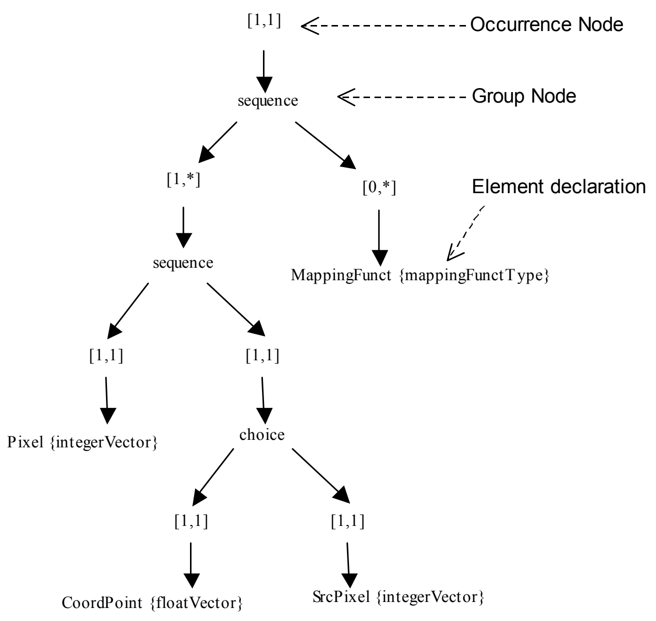


Figure A.2 — The syntax tree of CoordinateMappingType

A.2.4 Syntax tree transformations

No syntax tree transformation can be applied on this example.

A.2.5 Signatures of the syntax tree nodes of the CoordinateMappingType

— The signature of the choice is:

:choice http://www.mpeg7.org/example:CoordPoint http://www.mpeg7.org/example:Srcpixel

— The signature of the lower sequence is

:sequence http://www.mpeg7.org/example:Pixel :choice http://www.mpeg7.org/example:CoorPoint http://www.mpeg7.org/example:Srcpixel

— The signature of the upper sequence is

:sequence :sequence http://www.mpeg7.org/example:Pixel :choice http://www.mpeg7.org/example:CoorPoint http://www.mpeg7.org/example:Srcpixel http://www.mpeg7.org/example:MappingFunct

A.2.6 Finite state automaton decoder of the CoordinateMappingType complex content

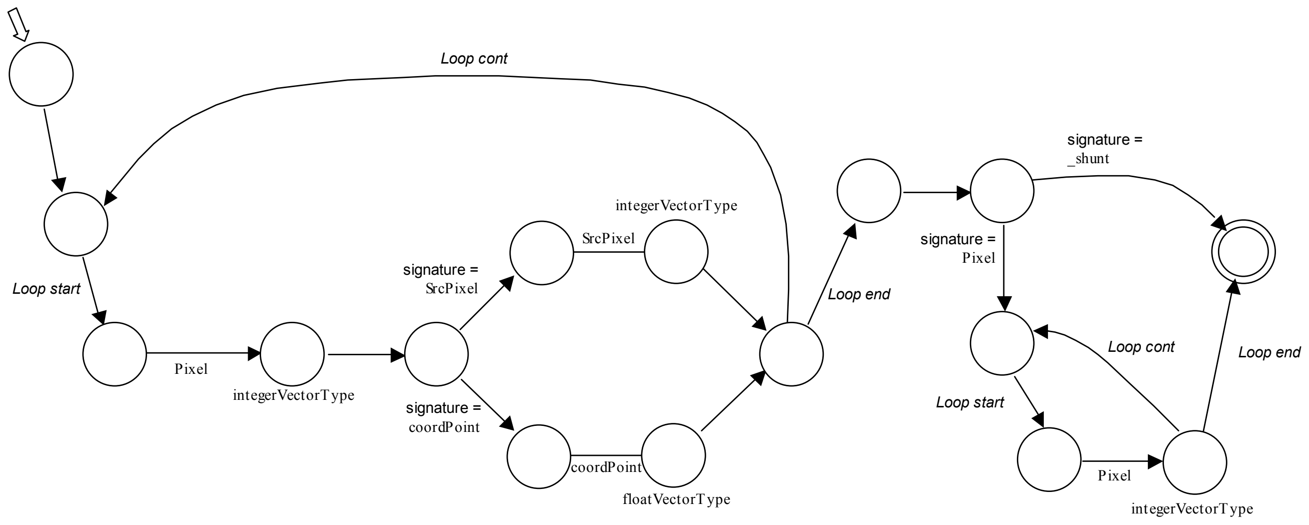


Figure A.3 — A Finite State Automaton Decoder for the CoordinateMappingType

Annex B
(informative)

FUContextType definition based on W3C XPath specification

The FUContextType specifications is a subsets of XPath expressions specified in http://www.w3.org/TR/1999/REC-xpath-19991116 with respect to the axis and predicates.

The subset specified for the FUContextType allows the selection of a node in the Current Description. In the following, the subset is defined by using the EBNF with corresponding line numbering to http://www.w3.org/TR/1999/REC-xpath-19991116:

FUContext := LocationPath

[1] LocationPath ::= RelativeLocationPath | AbsoluteLocationPath

[2] AbsoluteLocationPath ::='/' RelativeLocationPath

[3] RelativeLocationPath ::= (Step '/')* (Step | '@' NameTest)

[4] Step ::= NodeTest Predicate | AbbreviatedStep

[5]

[6]

[7] NodeTest ::= NameTest

[8] Predicate ::= '[' Number ']'

[9]

[10]

[11]

[12] AbbreviatedStep ::= '.' | '..'

[13]

[14]

[15]

[16]

[17]

[18]

[19]

[20]

[21]

```
[22]

[23]

[24]

[25]

[26]

[27]

[28]

[29]

[30]   Number    ::=    Digits

[31]   Digits    ::=    [0-9]+

[32]

[33]

[34]

[35]

[36]

[37]   NameTest    ::=         QName

[38]

[39]
```

QName is specified according to XPath recommendation.

Annex C
(informative)

Patent statements

The International Organization for Standardization and the International Electrotechnical Commission (IEC) draw attention to the fact that it is claimed that compliance with this part of ISO/IEC 15938 may involve the use of patents.

ISO and IEC take no position concerning the evidence, validity and scope of these patent rights.

The holders of these patent rights have assured ISO and IEC that they are willing to negotiate licences under reasonable and non-discriminatory terms and conditions with applicants throughout the world. In this respect, the statements of the holders of these patent rights are registered with ISO and IEC. Information may be obtained from the companies listed below.

Attention is drawn to the possibility that some of the elements of this part of ISO/IEC 15938 may be the subject of patent rights other than those identified in this annex. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Company	MPEG-7 Patent statements					
	Systems	DDL	Visual	Audio	MDS	Reference software
Bosch	X	X	X	X	X	X
Canon				X		
CIE	X	X	X	X	X	X
Denso	X				X	
Ericsson	X	X	X	X	X	X
ETRI	X	X	X	X	X	X
Expway	X	X	X	X	X	X
FhG				X		
Geocast	X	X	X	X	X	
HHI			X			X
Hitachi	X	X	X	X	X	X
Hyundai			X		X	X
IBM	X	X	X	X	X	X
JVC	X	X	X	X	X	X
KDDI			X	X	X	X
LG Electronics	X	X	X	X	X	X
Matsushita	X	X	X	X	X	X
Mitsubishi Electric	X	X	X	X	X	X
NEC						

NHK	X	X	X	X	X	X
Philips	X	X	X	X	X	X
Ricoh	X	X	X	X	X	X
Samsung	X	X	X	X	X	X
Sharp	X		X	X	X	X
Siemens	X	X	X	X	X	X
Sony	X	X	X	X	X	X
Toshiba	X	X	X	X	X	X
Vivcom					X	X

Bibliography

[1] ISO/IEC 11172 (all parts), *Information technology — Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s*

[2] ISO/IEC 13818 (all parts), *Information technology — Generic coding of moving pictures and associated audio information*

[3] ISO/IEC 14496 (all parts), *Information technology — Coding of audio-visual objects*

[4] ISO/IEC 15938-2, *Information technology — Multimedia content description interface — Part 2: Description definition language*

[5] ISO/IEC 15938-3, *Information technology — Multimedia content description interface — Part 3: Visual*

[6] ISO/IEC 15938-4, *Information technology — Multimedia content description interface — Part 4: Audio*

[7] ISO/IEC 15938-6, *Information technology — Multimedia content description interface — Part 6: Reference software*

[8] *UTF-8, a transformation format of ISO 10646,* IETF RFC 2279. F. Yergeau. January 1998

AMENDMENT 1: Systems extensions

Add the following reference in Clause 2:

- *RFC 1950, ZLIB Compressed Data Format Specification version 3.3.*

Add the following definition at the end of subclause 3.1.2.2.5:

ReservedBitsZero: a binary syntax element whose length is indicated in the syntax table. The value of each bit of this element shall be "0". These bits may be used in the future for ISO/IEC defined extensions.

Add the following definitions to subclause 3.2 (keep alphabetical order):

initial schema
The schema that is known by the decoder before the decoding process starts.

additional schema
A schema that can be updated after the start of the decoding process.

schema update unit
Information in an access unit, conveying a schema or a portion thereof. Schema update units provide the means to modify the current decoder schema knowledge.

description fragment reference
A reference to a description fragment.

Note - For instance, a fragment reference can be a URI which serves to locate the fragment on the world wide web.

fragment reference
short term for description fragment reference.

fragment reference resolver
An entity that is capable of resolving the fragment reference provided in the fragment update payload.

fragment reference marker
A specific information used to describe a deferred fragment reference, which is present within the current description tree. It consists of a fragment reference, the name and type of the top most element of the referenced fragment.

fragment reference format
An encoding format of fragment references.

deferred fragment reference
A fragment reference that can be resolved at any time by the application using the terminal.

non-deferred fragment reference
A fragment reference that shall be resolved by the terminal at the composition time of the access unit containing the fragment reference.

optimised decoder
A decoder associated to a type and dedicated to certain encoding methods better suited than the generic ones.

type codec
Synonym to optimised decoder.

fixed optimised decoder
An optimised decoder used to decode either a complex type or a simple type. Fixed optimised decoders are set up at decoder initialisation phase and their mapping to types can't be modified during binary description stream lifetime.

advanced optimised decoder
An optimised decoder used to decode a simple type. Advanced optimised decoders parameters and their mappings to types can be modified during binary description stream lifetime.

advanced optimised decoder instance
An advanced optimised decoder initialised and ready to be used for the decoding of some data types.

Note - There can be several instances of the same advanced optimised decoder with different or identical parameters.

advanced optimised decoder type
The type, identified by a URI, of an advanced optimised decoder.

advanced optimised decoder instances table
A table of all the advanced optimised decoders available at a certain instant in time.

contextual optimised decoder
An optimised decoder which behavior is dependent on the current context of the decoding.

Note - For instance, the ZLib optimised decoder (see Clause 9) is a contextual optimised decoder.

Note - Upon certain events, the context must be reset. Upon a certain command or events they are flushed to release their contents. Only contextual optimised decoders are flushable.

advanced optimised decoder parameters
The parameters of an advanced optimised decoder.

contextual optimised decoder reset
An operation that resets the optimised decoder to put it in a defined initial state. All contextual information is discarded.

skippable subtree
A subtree of an XML document that the decoder is permitted not to decode.

optimised decoder mapping
An association between a type and a set of optimised decoders.

Renumber all definitions in subclause 3.2.

Add the following abbreviations to the table in subclause 4.1:

MSB	Most Significant Bit
SU	Schema Update
SUU	Schema Update Unit

Add the following mnemonic to subclause 4.3:

Name	*Definition*
vlurmsbf5	Variable length code unsigned rational number, most significant bit first. The first n bits (Ext) which are ‘1’ except of the nth bit which is ‘0’, indicate that the rational number R in the interval 0≤R<1 is encoded by n times 4 bits. The ith bit of the n times 4 bits representing the rational number corresponds to a value of 2^-i. Thus the (n+1)st bit of the vlurmsbf5 code word (which corresponds to the MSB of the rational number) represents a value of ½, the (n+2)nd bit of the vlurmsbf5 code word represents a value of ¼., and so forth. An example for this type is shown in Figure AMD1-1. Note - Comparing two rational numbers A and B represented by a vlurmsbf5 code word can be done by comparing bit by bit the rational numbers starting from their respective MSBs. Then the rational number A is bigger if there is a ‘1’ bit at a position at which there is a ‘0’ for B. A is also bigger if there is a ‘1’ bit at a position which is not present for B and when A is longer than B.

In subclause 4.3, add the following figure after Figure 2:

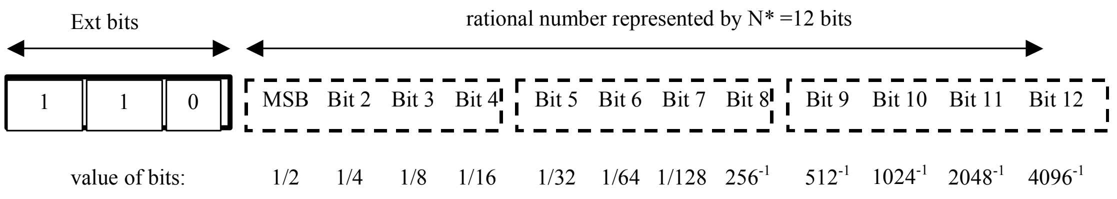


Figure AMD1-1 - Informative example for the vlurmsbf5 data type

Remove the ‘(informative)’ in the title of subclause 5.2.3.

In subclause 5.2.3, replace this paragraph:

The BiM, in order to gain its compression efficiency, relies on a schema analysis phase. During this phase, internal tables are computed to associate binary code to XML elements, types and attributes.This principle mandates the full knowledge of the same schema by the decoder and the encoder for maximum interoperability.

by

The BiM, in order to gain its compression efficiency, relies on a schema analysis phase. During this phase, internal tables are computed to associate binary code to schema components (XML elements, types and attributes). BiM defines two methods to address schema components.

The first method allows the decoder to resolve a schema, possibly including schema components originating from several namespaces, at initialization phase. This set of schema components form the *initial schema.* In this schema, all type and substitution codes are merged together no matter the namespace they belong to. This results in shorter codes in the binary description stream. The initial schema can't be updated and is considered fixed for the binary description stream lifetime. It contains by default and at minimal the type codes of the xml schema types: anyType, anySimpleType, and all xml schema simple types. In this specification, anySimpleType is considered as a subtype of anyType.

The second method allows the decoder both to resolve a schema at initialization phase (the *initial schema*) and to receive updated schema information called *additional schemas*. Additional schemas differ from the initial schema as the codes of their schema components defined in different namespaces are defined in different code spaces. This results in larger code size but has the required flexibility for late updating. For full flexibility it is also possible to receive exclusively additional schemas and thus to operate without initial schema.

Both initial schemas and additional schemas are part of a unique table in which each entry identifies a specific schema. The first entries identify schemas that are part of the initial schema. The following ones identify additional schemas.

To further improve compression, BiM allows the association of specific codecs to specific data types instead of using the generic mechanisms defined in Clause 8. These encoding schemes can be optimised with the full knowledge of the properties of that data type.

In subclause 5.2.3, replace this paragraph:

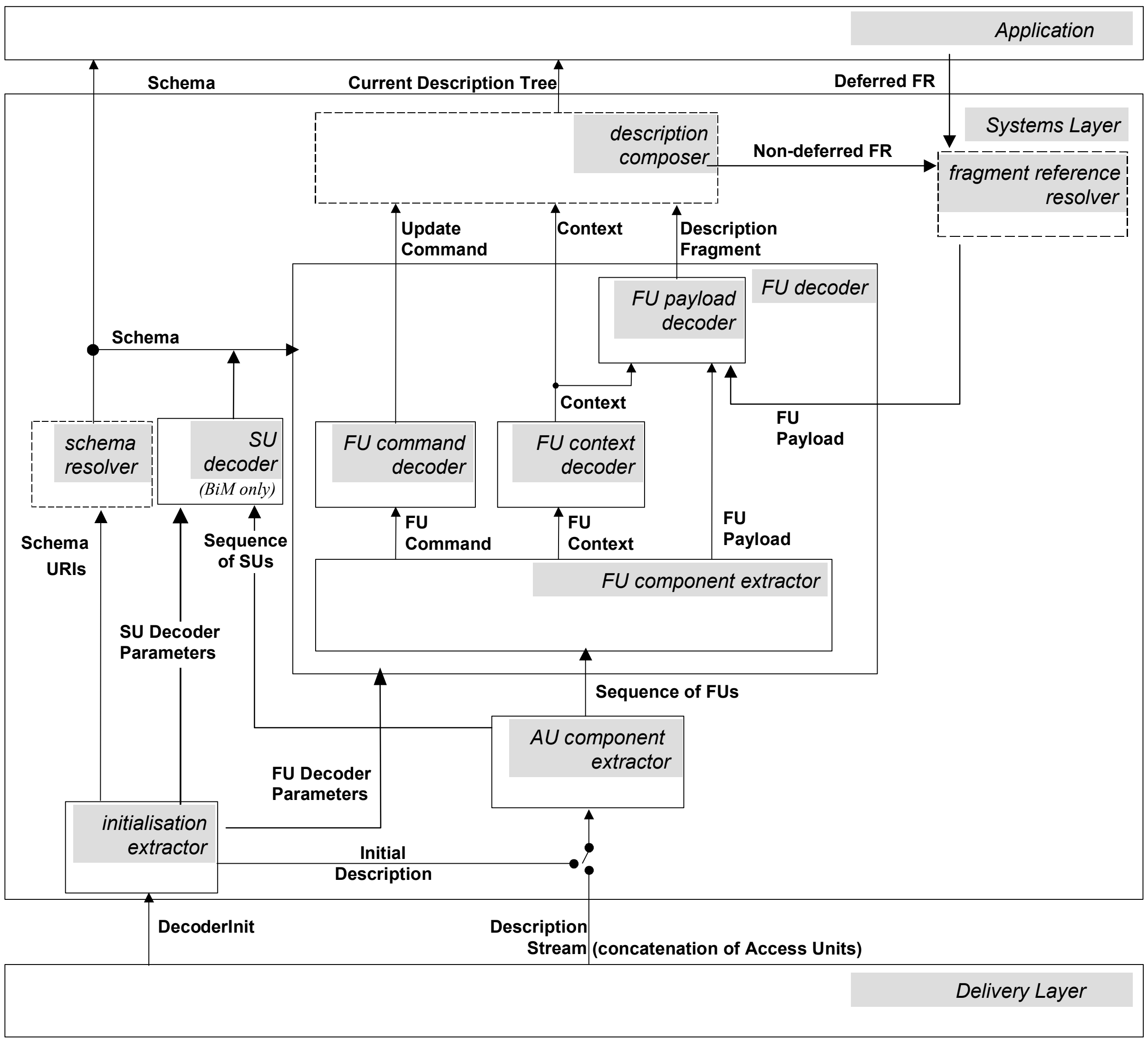
As with the textual decoder, the resulting current description tree may be topologically equivalent to the original description if desired by the encoder, but it may also exhibit dynamic characteristics such that certain parts of the description are present at the decoder only at chosen times, are never present at all, can be acquired on application demand, or appear in a different part of the tree.

by

As with the textual decoder, the resulting current description tree may be topologically equivalent to the original description if desired by the encoder, but it may also exhibit dynamic characteristics such that certain parts of the description are present at the decoder only at chosen times, are never present at all, can be acquired on application demand, or appear in a different part of the tree.

Note - The schema update capabilities provided by the BiM framework defined in this specification aims at upgrading BiM decoder. It is not a mean to transmit an XML schema as is. To do so, one should use the W3C schema of schema to encode its schema.

In subclause 5.3, replace the "Figure 4 - Terminal Architecture" by the following figure:



In subclause 5.3, second paragraph, replace:

…(FU Decoder Parameters, in Figure …

by

…(FU Decoder Parameters and SU Decoder Parameters, in Figure …

In subclause 5.4, add the following text after the first paragraph:

In the case of BiM, an access unit is composed of any number of schema update units followed by any number of fragment update units which are extracted by the access unit component extractor.

A schema update unit carries parts of an additional schema and is composed of

— a namespace identifier,

— a set of code tables to represent global elements, global types and global attributes,

— a binary encoded schema carrying the schema components definitions.

The full schema is not always necessary for the decoding of a particular binary description stream. To avoid unnecessary transmission, a schema update unit may contain only the definitions that are required for the decoding of the bitstream. In this case the code tables can also be sent partially.

Some further constraints are applied to the acquisition of schema update units, notably to ensure that a decoder will not break in case of a missed schema update unit. A specific schema update unit, the so-called first schema update unit, contains initialization information and shall be acquired by the decoder before any use of a received definition. The decoder behavior in case of such missed schema update units is not normative. A transmitted schema definition shall not change during binary description stream lifetime and there shall not be two schema identifiers associated to the same namespace. Finally, all the optimised decoders associated to existing types are immediately applied to all types they derive from in accordance to the rules defined for the optimized decoders in Clause 9.

Once received by the decoder, a schema update unit immediately updates schema information managed by the decoder. It becomes available for the fragment update unit carried in the same access unit as well as future access units.

In subclause 5.4, replace this paragraph:

An access unit is composed of any number of fragment update units, each of which is extracted in sequence by the fragment update component extractor. Each fragment update unit consists of:

by

In case of TeM, an access unit is composed of any number of fragment update units.

In both TeM and BiM, fragment update units are extracted in sequence by the fragment update component extractor. Each fragment update unit consists of:

In subclause 5.4, replace the following paragraph:

— a fragment update payload conveying the coded description fragment to be added or replaced.

by

— a fragment update payload conveying either the coded description fragment (extracted out of the original description) to be added or replaced, or a reference to it.

In subclause 5.4, replace the following text:

The corresponding update command and context are processed by the non-normative description composer, which either places the description fragment received from the fragment update payload decoder at the appropriate node of the current description tree at composition time, or sends a reconstruction event containing this information to the application. The actual reconstruction of the current description tree by the description composer is implementation-specific, i.e., the application may direct the description composer to prune or ignore unwanted elements as desired. There is no requirement on the format of this current description tree, e.g. it may remain a binary representation.

by

The corresponding update command and context are processed by the non-normative description composer, which either places the description fragment extracted out of the original description or a reference to it received from the fragment update payload decoder at the appropriate node of the current description tree, or sends a reconstruction event containing this information to the application. If the payload consists of a fragment reference, depending on its nature, the referenced fragment is either immediately acquired (non-deferred fragment reference) or its acquisition is left to the application (deferred fragment references). In case of a deferred fragment reference, a fragment reference marker is available to the application to help further acquisition. This marker consists of the fragment reference itself, the name and type of the top most element of the referenced fragment. The fragment reference marker is added to the current description tree at the location defined by the fragment update context.

The actual reconstruction of the current description tree by the description composer is implementation-specific, i.e., the application may direct the description composer to prune or ignore unwanted elements as desired. There is no requirement on the format of this current description tree, e.g. it may remain a binary representation.

Change the title of subclause 5.5.2 by:

Deferred nodes, fragment references and their use

In subclause 5.5.2, add the following sentence at the end of the first paragraph:

Some deferred nodes are marked with a fragment reference marker that specifies where the fragment can be acquired. It is then left to the application to decide when to acquire it.

In subclause 5.5.2, replace the following sentence:

… The deferred nodes may then be replaced in any subsequent access unit without changing the tree topology maintained internally in the decoder. …

by

… The deferred nodes may then be replaced in any subsequent access unit or on application demand without changing the tree topology maintained internally in the decoder. …

In subclause 5.5.3, remove “ISO/IEC 15938” of the second sentence of the first paragraph.

In subclause 5.5.3, add the following note at the end of the subclause:

Note – Forward compatibility can also be used to generate bitstreams that can be decoded even in case of a schema update unit has not been received (for example because an error occurred) or because the decoder is not able to accept schema update units.

In subclause 5.6.3, replace the following text:

In the TeM, the commands are AddNode, ReplaceNode, and DeleteNode. The AddNode is effectively an “append” command, adding an element of the target node. Insertion between two already-received, consecutive children of a node is not possible. One must replace a previously deferred node. By performing a DeleteNode on a node on the current description tree, the addressable indices of its siblings change appropriately.

by

In the TeM, the commands are AddNode, ReplaceNode, and DeleteNode. The AddNode is by default an “append” command, adding an element after the last child of the target node. The position of the added element can also be explicitely defined allowing an element for instance to be inserted before the first element or between two other elements. By performing a DeleteNode on a node on the current description tree, the addressable indices of its siblings change appropriately.

In subclause 5.6.3, replace the following text:

In the BiM, the commands are AddContent, ReplaceContent and DeleteContent. The AddContent conveys the node data for a node whose path within the description tree is predetermined from the schema evaluation as described in . Hence, internally to the BiM decoder, the paths to (or addresses of) non-empty sibling nodes may be non-contiguous, e.g., the second and fourth occurrence of an element may be present. The “hole” in the numbering is not visible in the current description tree generated by the description composer. Hence, if the third occurrence of said element is added (using AddContent) in a subsequent access unit, it appears to any further processing steps as an “inserted” element in the current description tree, while it simply fills the existing “hole” with respect to the internal numbering of the BiM decoder.

by

In the BiM, the commands are AddContent, ReplaceContent and DeleteContent. The AddContent conveys the node data for a node whose path within the description tree is predetermined from the schema evaluation as described in 5.6.2. The insertion point for the node data is indicated using so called position codes. BiM supports 2 mechanisms for representing these position codes: integer or rational numbers. The first method (integer numbers) requires that "holes" must be left to enable the insertion of new node data. Thus documents to be sent must be known in advance or some contingency holes must be allocated. Hence, if the third occurrence of said element is added (using AddContent) in a subsequent access unit, it appears to any further processing steps as an “inserted” element in the current description tree, while it simply fills the existing “hole” with respect to the internal numbering of the BiM decoder. The “hole” in the numbering is not visible in the current description tree generated by the description composer. However, it is not always possible to know in advance the number of nodes to be added and where within the description tree they are to be added. The second method (rational numbers) removes this limitation by enabling the insertion of new node data at any location within the current binary description tree. A single method shall be used for representing position codes within a given stream. This is signalled within the decoderInit.

In subclause 5.6.4, replace the following text:

Wildcards and mixed content models (defined in ISO/IEC 15938-2) are not supported at all by the BiM. Therefore a schema that uses these mechanisms cannot be supported by the binary format.

by

Wildcards (defined in ISO/IEC 15938-2) are supported by the BiM, only in the case the schema of the elements or attributes validated by the wildcard is known by the decoder (i.e. it is in the initial schema or in one of the additional schemas acquired by the decoder).

The BiM encodes only the canonical form of XML documents. Therefore comments as well as namespace prefixes are lost in the encoding process.

Add the following subclause (subclause 5.8):

5.8 Decoding of Fragment References

5.8.1 Decoding of Non-Deferred Fragment References

The result of decoding a non-deferred fragment reference shall be, passed to a mechanism (fragment reference resolver) which returns fragment update payload data to the FU payload decoder. This fragment update payload data may be in one of two forms:

— a TeM fragment update payload containing the description fragment data in case of a TeM bitstream;

— a BiM fragment update payload containing the description fragment data in case of a BiM bitstream.

Note - Examples of possible fragment resolver are:
- An HTTP communication session to a WEB server
- A DSM-CC Object carousel

5.8.2 Decoding of Deferred Fragment References

The result of decoding a deferred fragment reference shall be a fragment reference marker which consists of a fragment reference, the name and type of its top most element.

Note - This fragment reference marker is signalled to the application and can be used to acquire the fragment through the fragment reference resolver at any instant of the description stream.

In subclause 6.1, replace the following text:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
       xmlns:mpeg7s="urn:mpeg:mpeg7:systems:2001"
       targetNamespace="urn:mpeg:mpeg7:systems:2001"
       elementFormDefault="qualified"
       attributeFormDefault="unqualified">

       <!-- here clause 6 schema definition -->

</schema>
```

by

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"

       xmlns:mpeg7s="urn:mpeg:mpeg7:systems:amd1:2004"
       targetNamespace="urn:mpeg:mpeg7:systems:amd1:2004"
       elementFormDefault="qualified"
       attributeFormDefault="unqualified">

       <!-- here clause 6 schema definition -->

</schema>
```

In subclause 6.5.2, replace the add command definition by:

addNode	Adds the node conveyed within the `FUPayload` to the context node or the parent node of the context node according to the value of the position attribute of the FUContext. If the position is set to the value “lastChild”, the node conveyed within the FUPayload shall be added to the context node as the last child of the context node, or if the position attribute is set to the value “prevSibling”, the node conveyed within the FUPayload shall be added to the parent node of the context node as the previous sibling of the context node.

In subclause 6.6.1, rename the “`FUContextType`” in the DDL definition to “`FragmentUpdateContextTypeBase`”.

In subclause 6.6.1, add the following DDL definition:

```
<complexType name="FragmentUpdateContextType">
   <simpleContent>
      <extension base="mpeg7s:FragmentUpdateContextTypeBase">
         <attribute name="position" default="lastChild">
            <simpleType>
               <restriction base="string">
                  <enumeration value="lastChild"/>
                  <enumeration value="prevSibling"/>
               </restriction>
            </simpleType>
         </attribute>
      </extension>
   </simpleContent>
</complexType>
```

In subclause 6.6.2, add the following semantics:

position	This attribute shall be only present when the FUCommand takes the value “addNode”, and indicates the position of the node added by the “addNode” command against the context node specified by the navigation path. This attribute can have the following values: — “lastChild” : the last child of the context node. — “prevSibling” : the previous sibling of the context node.

In subclause 6.7.1, replace the following text:

```
<complexType name="FragmentUpdatePayloadType">
     <sequence>
        <any processContents="skip" minOccurs="0"/>
     </sequence>

     <attribute name="hasDeferredNodes" type="boolean"
                use="required" default="false"/>
     <anyAttribute namespace="##other" processContents="skip" use="optional"/>
</complexType>
```

by

```
<complexType name="FragmentUpdatePayloadType">
     <sequence>
       <any namespace="##other" processContents="skip" minOccurs="0"/>
       <element name="FragmentReference" type="mpeg7s:FragmentReferenceType"
                    minOccurs="0" />
     </sequence>

     <attribute name="hasDeferredNodes" type="boolean"
                  use="required" default="false"/>
     <anyAttribute namespace="##other" processContents="skip" use="optional"/>
</complexType>
```

In subclause 6.7.2, add the following semantics:

FragmentReference	Defines that the current fragment update payload carries a fragment reference instead of a complete fragment. This element shall not be present if fragment payload actually contains a fragment. The element name and type of the top most element of the fragment being referenced shall be carried before the fragment reference itself.

Add the following subclause (subclause 6.8):

6.8 Textual Fragment Reference

6.8.1 Syntax

```
<!-- ################################################ -->
<!--  Definition of FragmentReferenceType          -->
<!-- ################################################ -->

<complexType name="FragmentReferenceType" abstract="true">
   <attribute name="isDeferred" type="boolean" use="optional" default="false"/>
</complexType>

<complexType name="URIFragmentReferenceType" >
   <complexContent>
      <extension base="mpeg7s:FragmentReferenceType">
         <attribute name="href" type="anyURI" use="required" />
      </extension>
   <complexContent>
</complexType>
```

6.8.2 Semantics

The `FragmentReferenceType` is an abstract complex type which serves as a base type for specific implementation of a fragment reference. The `URIFragmentReferenceType` is a concrete complexType which defines fragment references as a URI.

Name	*Definition*
isDeferred	Defines the deferred nature of the fragment reference: — If the `isDeferred` attribute has a value of true, the fragment reference is deferred and shall be resolved as defined in subclause 5.8.2. — If the `isDeferred` attribute has value of false, the fragment reference is non-deferred and shall be resolved as defined in subclause 5.8.1.
href	Defines the URI of a fragment refererence of type `URIFragmentReferenceType`.

6.8.3 Examples

In the following, example of the instances of the FUPayload datatype using the fragment reference is shown:

```
<FUPayload>
   <mpeg7:VisualDescriptor xsi:type="mpeg7:ScalableColorType"/>
   <FragmentReference xsi:type="mpeg7s:URIFragmentReferenceType"
      isDeferred="true"
      href="http://aaa.bbb/ccc.xml"/>
</FUPayload>
```

In subclause 7.1, add the following text and figure at the end of the subclause:

Identifying schema components in the BiM framework

As described in Clause 5, BiM relies upon schema knowledge. In this specification, schema components (elements, types and attributes) are identified by both a *schema identifier* and a *component identifier*.

The decoder manages both a unique initial schema and several additional schemas. From the decoder point of view, both initial schemas and additional schemas are indentified through a unique table in which each entry identifies a specific schema: the first ‘`NumberOfSchemas`’ entries identify schemas that are part of the initial schema. The following ones identify additional schemas (starting at the ‘`NumberOfSchemas`’ entry and ending at the ‘`NumberOfSchemas + NumberOfAdditionalSchemas - 1`’).

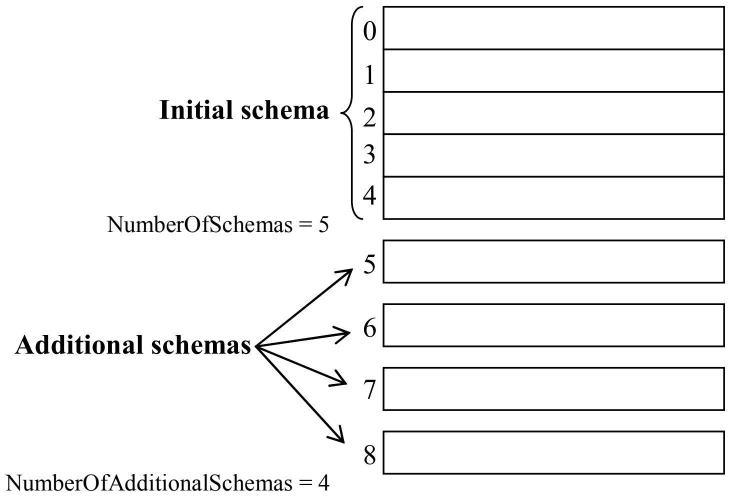


Figure AMD1-2 - Addressing the initial schema and the additional schemas

The schema component codes (type codes, element codes or attribute codes) are accessible through all these schemas. However codes are constructed differently depending on which schema they are defined. The initial schema aggregates all schema components possibly coming from different namespaces in a single code space. On the contrary, additional schemas contains only schema components which are defined in their namespace.

In subclause 7.2.1, replace the following text:

An optimised type codec specifies an encoding for a data type not using the generic mechanism specified in Clause 8 but optimised with full knowledge of the properties of that data type. Some optimised type codecs are specified in Part 3 of this specification.

by

An optimised type codec specifies an encoding for a data type not using the generic mechanism specified in Clause 8 but optimised with full knowledge of the properties of that data type. There are two kinds of optimised type codec (or optimised decoders). A fixed optimised decoder associates a specific encoding scheme to a type of the schema (complex as well as simple) and this association is fixed for the entire stream. An advanced optimised decoder associates a specific encoding scheme to any simple type and this association can be changed during the transmission of the bitstream. Moreover, several advanced optimised decoders can be associated to a single type and can accept parameters. Some fixed optimised type codecs are specified in Part 3 of the specification. Some advanced optimised decoders are defined in Clause 9.

In subclause 7.2.1, add the following text at the end of the subclause:

Several other coding modes are initialised in the `DecoderInit` related to the features used by the binary description stream: the insertion of elements, the transmission of schema information and references to fragments.

Transmission of additional schema is specified for two different use cases: The retrieval of schema information in binary format from a location indicated by a URI, the transmission of schema information in a binary description stream jointly or not with the transmission of a description. In the latter case there is a requirement that all schema information needed for the decoding of a fragment of the transmitted description must have been received before such fragment arrives.

Replace subclause 7.2.2 content by:

DecoderInit () {	**Number of bits**	**Mnemonic**
SystemsProfileLevelIndication	8+	vluimsbf8
UnitSizeCode	3	bslbf
NoAdvancedFeatures	1	bslbf
ReservedBits	4	bslbf
If (! NoAdvancedFeatures) {		
AdvancedFeatureFlags_Length	8+	vluimsbf8
/** FeatureFlags **/		
InsertFlag	1	bslbf
AdvancedOptimisedDecodersFlag	1	bslbf
AdditionalSchemaFlag	1	bslbf

AdditionalSchemaUpdatesOnlyFlag	1	bslbf
FragmentReferenceFlag	1	bslbf
MPCOnlyFlag	1	bslbf
HierarchyBasedSubstitutionCodingFlag	1	bslbf
ReservedBitsZero	FeatureFlags_Length*8-6	bslbf
/** FeatureFlags end **/		
}		
*/** Start FUUConfig **/*		
If(! AdditionalSchemaUpdatesOnlyFlag) {		
NumberOfSchemas	8+	vluimsbf8
for (k=0; k< NumberOfSchemas; k++) {		
SchemaURI_Length[k]	8+	vluimsbf8
SchemaURI[k]	8* SchemaURI_Length[k]	bslbf
LocationHint_Length[k]	8+	vluimsbf8
LocationHint[k]	8* LocationHint_Length[k]	bslbf
NumberOfTypeCodecs[k]	8+	vluimsbf8
for (i=0; i< NumberOfTypeCodecs[k]; i++) {		
TypeCodecURI_Length[k][i]	8+	vluimsbf8
TypeCodecURI[k][i]	8* TypeCodecURI_Length[k][i]	bslbf
NumberOfTypes[k][i]	8+	vluimsbf8
for (j=0; j< NumberOfTypes[k][i]; j++) {		
TypeIdentificationCode[k][i][j]	8+	vluimsbf8
}		
}		
}		
*/** FUUConfig - Advanced optimised decoder framework **/*		
If (AdvancedOptimisedDecodersFlag) {		
NumOfAdvancedOptimisedDecoderTypes	8+	vluimsbf8
for (i=0; i< NumOfAdvancedOptimisedDecoderTypes; i++) {		
AdvancedOptimisedDecoderTypeURI_Length[i]	8+	vluimsbf8
AdvancedOptimisedDecoderTypeURI[i]	8* AdvancedOptimisedDecoderTypeURI_Length[i]	bslbf
}		
AdvancedOptimisedDecodersConfig ()		
}		

*<u>/** FUUConfig - Fragment reference framework **/</u>*		
If (FragmentReferenceFlag) {		
NumOfSupportedFragmentReferenceFormat	8	uimsbf
for (i=0;i< NumOfSupportedFragmentReferenceFormat;i++) {		
SupportedFragmentReferenceFormat[i]	8	blsbf
}		
}		
}		
*/** <u>end FUUConfig **/</u>*		
If (AdditionalSchemaFlag) {		
AdditionalSchemaConfig ()		
}		
*/** Initial description **/*		
If (!AdditionalSchemaUpdateOnlyFlag) {		
InitialDescription_Length	8+	vluimsbf8
InitialDescription()		
}		
}		

In subclause 7.2.3, add the following semantics after the sematics of the `UnitSizeCode` *syntax element:*

NoAdvancedFeatures	Signals that none of the following advanced features is used in the binary stream:
	— dynamic insertions of child elements in the binary current description tree;
	— advanced optimised decoders;
	— schema transmission;
	— fragment references;
	— position codes only based on MPC.

AdvancedFeatureFlags_Length	Defines the number of bytes used for the indication of the advanced features. Note – This length provides a simple framework for future extensions.
InsertFlag	Signals that the insertion of child elements in the binary description tree at specific positions is performed by the use of rationale position code as described in subclause 7.6.5.5.
AdvancedOptimisedDecodersFlag	Signals that advanced optimised decoders are supported as described in Clause 9.
AdditionalSchemaFlag	Signals that additional schemas are supported.

AdditionalSchemaUpdatesOnlyFlag	Signals that the description stream contains only additional schema updates i.e. no fragment update units. The `AdditionalSchemaFlag` shall be set to true when this flag is set to true.
FragmentReferenceFlag	Signals that fragment references are supported.
MPCOnlyFlag	Signals that position codes in the fragment update context are encoded in MPC mode only.
HierarchyBasedSubstitutionCodingFlag	Signals that element substitution codes are computed taking into account their substitution hierachy. If additional schemas are supported (i.e. `AdditionalSchemaFlag`==true) this flag shall be set to true.

In subclause 7.2.3, replace the "NumberOfSchemas" semantics by:

NumberOfSchemas	Indicates the number of schemas on which the description stream is based. These schemas compose the initial schema. A zero-value is forbidden.

In subclause 7.2.3, add the following semantics after the sematics of the `TypeIdentificationCode` *syntax element*

NumOfAdvancedOptimisedDecoderTypes	Defines the number of advanced optimised decoder types that are necessary to properly decode the binary description stream.
AdvancedOptimisedDecoderTypeURI_Length[i]	Indicates the size in bytes of the `AdvancedOptimisedDecoderTypeURI[i]` syntax element.
AdvancedOptimisedDecoderTypeURI[i]	Defines the UTF-8 representation of the URI referencing the advanced optimised decoder type with index i.
AdvancedOptimisedDecodersConfig()	See subclause 9.2.
NumOfSupportedFragmentReferenceFormat	Specifies the number of fragment reference format that shall be supported by the decoder.
SupportedFragmentReferenceFormat[i]	Specifies the ith fragment reference format, according to Table AMD1-1, that shall be supported by the decoder. The `SupportedFragmentReferenceFormat[0]` indicates the default fragment reference format.
AdditionalSchemaConfig()	See subclause 7.2.4.

In subclause 7.2.3, add the following table:

Table AMD1-1 - Fragment Reference Formats

Fragment Reference Type	***Fragment Reference Format***	***Description***
0		ISO reserved
1	URIFragmentReference	This fragment reference format should be used where the reference can be expressed as a URI.
2 - 224		ISO reserved
225 – 255		Private Use

Add the following subclause (subclause 7.2.4):

7.2.4 Syntax of AdditionalSchemaConfig

AdditionalSchemaConfig () {	**Number of bits**	**Mnemonic**
NumberOfAdditionalSchemas	8+	vluimsbf8
NumberOfKnownAdditionalSchemas	8+	vluimsbf8
for (int t=0;t<NumberOfKnownAdditionalSchemas;t++){		
KnownAdditionalSchemaID	8+	
AdditionalSchemaURI_Length[KnownAdditionalSchemaID]	8+	vluimsbf8
AdditionalSchemaURI[KnownAdditionalSchemaID]	8* AdditionalSchemaURI_Length [KnownAdditionalSchemaID]	bslbf
BinaryLocationHint_Length[KnownAdditionalSchemaID]	8+	vluimsbf8
BinaryLocationHint[KnownAdditionalSchemaID]	8*BinaryLocationHint_Length[KnownAdditionalSchemaID]	bslbf
NumberOfTypeCodecs[KnownAdditionalSchemaID]	8+	vluimsbf8
for (i=0; i< NumberOfTypeCodecs[KnownAdditionalSchemaID]; i++) {		
TypeCodecURI_Length[KnownAdditionalSchemaID][i]	8+	vluimsbf8
TypeCodecURI[KnownAdditionalSchemaID][i]	8* TypeCodecURI _Length[KnownAdditionalSchemaID][i]	bslbf
NumberOfTypes[KnownAdditionalSchemaID][i]	8+	vluimsbf8
for (j=0; j< NumberOfTypes[KnownAdditionalSchemaID][i]; j++) {		
TypeIdentificationCode[KnownAdditionalSchemaID][i][j]	8+	vluimsbf8
}		
}		

}		
SchemaEncodingMethod	8	blsbf
ExternallyCastableTypeTable(InitialSchema)		
ExternallySubstitutableElementTable(InitialSchema)		
ReservedBitsZero	7	blsbf
}		

Add the following subclause (subclause 7.2.5):

7.2.5 Semantics of AdditionalSchemaConfig

Name	*Definition*
NumberOfAdditionalSchemas	Indicates the number of schemas that can be transmitted and that are not declared in the list of `schemaURI`. If additional schemas are not supported, this value is set to zero.
NumberOfKnownAdditionalSchemas	Indicates the number of additional schemas that are known to be updated in the bitstream.
KnownAdditionalSchemaID	Identifies a schema known to be updated in the bistream. This identifier shall only address an additional schema i.e. its value shall be superior to '`NumberOfSchemas-1`'
AdditionalSchemaURI_Length[KnownAdditionalSchemaID]	Indicates the size in bytes of the `AdditionalSchemaURI[KnownAdditionalSchemaID]` length. A value of zero is forbidden.
AdditionalSchemaURI[KnownAdditionalSchemaID]	Indicates the UTF-8 representation of the URI of the additional schema identified by `KnownAdditionalSchemaID`. Note – This field allows to identify some of the additional schemas that are expected to be updated. This information allows one decoder not to monitor the schema updates for which it already knows the schema.
BinaryLocationHint_Length[KnownAdditionalSchemaID]	Indicates the size in bytes of the `Binary LocationHint_Length[KnownAdditionalSchemaID]`. A value of zero indicates that for the schema that is referenced by the index `KnownAdditionalSchemaID` there is no binary encoded schema available.
BinaryLocationHint[KnownAdditionalSchemaID]	This is the UTF-8 representation of the URI to unambiguously reference the location of the binary encoded schema that is referenced by the index `KnownAdditionalSchemaID`. The schema can be fetched by the schema resolver and is then received as a description stream composed only of schema update units i.e. for which the `SchemaOnlyFlag` is set to true.
NumberOfTypeCodecs[KnownAdditionalSchemaID]	see `NumberOfTypeCodecs[k]` in 7.2.3.

TypeCodecURI_Length[KnownAdditionalSchemaID]	see `TypeCodecURI_Length[k]` in 7.2.3.
TypeCodecURI[KnownAdditionalSchemaID][i]	see `TypeCodecURI[k][i]` in 7.2.3.
NumberOfTypes[KnownAdditionalSchemaID][i]	see `NumberOfTypes[k][i]` in 7.2.3.
TypeIdentificationCode[KnownAdditionalSchemaID][i][j]	see `TypeIdentificationCode[k][i][j]` in 7.2.3.
SchemaEncodingMethod	Indicates the encoding method of the schema update units.
ExternalCastableTypeTable	Defines the types of the initial schema that are externally castable as defined in subclause 7.7.5.4 and 7.7.5.5.
ExternalSubstitutableElementTable	Defines the elements of the initial schema that are substitutable as defined in subclause 7.7.6.4 and 7.7.6.5.

Table AMD1-2 - Schema encoding method

SchemaEncodingMethod	***definition***
0	ISO reserved
1	BiM encoded schema as described in subclause 7.7.8
2-224	ISO reserved
225-255	Private use

In subclause 7.3.2, replace the `AccessUnit` *syntax by:*

AccessUnit () {	**Number of bits**	**Mnemonic**
If (AdditionalSchemaFlag) {		
NumberOfSUU	8+	vluimsbf8
for (i=0; i< NumberOfSUU ; i++) {		
SchemaUpdateUnit()		
}		
}		
If(! AdditionalSchemaUpdateOnlyFlag) {		
NumberOfFUU	8+	vluimsbf8
for (i=0; i< NumberOfFUU ; i++) {		
FragmentUpdateUnit()		
}		
}		
}		

In subclause 7.3.3, replace the semantics table by:

Name	*Definition*
NumberOfSUU	Indicates the number of schema update units in this access unit. Value '0' signifies that no schema update unit is carried.
NumberOfFUU	Indicates the number of fragment update units in this access unit. Value '0' signifies that no fragment update unit is carried.

In subclause 7.4.2, insert the following syntax elements, between the `FUU_Length` *and the* `FragmentUpdateCommand` *syntax elements:*

If (AdvancedOptimisedDecodersFlag){		
OptimisedDecoderReparameterization	2	bslbf
if (OptimisedDecoderReparameterization == '00') {		
AdvancedOptimisedDecodersConfig ()		
}		
}		

In subclause 7.4.3, insert the following semantics, between the `FUU_Length` *and the* `FragmentUpdateCommand` *semantics:*

OptimisedDecodersReparameterization	This 2-bit flag signals if the parameters of the optimised decoders shall be updated. It can take the following values: — '00' – the optimised decoder instance table and mappings shall be redefined; — '01' – the optimised decoder instance table and mappings shall not be redefined; — '10' – the optimised decoder instance table and mappings are reset to the default table and mappings defined in the `DecoderInit`; — '11' – reserved.
AdvancedOptimisedDecodersConfig()	See subclause 9.2.

In subclause 7.6.1, replace the following text:

There are two different TBC tables associated to each complexType: The ContextTBC table contains only references to the child elements of complexType and additionally one code word to signal the termination of the path (Path Termination Code). The ContextTBC table contains also one TBC to refer to the parent node. It allows relative navigation within the binary format description tree and move upwards to the parent node. The OperandTBC table additionally contains also the references to the attributes and either to the elements of simpleType or to a simple content, but does not contain the Path Termination Code nor the reference to the parent node. Furthermore, in the OperandTBC table one TBC is reserved for User Data Extension as defined in section 7.6.5.2. Example TBC tables are shown in Table 4 and Table 5.

by

There are two different TBC tables associated to each complexType: The ContextTBC table contains only references to the child elements of complexType and additionally one code word to signal the termination of the path (Path Termination Code). The ContextTBC table contains also one TBC to refer to the parent node. It allows relative navigation within the binary format description tree and move upwards to the parent node. The OperandTBC table additionally contains also the references to the attributes and either to the elements of simpleType or to a simple content, but does not contain the Path Termination Code nor the reference to the parent node. Furthermore, in the OperandTBC table one TBC is reserved for User Data Extension as defined in subclause 7.6.5.2. In case of a mixed content model the OperandTBC table also contains a reference to the character data that may appear between the elements. Example TBC tables are shown in Table 4 and Table 5.

Add the following sentence before the Table “Example of a Context TBC Table”:

In this example the content model of the complex type definition is not ‘mixed’.

In subclause 7.6.2, replace the `SchemaID` syntax element in the `FragmentUpdateContext` syntax table:

SchemaID	ceil(log2(`NumberOfSchemas` + `NumberOfAdditionalSchemas`))	uimsbf

In subclause 7.6.2, replace the `ContextPath` syntax table by:

ContextPath () {	**Number of bits**	**Mnemonic**
TBC_Counter = 0		
NumberOfFragmentPayloads = 1		
do {		
if ((ContextModeCode == ‘001’ || ContextModeCode == ‘011’) && TBC_Counter ==0) { */* absolute addressing mode and first TBC of the context path */*		
If (AdditionalSchemaFlag) {		
SchemaIDOfSBC_Context_Selector	ceil(log2(`NumberOfSchemas` + `NumberOfAdditionalSchemas`))	uimsbf
Extended_SBC_Context_Selector	ceil(log2(number_of_global elements in `SchemaIDOfSBC_Context_Selector`))	bslbf
} else {		
SBC_Context_Selector	ceil(log2(number of global elements +1))	bslbf
}		
PathTypeCode()		

}		
else {		
SBC_Context	ceil(log2(number of child elements of complexType + 2))	bslbf
If (SBC_Context == “SBC_any”) {		
AnyElementDecoding ()		
} else {		
SubstitutionCode()		
}		
PathTypeCode()		
}		
TBC_Counter ++		
} while ((SBC_Context_Selector != “Path Termination Code”) && (SBC_Context != “Path Termination Code”))		
if (SBC_Context_Selector == “Path Termination Code”)){		
If (AdditionalSchemaFlag) {		
SchemaIDOfSBC_Operand_Selector	ceil(log2(`NumberOfSchemas` + `NumberOfAdditionalSchemas`))	uimsbf
Extended_SBC_Operand_Selector	ceil(log2(number_of_global elements in `SchemaIDOfSBC_Operand_Selector`))	bslbf
} else {		
SBC_Operand_Selector	ceil(log2(number of global elements))	bslbf
}		
PathTypeCode()		
}		
else {		
SBC_Operand	ceil(log2(number of child elements + number of attributes + has_simpleContent + 1))	bslbf
if (SBC_Operand == “SBC_anyAttribute”) {		
SingleAnyAttributeDecoding()		
}		
if (SBC_Operand == “SBC_any”) {		
AnyElementDecoding()		
}		
SubstitutionCode()		
PathTypeCode()		
}		

TBC_Counter ++		
for (i=0; i < TBC_Counter; i++) {		
PositionCode()		
}		
if ((ContextModeCode == ‘011’) || (ContextModeCode == ‘100’)) { /* multiple fragment update payload mode*/		
do {		
IncrementalPositionCode	ceil(log2(NumberOfMultiOccurrenceLayer+2))	bslbf
if (IncrementalPositionCode != “Skip_Indication“) {		
NumberOfFragmentPayloads++		
}		
else {		
IncrementalPositionCode */* indicating the skipped position */*	ceil (log2(NumberOfMultiOccurrenceLayer+2))	bslbf
}		
} while (IncrementalPositionCode != “IncrementalPositionCodeTermination”)		
NumberOfFragmentPayloads-- */* there is no fragment update payload corresponding to the IncrementalPositionCodeTermination */*		
}		
}		

In subclause 7.6.3, in the first table, replace the `SchemaID` *semantics by:*

SchemaID	Identifies the schema (from the list of `schemaURI`s transmitted in the `DecoderInit` (optionally extended by a list of additional schemas) which is used as basis for the fragment update context coding. The `SchemaID` code word is built by sequentially addressing the list of `SchemaURI` contained in the `DecoderInit` (optionally followed by the additional schemas). The length of this field is determined by: “ceil(log2(`NumberOfSchemas`))” or “ceil(log2(`NumberOfSchemas + NumberOfAdditionalSchemas`))” depending on the presence of additional schemas. The value of this code word is the same as the variable "k" in the definition of the `SchemaURI[k]` syntax element as specified in 7.2.3 optionally extended to additional schemas. The `SchemaID` syntax element is also used for the decoding of the fragment update payload as described in subclause 8.4.4. If the `ContextModeCode` selects a relative addressing mode then the `SchemaID` shall have the same value as in the previous fragment update unit.

In subclause 7.6.3, in the second table, add the following semantics after the `TBC_Counter` semantic:

SchemaIDOfSBC_Context_Selector	Identifies the schema in which the `Extended_SBC_Context_Selector` selects a declared global element.
Extended_SBC_Context_Selector	Selects one global element of the schema referenced by `SchemaIDOfSBC_Context_Selector` using the ContextTBC table as specified in 7.6.5.2.3.

In subclause 7.6.3, in the second table, add the following semantic after the `SBC_Context` semantic:

AnyElementDecoding()	See 8.5.2.4.5.2 and 8.5.2.4.5.3.

In subclause 7.6.3, in the second table, add the following semantics after the `SubstitutionCode` semantic:

SchemaIDOfSBC_Operand_Selector	Identifies the schema in which the `Extended_SBC_Operand_Selector` selects a declared global element.
Extended_SBC_Operand_Selector	Selects one global element of the schema referenced by `SchemaIDOfSBC_Operand_Selector` using the ContextTBC table as specified in 7.6.5.2.3.

In subclause 7.6.3, in the second table, add the following semantic after the `SBC_Operand` semantic:

SingleAnyAttributeDecoding()	See 8.5.3.3

In subclause 7.6.5.2.2, add the following bullet after the 6th bullet (i.e. "In the table for OperandTBCs the all-zero SBC…"):

— In the table for OperandTBCs the all-zero-and-one SBC (ex. 00001) is assigned to the character data in the mixed content of a datatype if the datatype has a mixed content model.

In subclause 7.6.5.2.2, replace the following text:

— All other Schema Branch Codes are assigned to the children nodes of the complexType. The children are defined as the attributes of the complex type as well as, either the contained elements or a dedicated child representing a simple content. If there are two or more element declarations with the same name in the complexType definition then each shall be assigned a different SBC.

by

— All other Schema Branch Codes are assigned to the children nodes of the complexType. The children are defined as the attributes of the complex type as well as, either the contained elements or a dedicated child representing a simple content. If there are two or more element declarations with the same name in the complexType definition then each shall be assigned a different SBC. If there is an "any" element declared in the complex type then a SBC is also assigned to this element and this SBC is called

"SBC_any". If there is an "anyAttribute" declaration in the complex type then a SBC is also assigned to it and this SBC is called "SBC_anyAttribute".

In subclause 7.6.5.2.2, replace the following text:

— The SBCs for child elements and simple content are assigned first, the SBCs for attributes are assigned last. The attributes are ordered lexicographically for the assignment of the Schema Branch Codes.

by

— The SBCs for child elements and simple content are assigned first, the SBCs for attributes are assigned last. The attributes are ordered lexicographically for the assignment of the Schema Branch Codes. The lexicographical ordering for an "any" element and for an "anyAttribute" is done with respect to their signature as defined in subclause 8.5.2.2.4

Replace the entire content of subclause 7.6.5.2.3 by:

For the special case of the selector node the following rules apply:

If the `AdditionalSchemaFlag` in the binary `DecoderInit` equals '0' then

— The length in bits of these SBCs is determined by the number of global elements declared in the schema referred by the `SchemaID` as follows:

— SBC_Context_Selector: ceil(log2(number of global elements + 1)).

— SBC_Operand_Selector: ceil(log2(number of global elements)).

— The SBCs are assigned sequentially to the global elements defined in the schema referred by the `SchemaID`. Before the assignment:

— in case the `HierarchyBasedSubstitutionCodingFlag` is set to false, a lexicographical ordering of all global elements is performed.

— in case the `HierarchyBasedSubstitutionCodingFlag` is set to true, a depth first ordering is performed with respect to the hierarchy of element substitutions which forms one or several trees as shown in Figure AMD1-3. For elements which are siblings within the element substitution hierarchy or roots of a substitution hierarchy a lexicographical ordering is performed based on their expanded name as defined in 8.2.

— No codes are assigned for a reference to the parent node nor for the User Data Extension Code. The Path Termination Code, however, is present in the ContextTBC table.

If the `AdditionalSchemaFlag` in the binary `DecoderInit` equals '1' then

— The length in bits of the `Extended_SBC_Context_Selector` respectively the `Extended_SBC_Operand_Selector` is determined by the number of global elements declared in the schema referred by the `SchemaIDOfSBC_Context_Selector` respectively `SchemaIDOfSBC_Operand_Selector` as follows:

— Extended_SBC_Context_Selector: ceil(log2(number of global elements + 1)).

— Extended_SBC_Operand_Selector: ceil(log2(number of global elements)).

- The SBCs are assigned sequentially to the global elements defined in the schema referred by the `SchemaIDOfSBC_Context_Selector` respectively `SchemaIDOfSBC_Operand_Selector`. No SBCs are assigned to elements imported from other namespaces into this schema. Before the assignment:
  - in case the `HierarchyBasedSubstitutionCodingFlag` is set to false, a lexicographical ordering of all global elements is performed.
  - in case the `HierarchyBasedSubstitutionCodingFlag` is set to true, a depth first ordering is performed with respect to the hierarchy of element substitutions which forms one or several trees as shown in Figure AMD1-3. For elements which are siblings within the element substitution hierarchy or roots of a substitution hierarchy a lexicographical ordering is performed based on their expanded name as defined in 8.2.
- No codes are assigned for a reference to the parent node nor for the User Data Extension Code. The Path Termination Code, however, is present in the ContextTBC table.

In subclause 7.6.5.3.1, add the following sentence at the end of the subclause:

The `GlobalSubstitutionSelect` is used when the substitute element is defined in a other schema than the expected element. In that case, the `GlobalSubstitutionSelect` selects the substitute element from the set of all elements defined in the schema referenced by the `SchemaID`.

In subclause 7.6.5.3.2, replace the `SubstitutionCode` *syntax table by:*

SubstitutionCode () {	**Number of bits**	**Mnemonic**
if (substitution_possible == 1 || external_element_substitution_possible == 1 || all_element_externally_substitutable == 1) {		
SubstitutionFlag	1	bslbf
if (SubstitutionFlag == 1) {		
if (external_element_substitution_possible == 1 || all_element_externally_substitutable == 1) {		
SchemaSwitching	1	bslbf
if (SchemaSwitching) {		
SchemaID	ceil(log2(`NumberOfSchemas` + `NumberOfAdditionalSchemas`))	uimsbf
GlobalSubstitutionSelect	ceil(log2 (number_of_global_elements_in_schema_referred_by_SchemaID))	bslbf
} else {		
SubstitutionSelect	ceil(log2(number_of_possible_substitutes))	bslbf
}		

} else {		
SubstitutionSelect	ceil(log2(number_of_possible_substitutes))	bslbf
}		
}		
}		
}		

In subclause 7.6.5.3.3, add the following semantics after the `substitution_possible` *semantic:*

external_element_substitution_possible	This internal flag indicates whether the element can be subject to a substitution occurring in an other schema than the one in which the element is defined. This flag is set by the `ExternallySubstitutableType` table defined in subclause 7.7.6.4 and 7.7.6.5.
all_element_externally_substitutable	This internal flag indicates whether every element defined in the schema of the expected element can be subject to a substitution occurring in an other schema. This flag is set by the `ExternallySubstitutableType` table defined in subclause 7.7.6.4 and 7.7.6.5.

In subclause 7.6.5.3.3 add the following semantics after the `SubstitutionFlag` *semantic:*

SchemaSwitching	Indicates whether the element substitution occurs in an other schema than the schema where the expected element is defined.
SchemaID	Identifies the schema in which the substitute element is defined.
GlobalSubstitutionSelect	This code identifies the substitute element in the schema of index '`SchemaID`' in the `SchemaURI[k]` table. When the `HierarchyBasedSubstitutionCondingFlag` is set to false or when it is not defined in the `DecoderInit`, the code referring to the elements are assigned sequentially starting from zero after lexicographical ordering of all global elements using their expanded names as defined in subclause 8.2. When the `HierarchyBasedSubstitutionCodingFlag` is set to true, the `SubstitutionSelect` codes are assigned in a depth-first manner with respect to the hierarchy of element substitutions which forms on or several trees as shown in an example in Figure AMD1-3. For elements which are siblings within the element substitution hierarchy or elements which are roots of a substitution hierarchy the code words are assigned in a lexicographical order based on their expanded names. The order of elements that have a head of substitution in an other schema than the one identified by `SchemaID` are defined in the same relative order than if they were in the *initial schema*. The length of this field is determined by "ceil(log2(number_of_global elements in the schema identified by `SchemaID`))" in both cases. Note – If schema identified by `SchemaID` is an additional schema, the substitution select codes are computed on the set of all elements defined in the namespace identified by the `SchemaID` entry in the `SchemaURI` table of the `DecoderInit`.

In subclause 7.6.5.3.3, replace the semantic of `SubstitutionSelect` *by:*

SubstitutionSelect	This code is used as address within a substitution group where each element defined in the schema of the expected element is assigned a `SubstitutionSelect` code. When the `HierarchyBasedSubstitutionCondingFlag` is set to false or when it is not defined in the `DecoderInit`, the `SubstitutionSelect` codes referring to the elements are assigned sequentially starting from zero after lexicographical ordering of the element using their expanded names as defined in subclause 8.2. The length of this field is determined by "ceil(log2(number_of_possible_substitutes in the schema of the expected element))". In case the `HierarchyBasedSubstitutionCondingFlag` is true, the `SubstitutionSelect` codes are assigned in a depth-first manner with respect to the hierarchy of element substitutions which forms one or several trees as shown in an example in Figure AMD1-3. For elements which are siblings within the element substitution hierarchy or for elements which are roots of a substitution hierarchy the code words are assigned in a lexicographical order based on their expanded names. The element substitution code identifies the substitute element which is used in the encoded document. The length of the code word for the element substitution code is equal to "ceil(log2(number of possible_substitute in the schema of the expected element))". Note – If the schema identified by `SchemaID` is an additional schema, the substitution select codes are computed on the set of all elements defined in the namespace identified by the `SchemaID` entry in the `SchemaURI` table of the `DecoderInit`.

In subclause 7.6.5.3.3, add the following figure after the semantics table:

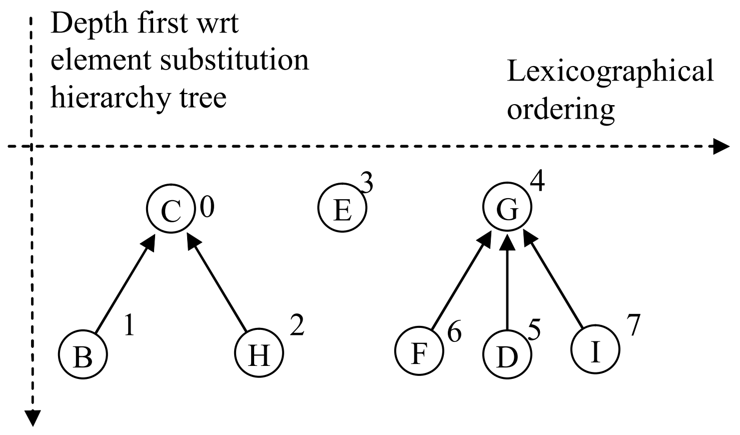


Figure AMD1-3 - Example for the Element Substitution Identification Code assignment for some elements in the hierarchy based coding mode

In subclause 7.6.5.4.1, add the following sentence at the end of the subclause:

The `GlobalTypeIdentificationCode` is used when the effective type is defined in an other schema than the expected type. In that case, the `GlobalTypeIdentificationCode` selects the effective type from the set of all types defined in the schema referenced by the `SchemaID`.

In subclause 7.6.5.4.2, replace the `PathTypeCode` *syntax table by:*

PathTypeCode () {	**Number of bits**	**Mnemonic**
if (type_cast_possible == 1 || external_type_cast_possible == 1 || all_type_externally_castable == 1) {		
TypeCodeFlag	1	bslbf
if ((TypeCodeFlag == 1) {		
if (external_type_cast_possible == 1 || all_type_externally_castable == 1) {		
SchemaSwitching	1	bslbf
if (SchemaSwitching) {		
SchemaID	ceil(log2(`NumberOfSchemas` + `NumberOfAdditionalSchemas`))	uimsbf
GlobalTypeIdentificationCode	ceil(log2 (number_of_global_types_in_schema_referred_by_SchemaID))	bslbf
} else {		
TypeIdentificationCode	ceil(log2(number of derived types))	bslbf
}		
} else {		
TypeIdentificationCode	ceil(log2(number of derived types))	bslbf
}		
}		
}		
}		

In subclause 7.6.5.4.3 add the following semantics after the `type_cast_possible` *semantic:*

external_type_cast_possible	Indicates whether the expected type can be subject to a type casting occurring in an other schema than the one in which the type is defined. This flag is set by the `ExternallyCastableType` table defined in subclause 7.7.5.4 and 7.7.5.5.
all_type_externally_castable	Indicates whether every type defined in the schema of the expected type can be subject to a type casting occurring in an other schema. This flag is set by the `ExternallyCastableType` table defined in subclause 7.7.5.4 and 7.7.5.5.

In subclause 7.6.5.4.3 add the following semantics after the `TypeCodeFlag` *semantic:*

SchemaSwitching	Indicates whether the type cast occurs in an other schema than the schema of the expected type.
SchemaID	Identifies the schema in which the derived type element is addressed.
GlobalTypeIdentificationCode	Identifies a type defined in `SchemaIDOfDerivation` by a code word. The Type Identification Code is generated for a given type (simpleType or complexType) from the set of all types (itself being not included) including abstract types defined in the schema referenced by `SchemaID`. The Type Identification Codes are assigned in a depth-first manner with respect to the hierarchy of types which forms a tree as shown in an example in Figure 9. For types which are siblings within the type hierarchy the code words are assigned in a lexicographical order based on their expanded names. The Type Identification Code identifies the derived type which is used for the type cast. The length of the code word for the Type Identification Code is equal to “ceil(log2(number of types in the schema))”. The order of types that have a super type in an other schema than the one identified by `SchemaID` are defined in the same relative order than if they were in the initial schema. Note – If schema identified by `SchemaID` is an additional schema, the type codes are computed on the set of all types defined in the namespace identified by the `SchemaID` entry in the `SchemaURI` table of the `DecoderInit`.

In subclause 7.6.5.5.1, replace the following text:

Within a TBC a Position Code shall uniquely identify the position of a node among its sibling nodes in the binary description tree. It is present only if multiple occurrences are possible for the element referenced by the SBC or for any model group declared in the corresponding complexType definition. Position Codes are distinguished in Multiple element Position Codes (MPC) and Single element Position Codes (SPC) for efficiency reasons. The presence of the Position Code and the decision whether SPC or MPC are used is determined by the complexType definition.

by

Within a TBC a Position Code shall uniquely identify the position of a node among its sibling nodes in the binary description tree. Position Codes are distinguished in Multiple element Position Codes (MPC) and Single element Position Codes (SPC) for efficiency reasons. If the `MPCOnlyFlag` in the `DecoderInit` is set to true, MPC are always used. the `MPCOnlyFlag` in the `DecoderInit` is set to false, the presence of the Position Code and the decision whether SPC or MPC are used is determined by the complexType definition. In the second case, a position code is present only if multiple occurrences are possible for the element referenced by the SBC or for any model group declared in the corresponding complexType definition. Also the OperandTBC of character content in a mixed content model contains a position code.

In subclause 7.6.5.5.1, add the following text and figures after the note:

If the `InsertFlag` in the Binary `DecoderInit` is set to true then the Position Codes represent rational numbers (Rational Position Codes). Otherwise Position Codes represent integer numbers. In both cases the child elements are sorted in increasing order of these values.

Rational Position Codes are used to allow the insertion of child elements at any specific possible position in the binary description tree. Position Codes representing rational numbers are specified by the following rules:

— Rational Position Codes represent rational numbers in the interval]0<n<1[.

— Rational Position Codes are encoded in the *vlurmsbf5* format.

In Figure AMD1-4 an example of a binary description tree of the element A including an assignment of position codes to child elements B is given. The Position Codes representing rational numbers specify the order in which the child elements B reside in the binary description tree.

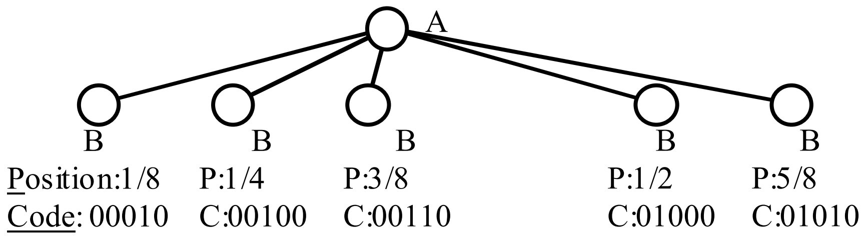


Figure AMD1-4 - Assignment of Position Codes to a set of child elements

When a new element B is inserted at any position then a new Position Code representing rational numbers is used so that the correct ordering in the binary description tree is unambiguously specified (see Figure AMD1-5).

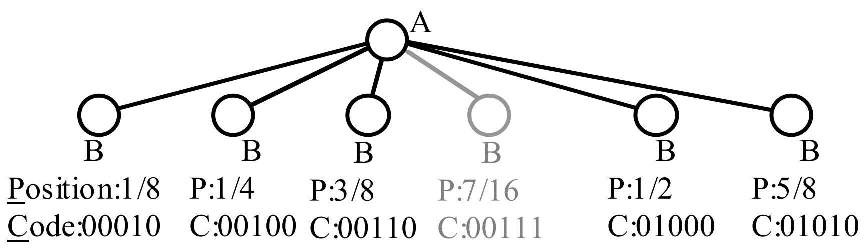


Figure AMD1-5 - Position Code of an inserted child element B (grey node)

In subclause 7.6.5.5.2, replace the following text:

A SPC is used, if a Position Code is present according to 7.6.5.5.1and if the corresponding complexType does not contain model groups with maxOccurs > 1. The SPC is only present if the SBC addresses an element with maxOccurs > 1. The SPC indicates the position of the node among the nodes addressed by the same SBC.

by

A SPC is used, if a Position Code is present according to 7.6.5.5.1, if the MPCOnlyFlag is set to false, and if the corresponding complexType does not contain model groups with maxOccurs > 1. The SPC is only present if the SBC addresses an element with maxOccurs > 1. The SPC indicates the position of the node among the nodes addressed by the same SBC.

In subclause 7.6.5.5.2, replace the following text:

The position is represented as integer value. The length in bits of the SPC is the equal to “ceil(log2(maxOccurs of the element addressed by the SBC))”. If this length exceeds 4 bits then vluimsbf5 is used for coding the SPC.

by

If, according to 7.6.5.5.1, the position is represented as rational number, the value is encoded as vlurmsbf5 .The value 0 shall be omitted.

If, according to 7.6.5.5.1, the position is represented as integer value. The length in bits of the SPC is the equal to “ceil(log2(maxOccurs of the element addressed by the SBC))”. If this length exceeds 4 bits then vluimsbf5 is used for coding the SPC.

In subclause 7.6.5.5.2, add the following text at the end of the subclause:

In the case of a complex type for which SPCs are used and that has mixed content the OperandTBC assigned to the character data in the mixed content also has a SPC. The position code of this OperandTBC is encoded assuming a maxOccurrence=MPA+1 since before and after each instantiated element character data can be present. The value MPA is specified in subclause 7.6.5.5.3.

Add the following text at the beginning of subclause 7.6.5.5.3:

If according to 7.6.5.5.1 the position is represented as rational number then the value is encoded as vlurmsbf5 and the value 0 shall be omitted.

If according to 7.6.5.5.1 the position is represented as integer number then the length in bits of the MPC is determined by the following method, which uses the ‘max occurs’ property of the effective content particles of the type definition.

Add the following text after the first paragraph:

In the case of a complex type that has mixed content model the OperandTBC assigned to the character data also uses a MPC.

Add the following text after the “For an element declaration particle” bullet:

In the case of a mixed content model the MPA_{mixed}=2MPA+1 since before and after each instantiated element character data can be present.

Replace subclause 7.6.5.5.4 by the following text and figure:

If an instantiated element was conveyed as part of a fragment update payload then the corresponding node has not been explicitly assigned a position in the binary format description tree. In this case, the following implicit positions are assigned to each added node for which a position code is expected in the TBC addressing this node:

— If Position Codes represent integer numbers:

  — in the case a MPC is expected: a position is assigned incrementally (starting from zero) to the added elements.

  — in the case a SPC is expected: a position is assigned incrementally (starting from zero) to the added elements corresponding to the same SBC.

— If Position Codes represent rational numbers: to Z consecutive elements the positions represented by rational numbers are assigned by the following steps. In the case a MPC is expected: Z is the number of all elements which have the same parent node. In the case a SPC is expected: Z is the number of all elements which have the same parent node and which correspond to the same SBC.

  — In this steps, P(i) denotes the i-th assigned position.

```
Step1:  Determine Z.

    Calculate N=2^{ceil(log2(Z+1))},

    Set i=0,P=0.

Step2:  while(i≤(3Z-N+3)/2 | mod_2(i)==0) {P(i)=P+=1/(N); i++;}.
```

Step3: while (i<Z) {P(i)=P+=2/N; i++;}. End.In the implicit assignment of rational position codes, the step 2 performs an ascending-oriented assignment and the step 3 performs an balance-oriented assignment. The ascending-oriented assignment is efficient in case of appending subsequent fragments context paths, whereas the balance-oriented assignment is efficient in case of inserting/replacing subsequent fragments context paths. The condition of step 2 controls the ratio between such ascending and balance-oriented assignment. Figure AMD1-6 shows an example of the implicit assignment of positions represented by rational number.

Step1: Z=10, N=16

	P(0)	P(1)	P(2)	P(3)	P(4)	P(5)	P(6)	P(7)	P(8)	P(9)
Step2:	1/16	1/8	3/16	1/4	5/16	3/8	7/16	1/2		
Step3:									5/8	3/4

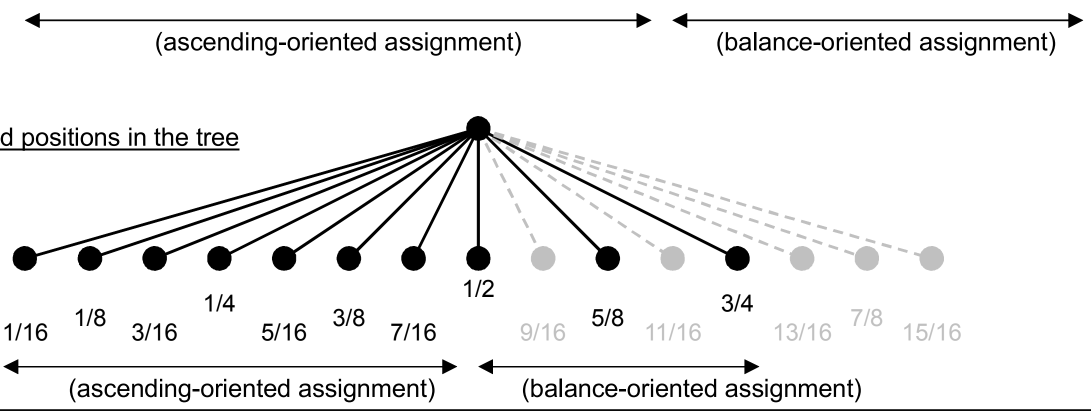

(ascending-oriented assignment)
(balance-oriented assignment)
assigned positions in the tree
1/16
1/8
3/16
1/4
5/16
3/8
7/16
1/2
9/16
5/8
11/16
3/4
13/16
7/8
15/16
(ascending-oriented assignment)
(balance-oriented assignment)

Figure AMD1-6 - an example on implicit assignment of positions represented by rational numbers

In subclause 7.6.5.6, replace the first paragraph by:

A fragment update unit can contain multiple fragment update payloads of the same type if the context paths of those fragment update payloads are identical except for their position codes. If position codes represent integer numbers, the position codes for the first fragment update payload are coded in the same way as in the case of a single payload, while the position codes for the other fragment update payloads within this fragment update unit are indicated in the context path by "Incremental Position Codes" as shown in Figure 10.

In subclause 7.6.5.6, add the following paragraph before Figure 10:

In the case of rational position codes, the structure of the context path is the same as the case of integer position codes. Only the Position Codes and the Incremental Position codes differ. For the Context Path rational position codes are used. Also for the incremental position codes rational position increments are specified. The order of rational position increments are predefined (see Figure AMD1-7).

In subclause 7.6.5.6, replace the following text:

The set of multiple-occurrence nodes is indexed beginning from the starting node of the context path. An Incremental Position Code indicates the index of the multiple-occurrence node in the context path for which the position code shall be incremented by 1. The position code for all multiple-occurrence nodes with a higher index is set to "0".

by

The set of multiple-occurrence nodes is indexed beginning from the starting node of the context path. An Incremental Position Code indicates the index of the multiple-occurrence node in the context path for which the position code shall be incremented. The position code for all multiple-occurrence nodes with a higher index is set to an initial value.

If Position Codes represent integer numbers, the position code for the multiple-occurrence node indicated by the Incremental Position Code shall be incremented according to the ascending order, i.e. it shall be incremented by "1", and the initial value that the position code for all multiple-occurrence nodes with a higher index is set to is "0".

If Position Codes represent rational numbers, the position codes shall be sorted in the following increment order of rational numbers before they are encoded:

1/2 → 1/4 → 3/4 → 1/8 → 3/8 → 5/8 → 7/8 → 1/16 → …

where, the i-th (i=0,1,2,…) rational number r[i] of this order is expressed by;

r[i] = (2(i+1) +1 - 2^j) / 2^j, where j = 1+int(log2(i+1)).

This order is defined first based on the resolution of the rational numbers which is the order of dividing the area (0,1) into halves repeatedly (Figure AMD1-7 (1)), i.e. the value of denominator. After that the ascending order is applied to the numbers in the same resolution (Figure AMD1-7 (2)).

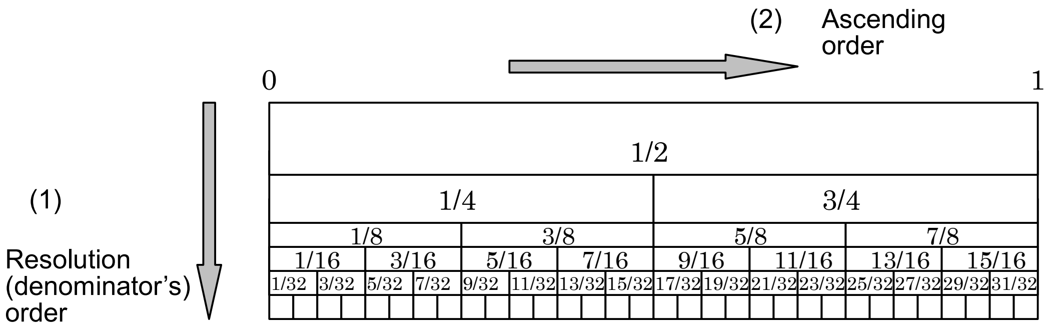


Figure AMD1-7 - Order of rational numbers

The position code for the multiple-occurrence node indicated by the Incremental Position Code shall be incremented according to the order of rational numbers. The initial value of the order is "1/2".

The order of fragment update payloads in a fragment update unit is kept in the ascending order of their rational position code values. After decoding the position codes of rational numbers, the decoded position codes shall be re-sorted into the ascending order of their rational code values and be assigned to the multiple payload in this order.

In subclause 7.6.5.6, replace the following text:

An example for the multiple fragment update payload mode is given below:

by

An example for the multiple fragment update payload mode with integer position codes is given below:

Add the following text and figure at the end of subclause 7.6.5.6:

Figure AMD1-8 shows an example of the encoding/decoding processes for the multiple payload mode with rational position codes. If Position Codes represent rational numbers, the position codes are sorted in the order of rational numbers before they are encoded (Figure AMD1-8 (1)). Once the positions are sorted, incremental position coding is applied to the rational position codes with the increment order of rational numbers and the initial value "1/2". After the position codes are decoded, they are re-sorted in the ascending order (Figure AMD1-8 (2)).

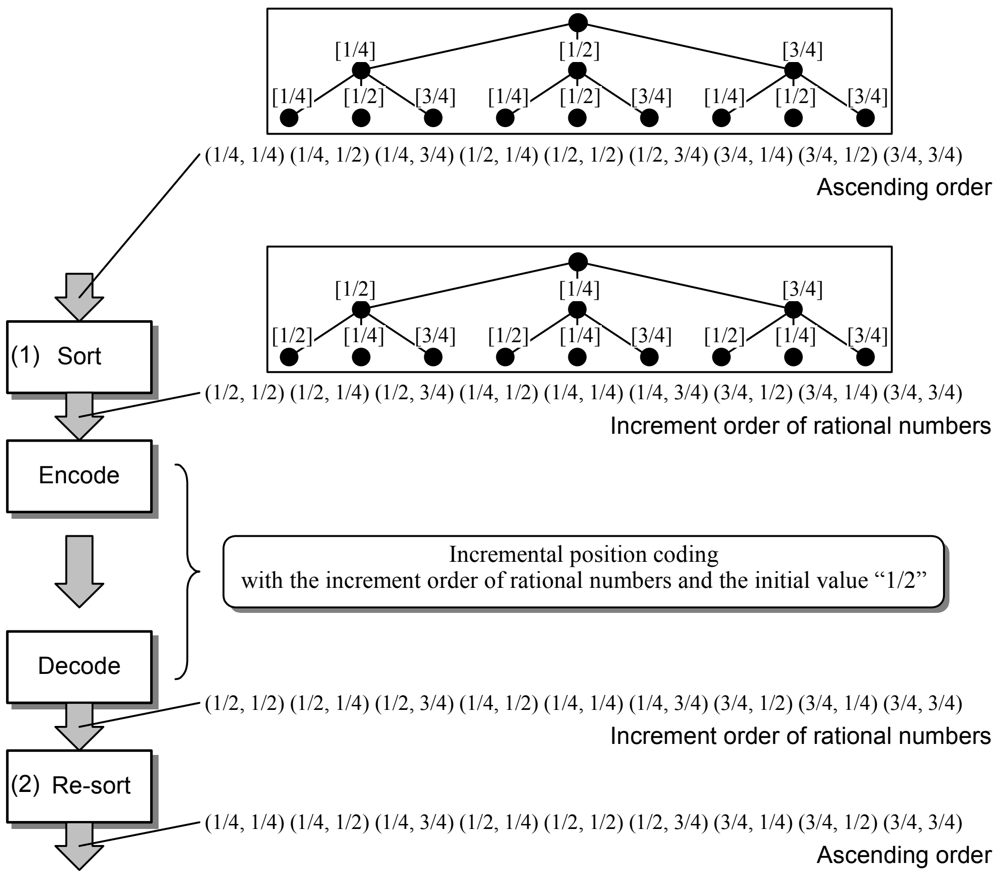


Figure AMD1-8 - Encoding/Decoding processes for the multiple payload mode with rational position codes

Add the following subclause (subclause 7.7):

7.7 Binary Schema Update Unit

7.7.1 Overview

In addition to the initial schema and known additional schemas, the decoder accept unknown additional schemas. These *unknown additional schemas* are subject to updates as described in this subclause. Uknown additional schema updates are carried in an access unit by a *schema update unit*.

A schema update unit is composed of a namespace identifier, a set of code tables to represent global elements, global types and global attributes, followed by a binary encoded schema carrying the schema components definitions. This binary encoded schema is encoded using a specific profile of BiM specified in subclause 7.7.8 using a simple XML schema for schema encoding has been defined for this purpose in this specification.

Note - The binary encoded schema only contains information needed by a BiM decoder to properly decode the bitstream. For instance the binary encoded schema does not carry key, unique elements or block. The schema update feature should not be used to carry XML schemas.

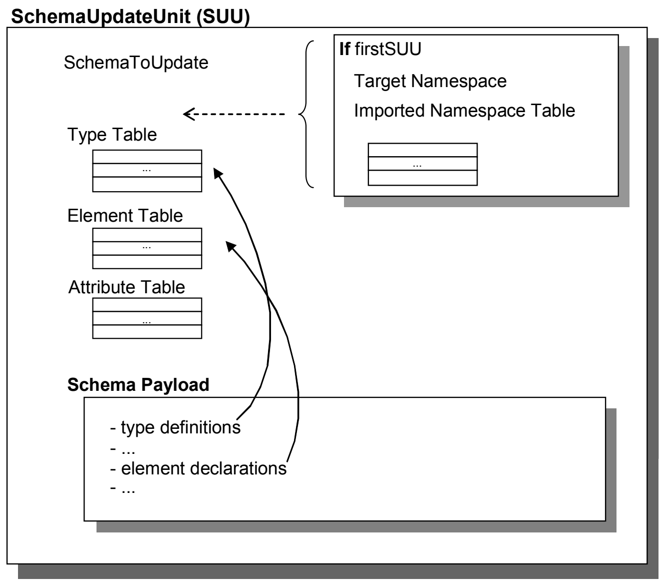


Figure AMD1-9 - A schema update unit

Some constraints are applied to the acquisition of schema update units. A specific schema update unit, the so-called first schema update unit, contains initialization information and shall be acquired by the decoder before any description conformant to the transmitted additional schema is decoded. The decoder behavior in case of a first schema update units is missed is not normative. A schema definition already transmitted shall not change during binary description stream lifetime and there shall not be two schema identifiers associated to the same namespace.

The full schema is not always necessary for the decoding of a particular binary description stream. To avoid unnecessary transmission, a schema update unit may contain only the definitions that are required for the decoding of the description stream.

Once received by the decoder, a schema update unit immediately updates schema information managed by the decoder. All the optimised decoders associated to existing types are immediately applied to all types they derive from in accordance to the rules defined for the optimized decoders in Clause 9.

7.7.2 Syntax

SUU () {	**Number of bits**	**Mnemonic**
SchemaToUpdate	ceil(log2(NumberOfAdditionalSchemas))	uimsbf
FirstSUU	1	blsbf
If (FirstSUU) {		
NamespaceURI_Length	8+	vluimsbf8
NamespaceURI_String	8*NamespaceURILength	blsbf
ImportedNamespaceTable()		
}		
SchemaTypeTable(SchemaToUpdate)		
SchemaElementTable(SchemaToUpdate)		
SchemaAttributeTable(SchemaToUpdate)		
BinaryEncodedSchema(SchemaToUpdate)		
}		

7.7.3 Semantics

Name	*Definition*
SchemaToUpdate	Specifies the index in the table of schema which is updated by this fragment update unit.
FirstSUU	This flag is set to true if the SUU is a FirstSUU.
NamespaceURI_Length	Signals the length in bytes of the NamespaceURI_String.
NamespaceURI_String	UTF-8 representation of the namespace URI on which the SUU applies.
ImportedNamespaceTable	This table conveys the namespace referenced in the binary encoded schema as specified in subclause 7.7.4.
SchemaTypeTable	This table conveys the type code tables as specified in subclause 7.7.5.
SchemaElementTable	This table conveys the global elements and their possible substitutions as specified in subclause 7.7.6.
SchemaAttributeTable	This table conveys the global attributes as specified in subclause 7.7.7.
BinaryEncodedSchema	This conveys the binary encoded schema definitions as specified in subclause 7.7.8.

7.7.4 Imported NamespaceTable

7.7.4.1 Overview

This table conveys the table of namespaces that are referenced in the binary encoded schema.

7.7.4.2 Syntax

ImportedNamespaceTable(Schema){	**Number of bits**	**Mnemonic**
NumberOfImportedNamespaces	8+	vluimsbf8
for (i=0; i < NumberOfNamespaces; i++) {		
ImportedNamespace_Length[i]	8+	vluimsbf8
ImportedNamespace[i]	8* MappedNamespace_Length[i]	blsbf
}		
}		

7.7.4.3 Semantics

Name	*Definition*
NumberOfImportedNamespaces	Indicates the number of namespaces that can be referred by a schema component definition in the binary encoded schema.
ImportedNamespace_Length[i]	Indicates the size in bytes of the `ImportedNamespace[k]`. A value of zero is forbidden.
ImportedNamespace[i]	This is the UTF-8 representation of the namespace.

7.7.5 Schema Type Table

7.7.5.1 Overview

This table conveys the table of global types defined in the namespace on which the SUU applies.

7.7.5.2 Schema Type Table Syntax

SchemaTypeTable (Schema){	**Number of bits**	**Mnemonic**
if (FirstSUU) {		
NumberOfGlobalTypes	8+	vluimsbf8
ExternallyCastableTypeTable(Schema)		
}		
PartialTransmission	1	blsbf
if (PartialTransmission){		
NumberOfTransmittedTypes	5+	vluimbsf5
for (i=0; i < NumberOfTransmittedTypes; i++) {		
TransmittedType	5+	vluimsbf5
NumOfSubtypes[TransmittedType]	5+	vluimsbf5
} else {		
for (i=0; i < NumberOfGlobalTypes; i++) {		
NumOfSubtypes[i]	5+	vluimsbf5
}		
}		

7.7.5.3 Schema Type Table Semantics

The types are defined in the order of their type codes within the namespace as specified in subclause 7.6.5.4.

Name	*Definition*
NumberOfGlobalTypes	Defines the number of global types defined in the namespace.
PartialTransmission	Indicates that the transmission of the type table is partial.
NumberOfTransmittedTypes	Indicates the number of type definitions that are transmitted in the current SUU.

TransmittedType	Indicates the TypeCode of the type to be updated.
NumOfSubtypes[TransmittedType]	Indicates the number of subtype of the type ‘TransmittedType’ in the namespace.
NumOfSubtypes[i]	Indicates the number of subtype of the ‘ith’ type of the namespace.

7.7.5.4 Externally Castable Type Table Syntax

ExternallyCastableTypeTable(Schema) {	**Number of bits**	**Mnemonic**
IsThereExternallyCastableType	1	blsbf
If (IsThereExternalCastableType) {		
all_type_externally_castable	1	blsbf
If(!all_type_externally_castable) {		
NumberOfExternallyCastableType	5+	vluimsbf5
for(i=0;i< NumberOfExternallyCastableType; i++){		
ExternallyCastableType	ceil(log2(NumberOfGlobalTypes in Schema))	blsbf
}		

7.7.5.5 Externally Castable Type Table Semantics

This table allows to specify which types can be subject to a type casting where the subtype is defined in an other namespace than the one carried in the schema update unit.

Name	*Definition*
IsThereExternallyCastableType	Signals that some types in the schema to update can be casted into types defined defined in other namespaces.
all_type_externally_castable	Signals that all types in the schema to update can be casted into types defined in other namespaces.
NumberOfExternallyCastableType	Indicates the number of types that can be casted into types defined in other namespaces.
ExternallyCastableType	Indicates the type code of a type which can be casted into types defined in other namespaces. In case this element is subject to a substitution (subclause 7.6.5.4), its `external_type_cast_possible` flag is set to ‘1’.

7.7.6 Schema Element Tables

7.7.6.1 Overview

This table conveys the global elements and their substitutions on which the SUU applies. They are used to efficiently encode XML Schema substitution groups.

7.7.6.2 Schema Element Table Syntax

SchemaElementTable (Schema){	**Number of bits**	**Mnemonic**
if (FirstSUU) {		
NumberOfGlobalElements	8+	vluimsbf8
ExternallyCastableElementTable(Schema)		
}		
PartialTransmission	1	blsbf
if (PartialTransmission){		
NumberOfTransmittedElements	5+	vluimbsf5
for (i=0; i < NumberOfTransmittedElements; i++) {		
TransmittedElement	5+	vluimsbf5
NumOfSubstituteElements[TransmittedElement]	5+	vluimsbf5
} else {		
for (i=0; i < NumberOfGlobalElements; i++) {		
NumOfSubstituteElements[i]	5+	vluimsbf5
}		
}		

7.7.6.3 Schema Element Table Semantics

The elements are defined in the order of their element codes within the namespace as specified in subclause 7.6.5.3.

Note – `HierarchyBasedSubstitutionCondingFlag` is always set to true when additional schemas are supported.

Name	*Definition*
NumberOfGlobalElements	Defines the number of global elements defined in the namespace.
PartialTransmission	Indicates that the transmission of the element table is partial.
NumberOfTransmittedElements	Indicates the number of element definitions that are transmitted in the current SUU.
TransmittedElement	Indicates the `SubstitutionSelect` code of the element to be updated.
NumOfSubstituteElements [TransmittedElement]	Indicates the number of substitute elements of the TransmittedElement in the updated namespace.
NumOfSubstituteElements [i]	Indicates the number of substitute elements of the 'ith' element of the namespace.

7.7.6.4 Externally Substitutable Element Table Syntax

ExternallySubstitutableElementTable(Schema) {	**Number of bits**	**Mnemonic**
IsThereExternallySubstitutableElement	1	blsbf
If (IsThereExternallySubstitutableElement) {		
all_element_externally_substitutable	1	blsbf
If(!all_element_externally_substitutable) {		
NumberOfExternallySubstitutableElement	5+	vluimsbf5
for(i=0;i< NumberOfExternallySubstitutableElement;i++){		
ExternallySubstitutableElement	ceil(log2(Number OfGlobalElements in Schema))	blsbf
}		

7.7.6.5 Externally Substitutable Element Table Semantics

This table allows to specify which elements can be subject to an “external” element substitution i.e. a substitution in which the substitute element is defined in an other namespace.

Name	*Definition*
IsThereExternallySubstituableElement	Signals that some elements in the schema to update can be substituted into elements defined defined in other namespaces.
all_element_externally_substitutable	Signals that all elements in the schema to update can be substituted into elements defined in other namespaces.
NumberOfExternallySubstitutableElement	Indicates the number of elements that can be substituted into elements defined in other namespaces.
ExternallySubstitutableElement	Indicates the element code of an element which can be substituted into elements defined in other namespaces. In case this element is subject to a substitution (subclause 7.6.5.3), its `external_element_substitution_possible` flag is set to ‘1’.

7.7.7 Schema Attribute Table

7.7.7.1 Overview

This table conveys the global attributes of the updated schema.

7.7.7.2 Syntax

SchemaAttributeTable(Schema){	**Number of bits**	**Mnemonic**
if (FirstSUU) {		
NumberOfGlobalAttributes	8+	vluimsbf8
}		
PartialTransmission	1	blsbf
if (PartialTransmission){		
NumberOfTransmittedAttributes	5+	vluimbsf5
for (i=0; i < NumberOfTransmittedAttributes; i++) {		
TransmittedAttribute	ceil(log2(NumberOfGlobal Attributes in SchemaToUpdate))	uimsbf
}		
}		

7.7.7.3 Semantics

Name	*Definition*
NumberOfGlobalAttributes	Defines the number of global attributes defined in the namespace.
PartialTransmission	Indicates that the transmission of the element table is partial.
NumberOfTransmittedAttributes	Indicates the number of global attribute definitions that are transmitted in the current SUU.
TransmittedAttribute	Indicates the code of the received attribute.

7.7.8 Binary Encoded Schema

7.7.8.1 Overview

Each schema update unit carries a set of schema components definition in its `BinaryEncodedSchema`. This set is represented by an XML file conformant to a specific schema called the schema for encoding schema components. It is carried in a BiM encoded form using the schema for encoding schema components.

Note – The schema for encoding schema components is similar in its spirit to the XML Schema for schema. It has been however dedicated to the encoding of XML in BiM and not for validation as it is the case for the XML Schema for schema. It therefore concentrates on the features that are only used by a BiM decoder for decoding purposes only.

7.7.8.2 Decoding schema components using BiM

7.7.8.2.1 Binary Encoded Schema - DecoderInit

The following specific `DecoderInit` is used by the decoder for the decoding of binary encoded schema.

DecoderInit() {	**Value**	**Number of bits**
SystemsProfileLevelIndication	0x00	8
UnitSizeCode	000	3
NoAdvancedFeatures	0	1
ReservedBits	1111	4
AdvancedFeatureFlags_Length	0x01	8
InsertFlag	0	1
AdvancedOptimisedDecodersFlag	1	1
AdditionalSchemaFlag	0	1
AdditionalSchemaUpdatesOnlyFlag	0	1
FragmentReferenceFlag	0	1
MPCOnlyFlag	0	1
ReservedBitsZero	00	2
NumberOfSchemas	1	8+
SchemaURI_Length[0]	0x20 (i.e. 32)	8+
SchemaURI[0]	"urn:mpeg:mpeg7:schema Update:2002"	8* SchemaURI_Length[0]
LocationHint_Length[0]	0x00	8
NumOfAdvancedOptimisedDecoderTypes	0x01	8+
AdvancedOptimisedDecoderTypeURI_Length[0]	0x40 (i.e. 64)	8+
AdvancedOptimisedDecoderTypeURI[0]	"urn:mpeg:mpeg7:systems :SystemsAdvancedOptimisedDecodersCS:2003:1"	8* AdvancedOptimisedDecoderTypeURI_Length[0]
AdvancedOptimisedDecodersConfig () {		
NumOfAdvancedOptimisedDecoderInstances	0x00	8
NumOfMappings	0x00	8
}		
InitialDescription_Length	0x00	8
}		
}		

7.7.8.2.2 Binary Encoded Schema - Access Unit Constraints

A schema update unit is carried in one access unit constrained by the following rules:

- The access unit shall contain only one fragment update unit ;
- The fragment update unit shall update the top most node ;
- The fragment update unit shall use a 'AddContent" command ;
- The fragment update unit shall have a context mode code set to '001' ;
- The `lengthCodingMode` code of the fragment update payload shall be set to '00' ;

— The `hasDeferredNodes` flag of the fragment update payload shall be set to ‘0’ ;

— The `hasTypeCasting` flag of the fragment update payload shall be set to ‘1’ ;

— The `hasNoFragmentReference` flag of the fragment update payload shall be set to ‘1’.

Moreover in the fragment update payload the following rules applies:

— The references (e.g. “base” or “type” attributes in XML Schema) to elements, types or attributes are encoded with “SchemaID” (in the local imported namespaces table) + “global code”

7.7.8.2.3 Binary Encoded Schema – Schema

The encoded schema shall respect the following constraints:

— Global types, elements and attributes are encoded in the same order than the one defined by their respective tables in the schema update units (`SchemaTypeTable`, `SchemaElementTable` and `SchemaAttributeTable`);

— Attributes in complex type definitions are sorted according to their expanded name;

— Content models shall be normalized as described in subclause 8.5.2.2.4.

7.7.8.3 Mapping schema components to the schema for encoding

7.7.8.3.1 Overview

The following subclauses specify the syntax elements and associated semantics of the schema for encoding schema updates.

The following schema wrapper shall be applied to the syntax defined in subclause 7.7.8.3.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"

   xmlns:m7s="urn:mpeg:mpeg7:systems:encodingschema:amd1:2004"
   targetNamespace="urn:mpeg:mpeg7:systems:encodingschema:amd1:2004"
     elementFormDefault="qualified"
     attributeFormDefault="unqualified">

   <!-- here clause 7.7.8.3 schema definitions -->

</schema>
```

7.7.8.3.2 Main schema element and type

7.7.8.3.2.1 Syntax

```
<xs:element name="schema" type="schemaType"/>

<xs:complexType name="schemaType">
  <xs:sequence>
```

```
   <xs:element name="typeDefinitions" type="typeDefinitionsType" minOccurs="0"/>
   <xs:element name="anonymousTypeDefinitions"
type="anonymousTypeDefinitionsType"
                                        minOccurs="0"/>
   <xs:element name="elementDeclarations" type="elementDeclarationsType"
                                        minOccurs="0"/>
   <xs:element name="attributeDeclarations" type="attributeDeclarationsType"
    minOccurs="0"/>
  </xs:sequence>
  <xs:attribute name="elementFormDefault"   type="qualificationType"
                         use="optional" default="unqualified"/>
  <xs:attribute name="attributeFormDefault" type="qualificationType"
                         use="optional" default="unqualified"/>
</xs:complexType>

<xs:simpleType name="qualificationType">
  <xs:restriction base="xs:string">
   <xs:enumeration value="qualified"/>
   <xs:enumeration value="unqualified"/>
  </xs:restriction>
</xs:simpleType>
```

7.7.8.3.2.2 Semantics

Name	*Definition*
schema	The root element of the schema update.
schemaType	The set of schema components updated by this schema update. Note - The namespace of this schema is encoded within the SchemaUpdateUnit and therefore not encoded here.
typeDefinitions	conveys the list of named types (or type globally defined).
anonymousTypeDefinitions	conveys the list of anonymous types (or type locally defined).
elementDeclarations	conveys the list of global elements.
attributeDeclarations	conveys the list of global attributes.
elementFormDefault	identical to the ‘elementFormDefault’ attribute defined in XML schema.
attributeFormDefault	identical to the ‘attributeFormDefault’ attribute defined in XML schema.
qualificationType	A type used to define the qualification (qualified/unqualified) of elements and attributes. This type is used by the 'form', 'elementFormDefault' and 'attributeFormDefault' attributes.

7.7.8.3.3 Element Declaration

7.7.8.3.3.1 Syntax

```
<xs:complexType name="elementDeclarationsType">
  <xs:sequence minOccurs="0" maxOccurs="unbounded">
    <xs:element name="globalElement">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="elementTypeReference" type="typeReferenceType"/>
        </xs:sequence>
        <xs:attribute name="name" type="xs:string" use="required"/>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>

<xs:simpleType name="nameStringType">
  <xs:restriction base="xs:string"/>
</xs:simpleType>
```

7.7.8.3.3.2 Semantics

Name	*Definition*
elementDeclarationsType	The list of all global element declarations carried by this schema update unit. This list shall be ordered as specified in the specification (See subclause 7.7.6).
nameStringType	Defines the type of all the names used in the schema i.e. attribute, element and type names.

7.7.8.3.4 Type Declaration

7.7.8.3.4.1 Syntax

```
<xs:complexType name="typeDefinitionsType">
  <xs:choice maxOccurs="unbounded">
    <xs:element name="complexType" type="namedComplexTypeType"/>
    <xs:element name="simpleType" type="namedSimpleTypeType"/>
  </xs:choice>
</xs:complexType>

<xs:complexType name="anonymousTypeDefinitionsType">
  <xs:element name="complexType" type="anonymousComplexTypeType"
                                maxOccurs="unbounded"/>
  <xs:element name="simpleType" type="anonymousSimpleTypeType"
    maxOccurs="unbounded"/>
</xs:complexType>
```

7.7.8.3.4.2 Semantics

Name	*Definition*
typeDefinitionsType	This type conveys the list of all the global types (complex and simple) carried by this schema update. The global type (or named type) are the types globally defined in an XML schema declaration.This list shall be ordered as defined in 7.7.5. The number of types contained in this list is encoded in the `SchemaTypeTable` (see 7.7.5.2)
complexType	conveys a complex type definition.
simpleType	conveys a simple type definition.
anonymousTypeDefinitionsType	The list of all the anonymous types (or locally defined). No order is required on this list. In the case of a partial transmission, all the anonymous type required for resolving the type referencing mechanism shall be present in the schema update unit (see type referencing mechanism in 7.7.8.3.7).
complexType	conveys a complex type definition.
simpleType	conveys a simple type definition.

7.7.8.3.5 Type Definition

7.7.8.3.5.1 Syntax

```
<xs:complexType name="typeType" abstract="true">
   <xs:sequence>
     <xs:element name="derivation" minOccurs="0">
       <xs:complexType>
         <xs:sequence>
              <xs:element name="baseTypeReference" type="typeReferenceType"
                                                minOccurs="1"/>
         </xs:sequence>
         <xs:attribute name="type" type="derivationType"
           use="required"/>
       </xs:complexType>
     </xs:element>
   </xs:sequence>
</xs:complexType>

<xs:simpleType name="derivationType">
   <xs:restriction base="xs:string">
     <xs:enumeration value="extension"/>
     <xs:enumeration value="restriction"/>
   </xs:restriction>
</xs:simpleType>

<xs:complexType name="complexTypeType" abstract="true">
   <xs:complexContent>
     <xs:extension base="typeType">
       <xs:sequence>
         <xs:element name="attributes" minOccurs="0">
```

```
          <xs:complexType>
            <xs:sequence>
              <xs:choice minOccurs="0" maxOccurs="unbounded">
                <xs:element name="attribute" type="localAttributeType"/>
                <xs:element name="attributeRef" type="attributeRefType"/>
              </xs:choice>
              <xs:element name="anyAttribute" type="anyAttributeType"
                minOccurs="0"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="content" type="contentModelType"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="namedComplexTypeType" >
  <xs:complexContent>
    <xs:extension base="complexTypeType">
      <xs:attribute name="name" type="nameStringType" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="anonymousComplexTypeType">
  <xs:complexContent>
    <xs:extension base="complexTypeType">
      <xs:attribute name="id" type="AnonymousTypeIDType" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="simpleTypeType" abstract="true">
  <xs:complexContent>
    <xs:extension base="typeType">
      <xs:sequence>
        <xs:choice>
          <xs:element name="list">
            <xs:complexType>
              <xs:sequence minOccurs="1">
                <xs:element name="itemTypeReference" type="typeReferenceType"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
          <xs:element name="union">
            <xs:complexType>
              <xs:sequence minOccurs="1" maxOccurs="unbounded">
                <xs:element name="memberTypeReference" type="typeReferenceType"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:choice>
        <xs:sequence minOccurs="0" maxOccurs="unbounded">
          <xs:element name="facet" type="facetType"/>
        </xs:sequence>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

```
<xs:complexType name="namedSimpleTypeType" >
   <xs:complexContent>
     <xs:extension base="simpleTypeType">
       <xs:attribute name="name" type="nameStringType" use="required"/>
     </xs:extension>
   </xs:complexContent>
</xs:complexType>

<xs:complexType name="anonymousSimpleTypeType">
   <xs:complexContent>
     <xs:extension base="simpleTypeType">
       <xs:attribute name="id" type="AnonymousTypeIDType" use="required"/>
     </xs:extension>
   </xs:complexContent>
</xs:complexType>
```

7.7.8.3.5.2 Semantics

Name	*Definition*
typeType	The abstract type of all types. This type defines derivation information for its subtypes `complexTypeType` or `simpleTypeType`.
derivation	defines the derivation methods by which a type is defined from its supertype (extension or restriction).
baseTypeReference	identifies the supertype of the types as defined in XML schema.
derivationType	identifies the derivation type (i.e. extension or restriction) as defined in XML schema.
complexTypeType	The abstract type of all complex type definitions. Its subtypes are `namedComplexTypeType` (used in case of type globally defined) and `anonymousComplexTypeType` (used in case of type locally defined)
attributes	The list of attributes declared within the complex type. The list shall be transmitted in the order defined in 8.5.3. In case of derivation by restriction, the entire list of attributes shall be listed. In case of derivation by extension, only new attributes shall be listed.
attribute	a locally defined attribute (cf. 7.7.8.3.8).
attributeRef	a reference to a global attribute defined in a schema (cf. 7.7.8.3.8).
anyAttribute	indicates the use of the anyAttribute (cf. 7.7.8.3.8).
content	defines the content model of a complexType. (cf.7.7.8.3.9).
namedComplexTypeType	A globally defined complex type. Its name shall be present.
anonymousComplexTypeType	A locally defined complexType. an ID is associated to each anonymous type (cf. 7.7.8.3.7)
simpleTypeType	The abstract type of all simple type definitions. Its subtypes are `namedSimpleTypeType` (used in case of type globally defined) and `anonymousSimpleTypeType` (used in case of type locally defined).

list	If the Simple type is defined as a list, this element contains a reference to the item type which constitutes the element of the list
union	If the simple type is defined as an union, this elements contains a reference to the different possible items of the union. The order of the elements has the same semantics than the defined in XML schema.
facet	conveys the facets of a simple type
namedSimpleTypeType	a globally defined simple type. Its name shall be present.
anonymousSimpleTypeType	a locally defined simpleType (or anonymous type). An ID shall be associated to each anonymous type (cf. 7.7.8.3.7)

7.7.8.3.6 Type and Element Referencement

7.7.8.3.6.1 Syntax

```
<xs:complexType name="typeReferenceType">
  <xs:choice>
    <xs:element name="namedTypeReference">
      <xs:complexType>
        <xs:attribute name="NamespaceID" type="NamespaceIDType"
              use="optional"/>
        <xs:attribute name="TypeID" type="TypeIDRefType"
          use="required"/>
      </xs:complexType>
    </xs:element>
    <xs:element name="anonymousTypeReference">
      <xs:complexType>
        <xs:attribute name="idref" type="AnonymousTypeIDRefType"
          use="required"/>
      </xs:complexType>
    </xs:element>
  </xs:choice>
</xs:complexType>

<xs:complexType name="elementReferenceType">
  <xs:sequence>
    <xs:element name="namedElementReference">
      <xs:complexType>
        <xs:attribute name="NamespaceID" type="NamespaceIDType"
          use="optional"/>
        <xs:attribute name="ElementID" type="ElementIDRefType"
          use="required"/>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>

<xs:simpleType name="NamespaceIDType">
  <xs:restriction base="xs:nonNegativeInteger"/>
</xs:simpleType>

<xs:simpleType name="TypeIDRefType">
  <xs:restriction base="xs:nonNegativeInteger"/>
```

```
</xs:simpleType>

<xs:simpleType name="ElementIDRefType">
  <xs:restriction base="xs:nonNegativeInteger"/>
</xs:simpleType>

<xs:simpleType name="AnonymousTypeIDType">
  <xs:restriction base="xs:nonNegativeInteger"/>
</xs:simpleType>

<xs:simpleType name="AnonymousTypeIDRefType">
  <xs:restriction base="AnonymousTypeIDType"/>
</xs:simpleType>
```

7.7.8.3.6.2 Semantics

Name	*Definition*
typeReferenceType	A reference to a type. This type is used when an element refers to a type or when a type refers to its super type. Two kind of types can be referenced: a named type (globally defined) or a anonymous type (locally defined).
namedTypeReference	The 'NamespaceID' attribute gives the index of the namespace, in the imported namespace table, in which the type is defined. The 'TypeID' attribute gives the index of the global type in the schema identified by the NamespaceID. If 'NamespaceID' attribute is not present, the namespace of the global type is the target namespace of the Schema Update.
anonymousTypeReference	The idref attribute gives the index in the table of anonymous type..
elementReferenceType	A reference to an element.The 'NamespaceID' attribute gives the index of the namespace, in the imported namespace table, in which the element is defined. The 'ElementID' attribute gives the index of the global element in the schema identified by the NamespaceID. If 'NamespaceID' attribute is not present, the namespace of the global element is the target namespace of the Schema Update.
NamespaceIDType	Defines the namespace id and refers to the table of imported namespace defined in the Schema Update Unit.
TypeIDRefType	Defines the type id and refers to the table of types carried in the Schema Update Unit.
ElementIDRefType	Defines the element id and refers to the table of types carried in the Schema Update Unit.
AnonymousTypeIDType	Defines the type id of an anonymous type. The scope of this id is limited to the current schema update unit. Therefore, in case of non complete transmission, Id values can be reused to identify different anonymous types.
AnonymousTypeIDRefType	Defines a reference to an anonymous type.

7.7.8.3.7 Attribute definition

7.7.8.3.7.1 Syntax

```
<xs:complexType name="attributeDeclarationsType">
  <xs:sequence>
    <xs:element name="attribute" type="globalAttributeType"
      maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="attributeType" abstract="true">
  <xs:sequence>
    <xs:element name="typeReference" type="typeReferenceType"/>
  </xs:sequence>
  <xs:attribute name="name" type="nameStringType"  use="required"/>
  <xs:attribute name="defaultValue" type="xs:string" use="optional"/>
</xs:complexType>

<xs:complexType name="attributeRefType">
  <xs:attribute name="idref" type="xs:IDREF" use="required"/>
</xs:complexType>

<xs:complexType name="localAttributeType">
  <xs:complexContent>
    <xs:extension base="attributeType">
      <xs:attribute name="use" type="useType" use="required"/>
      <xs:attribute name="form" type="qualificationType" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="globalAttributeType">
  <xs:complexContent>
    <xs:extension base="attributeType"/>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="anyAttributeType">
</xs:complexType>

<xs:simpleType name="useType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="optional"/>
    <xs:enumeration value="required"/>
  </xs:restriction>
</xs:simpleType>
```

7.7.8.3.7.2 Semantics

Name	*Definition*
attributeDeclarationsType	The list of all the global attribute declarations carried by this schema update. This list is ordered as specified in subclause 7.7.7.

attributeType	An abstract type conveying the definition of an attribute. The type of the defined attribute is identified by a type reference. The name of the attribute shall be present. The defaultValue, if it exists, shall be encoded.
attributeRefType	A reference to a global attribute. It is used when a type references a global attribute.
localAttributeType	Defines the type of an attribute defined within a complex type. The `use` attribute shall be present. Its semantics is identical to the one defined in XML schema. The `form` attribute shall be instantiated, its semantics is identical to the one defined in XML schema.
globalAttributeType	Defines the type of a global attribute.
anyAttributeType	Indicates that any attribute of any schema can be present in the complexType.
useType	The type of the `use` attribute. It has two possible values : ‘optional’ or ‘required’.

7.7.8.3.8 Content Model

7.7.8.3.8.1 Syntax

```
<xs:complexType name="contentModelType" abstract="true"/>

<xs:complexType name="emptyContentModelType">
  <xs:complexContent>
    <xs:extension base="contentModelType"/>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="simpleContentModelType">
  <xs:complexContent>
    <xs:extension base="contentModelType">
      <xs:sequence>
        <xs:element name="simpleTypeReference"
          type="typeReferenceType"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="complexContentModelType">
  <xs:complexContent>
    <xs:extension base="contentModelType">
      <xs:sequence>
        <xs:element name="particle" type="particleType"/>
      </xs:sequence>
      <xs:attribute name="mixed" type="xs:boolean" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

7.7.8.3.8.2 Semantics

Name	*Definition*
contentModelType	This abstract type defines the content model of a complex type. It has three subtypes addressing the three content models defined by XML Schema: `emptyContentModelType`, `simpleContentModelType` and `complexContentModelType`.
emptyContentModelType	An empty content model.
simpleContentModelType	A simple content model. It includes a reference to the simple type which defines the content model of a complex type.
complexContentModelType	A complex content model. This model contains a particle (see XML schema) and its 'mixed' attribute shall be instantiated.

7.7.8.3.9 Facet Definition

7.7.8.3.9.1 Syntax

```
<xs:complexType name="facetType" abstract="true">
  <xs:attribute name="name" type="possibleFacet" use="required"/>
  <xs:attribute name="value" type="xs:string"/>
</xs:complexType>

<xs:simpleType name="possibleFacet">
  <xs:restriction base="xs:string">
    <xs:enumeration value="maxExclusive"/>
    <xs:enumeration value="minExclusive"/>
    <xs:enumeration value="minInclusive"/>
    <xs:enumeration value="maxInclusive"/>
    <xs:enumeration value="enumeration"/>
    <xs:enumeration value="length"/>
    <xs:enumeration value="minLength"/>
    <xs:enumeration value="maxLength"/>
  </xs:restriction>
</xs:simpleType>
```

7.7.8.3.9.2 Semantics

Name	*Definition*
facetType	Defines the possible facets associated to a simple type. A facet is composed of a name and a value. The facet mechanism is equivalent to the one defined in XML schema.
possibleFacet	The set of possible facets are limited to the ones used by BiM (see 8.5.4).

7.7.8.3.10 Particle Defintion

7.7.8.3.10.1 Syntax

```
<xs:complexType name="particleType" abstract="true">
  <xs:attribute name="minOccurs" type="xs:unsignedInt" default="1"/>
  <xs:attribute name="maxOccurs" type="occurrenceType" default="1"/>
</xs:complexType>

<xs:complexType name="anyParticleType">
  <xs:complexContent>
    <xs:extension base="particleType"/>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="element">
  <xs:complexContent>
    <xs:extension base="particleType">
      <xs:sequence>
        <xs:element name="type" type="typeReferenceType"/>
      </xs:sequence>
      <xs:attribute name="name" type="nameStringType"/>
      <xs:attribute name="form" type="qualificationType"/>
      <xs:attribute name="nillable" type="xs:boolean"  use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="elementRef">
  <xs:complexContent>
    <xs:extension base="particleType">
      <xs:sequence>
        <xs:element name="ref" type="elementReferenceType"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="modelGroupType" abstract="true">
  <xs:complexContent>
    <xs:extension base="particleType">
      <xs:sequence maxOccurs="unbounded">
        <xs:element name="particle" type="particleType"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="sequence">
  <xs:complexContent>
    <xs:extension base="modelGroupType"/>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="all">
  <xs:complexContent>
    <xs:extension base="modelGroupType"/>
  </xs:complexContent>
```

```
</xs:complexType>

<xs:complexType name="choice">
  <xs:complexContent>
    <xs:extension base="modelGroupType"/>
  </xs:complexContent>
</xs:complexType>

<xs:simpleType name="occurrenceType">
  <xs:union memberTypes="unboundedType xs:unsignedInt"/>
</xs:simpleType>

<xs:simpleType name="unboundedType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="unbounded"/>
  </xs:restriction>
</xs:simpleType>
```

7.7.8.3.10.2 Semantics

Name	*Definition*
particleType	An abstract type defining the type of all particles (cf. XML Schema). The range of occurrences are defined by the `minOccurs` and `maxOccurs` attributes.
anyParticleType	The any wildcard. This particle indicates that any global element of any namespace can be present in the description.
element	A local declaration of an element. This declaration is composed of a reference to a type, a name, a qualification form and a nillable property.
elementRef	A reference to an element globally defined.
modelGroupType	This particle is the super type of all groups: sequence, choice and all
sequence	A 'sequence' particle.
all	An 'all' particle.
choice	A 'choice' particle.
occurrenceType	The type for the `maxOccurs` attribute. Its value can be either an 'xs:unsignedInt' or the string value 'unbounded'.

In subclause 8.1, replace the following sentence:

The binary fragment update payload syntax (`FUPayload`) is specified in subclause . It is composed of flags which define some decoding modes and a payload content which is either an `element` or a simple value (`simpleType`). …

by

The binary fragment update payload syntax (`FUPayload`) is specified in subclause 8.3. It is composed of flags which define some decoding modes and a payload content which is either an `element`, a simple value (`simpleType`) or a reference to a payload. …

In subclause 8.3.2.1, replace the DecodingModes syntax table by:

DecodingModes () {	**Number of bits**	**Mnemonic**
lengthCodingMode	2	bslbf
hasDeferredNodes	1	bslbf
hasTypeCasting	1	bslbf
hasNoFragmentReference	1	bslbf
ReservedBits	3	bslbf
}		

In subclause 8.3.2.2, add the following semantic after the `hasTypeCasting` semantic:

hasNoFragmentReference	A flag which specifies if this fragment update payload contains a fragment reference or the encoded fragment. This 1-bit flag can have the following values: — 0 – `hasNoFragmentReference` is equal to false, — 1 – `hasNoFragmentReference` is equal to true.

In subclause 8.4.1.1 replace the `Element` syntax table by:

Element (Enumeration SchemaModeStatus, SchemaComponent theType) {	**Number of bits**	**Mnemonic**
if (!hasNoFragmentReference) {		
FragmentReference()		
} else if (NumberOfSchemas >1) {		
if (SchemaModeStatus == "hot") {		
SchemaModeUpdate	1-3	vlclbf
}		
if (ElementContentDecodingMode == "mono"){		
Mono-VersionElementContent(ChildrenSchemaMode, theType)		
} else {		
Multiple-VersionElementContent(ChildrenSchemaMode, theType)		
}		
} else {		
Mono-VersionElementContent(“mono”, theType)		
}		
}		

In subclause 8.4.1.2, add the following semantic at the first position:

FragmentReference	See 8.4.8.

In subclause 8.4.3.3, replace the syntax element `SchemaIDOfSubstitution` *by:*

SchemaIDOfSubstitution	ceil(log2(`NumberOfSchemas + NumberOfAdditionalSchemas`))	bslbf

In subclause 8.4.3.4, replace the semantic of `SchemaIDOfSubstitution` *by:*

SchemaIDOfSubstitution	The version identifier which refers to the schema where the substitute element is defined. Its value is the index of the URI in the `SchemaURI` array defined in 7.2 (optionally extended with the list of additional schemas).

In subclause 8.4.4.1, replace the syntax element of `SchemaID` *by:*

SchemaID	ceil(log2(`NumberOfSchemas + NumberOfAdditionalSchemas`))	bslbf

In subclause 8.4.4.2, replace the semantic of `SchemaID` *by:*

SchemaID	Identifies the schema which is needed to decode this `ElementContentChunk`. Its value is the index of the URI in the `SchemaURI` array defined in 7.2 (optionally extended with the list of additional schemas).

In subclause 8.4.7.1, replace the syntax table of `SimpleType` *by:*

SimpleType(SchemaComponent theType) {	**Number of bits**	**Mnemonic**
If (! AdvancedOptimisedDecodersFlag) {		
if (useOptimisedDecoder(theType)) {		
optimisedDecoder(theType)		
} else {		
defaultDecoder(theType)		
}		
} else {		
if (numOfMappedOptimisedDecoder(theType) !=0) {		
optimisedDecoderID	ceil(log2(number of decoders associated to this type))	blsbf
advancedOptimisedDecoder(theType, optimisedDecoderID)		

} else {		
defaultDecoder(theType)		
}		
}		

In subclause 8.4.7.2, add the following text at the beginning of the subclause:

A simple type can be associated to a single default simple type decoder, a single simple optimised decoder and one or several optimised decoders. This syntax table describes the process to select the proper simple type decoder for each simple type to be decoded.

In subclause 8.4.7.2, add the following semantic after the `useOptimisedDecoder` semantic:

numOfMappedOptimisedDecoder ()	Returns the number of advanced optimised decoders associated to the type `theType` as described in Clause 9.

In subclause 8.4.7.2, replace the `optimisedDecoder` semantic by:

optimisedDecoder ()	Triggers the simple optimised type decoder associated to the type `effectiveType` as conveyed in the `DecoderInit` (refer to subclause 7.2).

In subclause 8.4.7.2, add the following semantic after the `defaultDecoder` semantic:

advancedOptimisedDecoder(aType, anInteger)	Triggers the advanced optimised decoder identified by the `optimisedDecoderID` field which is associated to the type `theType` in the optimised decoder mapping.

Add the following subclause (subclause 8.4.8):

8.4.8 FragmentReference

8.4.8.1 Syntax

FragmentReference () {	**Number of bits**	**Mnemonic**
isDeferred	1	bslbf
hasSpecificFragmentReferenceFormat	1	bslbf
ReservedBits	6	bslbf
If (hasSpecificFragmentReferenceFormat == '1') {		
FragmentReferenceFormat	8	bslbf
} else {		
FragmentReferenceFormat = SupportedFragmentReferenceFormat[0]		
}		

FragmentRefLength	8+	vluimsbf8
/** FragmentRef **/		
if(FragmentReferenceFormat == "0x01") {		
URIFragmentReference()		
} else {		
ReservedBits	FragmentRef Length	
}		
/** Fragment ref end **/		
}		

8.4.8.2 Semantics

Name	*Definition*
isDeferred	A flag which signals if the fragment is deferred, and therefore can be acquired later on by the terminal. If the `isDeferred` has a value of "0", the decoder shall resolve the reference and obtain the fragment payload according to the process described in subclause 5.8.1. If the `isDeferred` has a value of "1", then the decoded value of the fragment shall include a deferred reference containing the specified fragment reference as specified in subclause 5.8.2. A node that is represented by a deferred fragment reference shall be treated by the decoder as a deferred node.
hasSpecificFragmentReferenceFormat	A flag which signals if the fragment reference format is different from the default one defined in the `DecoderInit` (see subclause 7.2): — If `hasSpecificFragmentReferenceFormat` has a value of '1', the decoder shall use the `FragmentReferenceFormat` field as indication of the fragment reference format. — If `hasSpecificFragmentReferenceFormat` has a value of '0' then the decoder shall take the fragment reference format to be that defined within the `DecoderInit` i.e. the default setting.
FragmentReferenceFormat	An 8-bits value which uniquely identifies the format of the contained fragment reference as defined in Table AMD1-1. This field shall only be present when `hasSpecificFragmentReferenceFormat` has a value of '1'.
FragmentRefLength	The number of bits that follows this field, which denotes the size of the `FragmentRef` field.

Add the following subclause (subclause 8.4.9):

8.4.9 URIFragmentReference

8.4.9.1 Syntax

URIFragmentReference () {	**Number of bits**	**Mnemonic**
href	FragmentRefLength	bslbf
}		

8.4.9.2 Semantics

Name	*Definition*
href	Defines the URI of the URI fragment reference in UTF-8 format. The field is of variable length and its actual length shall be inferred from the `FragmentRefLength` field defined within the `FragmentReference` (see subclause 8.4.8).

In subclause 8.5.2.1, add the following definition after the element transitions definition:

— *Wildcard transitions:* Wildcard transitions, when crossed, specify to the decoder that an undefined element is present in the description.

In subclause 8.5.2.1, add the following definition after the shunt transitions definition:

— *Mixed transitions*: Mixed transitions are a special kind of code transitions. Their binary code value is always equal to 1. Mixed transitions, when crossed, specify to the decoder that a string is present between the previous decoded element and the next one.

In subclause 8.5.2.2.3.1, replace the following text:

A syntax tree associated to the complex type is generated based on the “reference-free effective content particle” generated in phase 1. The syntax tree associated to the type is composed of different syntax tree nodes: element declaration nodes, group nodes and occurrence nodes. Element declaration nodes associate an element name to its type. They are leaves of the syntax tree and are derived from element declaration particles. Group nodes define a composition group (sequence, choice or all) and are derived from group particles. A group node contains only occurrence nodes. Occurrence nodes are derived from the ‘min occurs’ and ‘max occurs’ property of the particle and contain group nodes or element declaration nodes.

by

A syntax tree associated to the complex type is generated based on the “reference-free effective content particle” generated in phase 1. The syntax tree associated to the type is composed of different syntax tree nodes: element declaration nodes, wildcard nodes, group nodes and occurrence nodes. Element declaration nodes associate an element name to its type. Wildcard nodes represent element wildcards. Both element declaration and wildcard nodes are leaves of the syntax tree and are derived respectively from their element declaration and wildcard particles. Group nodes define a composition group (sequence, choice or all) and are derived from group particles. A group node contains only occurrence nodes. Occurrence nodes are derived from the 'min occurs' and 'max occurs' property of the particle and contain group nodes, element declaration or wildcard nodes.

In subclause 8.5.2.2.4, replace the following text:

Syntax tree normalization gives a unique name to every element declaration node and group node of the syntax tree. It allows to order the sibling nodes and assign a binary code to them. This code is used during the automata construction phase.

by

Syntax tree normalization gives a unique name to every element declaration node, wildcard node and group node of the syntax tree. It allows to order the sibling nodes and assign a binary code to them. This code is used during the automata construction phase.

In subclause 8.5.2.2.4, add the following text after the 3rd bullet (i.e. "Element declaration node signatures are equal to the expanded name of the element"):

— A wildcard node signature is constructed from the wildcard schema component it is associated to (see XML Schema – Part 1, Chapter 3.10.1) by the concatenation, whitespace separated, of:

  — the ":wildcard" key word

  — the "process contents" property of the wildcard schema component preceded by the character ':' (i.e. ":skip", ":strict" or ":lax"),

  — the "namespace constraint" property represented by a set of whitespace separated keywords, where

    — the 'any', 'not' and 'absent' values are respectively identified by the ":any", ":not", ":absent" keywords,

    — the ":any" or ":not" keywords are always first,

    — the namespaces and, if present the ":absent" keyword, are alphabetically sorted.

In subclause 8.5.2.2.4, add the following text at the end of the subclause:

Wildcard signature example

Given the following wildcards defined in a schema which target namespace is "http://www.mpeg7.org/example":

```
<xs:any processContents="skip"/>

<xs:any namespace="##other" processContents="lax"/>

<xs:any namespace="urn:example:namespaceB urn:example:namespaceA"/>

<xs:any namespace="##targetNamespace"/>

<xs:any namespace="##local"/>
```

The signatures of these wildcards are respectively

:wildcard :skip :any

:wildcard :lax :not http://www.mpeg7.org/example

:wildcard :strict urn:example:namespaceA urn:example:namespaceB

:wildcard :strict http://www.mpeg7.org/example

:wildcard :strict :absent

In subclause 8.5.2.2.5.1, replace the following text:

— The complex content automaton of the complex type to decode is the node automaton produced by the root node of its syntax tree

by

— The complex content automaton of the complex type to decode is the node automaton produced by the root node of its syntax tree modified according to the following rule if the content type of the complex type is 'mixed':

  — a new final state is created. The old final state is linked with the new final state by two transitions, a shunt transition and a mixed transition.

In subclause 8.5.2.2.5.2, replace the following text:

— An automaton for an element declaration node is composed of two states, a start state and a final state, and a transition between them. It is used to specify the “element name” / “type” association declared in the complex type definition. The transition is an "element transition" to which the element name of the element declaration node is associated. The target state of the transition is a "type state" to which the element type of the element declaration node is

by

An automaton for an element declaration node can have two different forms depending on the content type of the complexType in which the element is declared:

— If the content type of the complexType is not 'mixed' the automaton is composed of two states, a start state and a final state, and a transition between them. It is used to specify the “element name” / “type” association declared in the complex type definition. The transition is an "element transition" to which the element name of the element declaration node is associated. The target state of the transition is a "type state" to which the element type of the element declaration node is associated.

— If the content type of the complexType is 'mixed', the automaton is composed of three states, a start state, an intermediate and a final one. The start state is linked with the intermediate state by two transitions, a shunt transition and a mixed transition. The intermediate state is linked with the final state by an "element transition" to which the element name of the element declaration node is associated. The final state of the transition is a "type state" to which the element type of the element declaration node is associated.

In subclause 8.5.2.2.5.2, add the following figures after Figure 22:

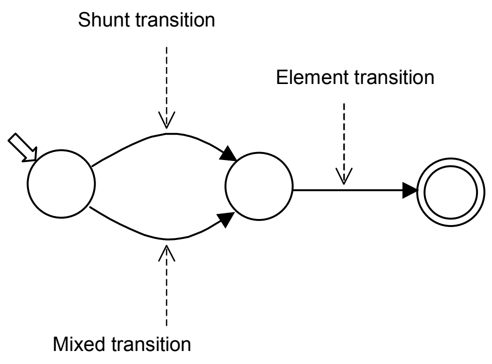


Figure AMD1-10 - Example of an element declaration node automaton in case of mixed content model

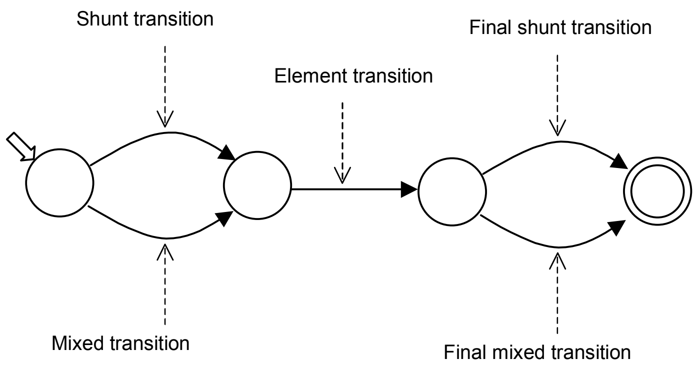


Figure AMD1-11 - Example of a complete complex content automaton in case of mixed content model

In subclause 8.5.2.2.5.2, add the following text at the end of the subclause:

Wildcard node automaton

The wildcard node automaton can have two different forms depending on the content type of the complexType in which the element is declared:

— If the content type of the complexType is not 'mixed' the automaton is composed of two states, a start state and a final state, and a wildcard transition between them.

— If the content type of the complexType is 'mixed', the automaton is composed of three states, a start state, an intermediate and a final one. The start state is linked with the intermediate state by two transitions, a shunt transition and a mixed transition. The intermediate state is linked with the final state by a wildcard transition.

Add the following subclause (subclause 8.5.2.4.4):

8.5.2.4.4 Mixed transition behavior

Mixed transitions are used to model the decoding of the strings present between two elements contained in a element of 'mixed' content type. When a mixed transition is crossed by the token, a string shall be decoded according to syntax defined in 8.4.7 where the parameters `theType` is set to the XML schema “string” datatype.

Add the following subclause (subclause 8.5.2.4.5):

8.5.2.4.5 Wildcard transition behaviour

8.5.2.4.5.1 Introduction

Wildcard transitions, when crossed, specify to the decoder that an element not known a priori is present in the description. The decoder shall execute the following decoding procedure.

8.5.2.4.5.2 AnyElementDecoding Syntax

AnyElementDecoding() {	**Number of bits**	**Mnemonic**
GlobalElementSchemaID	ceil(log2(`NumberOfSchemas + NumberOfAdditionalSchemas`))	uimsbf
SBC_Operand_Selector	ceil(log2(number_of_global elements in the schema identified by GlobalElementSchemaID))	bslbf
if (inPayloadDecoding()) {		
Element(ChildrenSchemaMode, theAnyType)		
}		
}		

8.5.2.4.5.3 AnyElementDecoding Semantics

Name	*Definition*
GlobalElementSchemaID	The schema in which the global element is defined. Its value is the index of the URI in the `SchemaURI` array defined in 7.2 (optionally extended with the list of additional schemas).
SBC_Operand_Selector	Selects one global element of the schema referenced by `GlobalElementSchemaID` using the `OperandTBC` table as specified in 7.6.5.2.3.
inPayloadDecoding()	returns true if the AnyElementDecoding procedure has been triggered from a payload decoding procedure.
Element()	see subclause 8.4.1

theAnyType	The type of the element identified by the `SBC_GlobalElement_SelectorCode` as defined in the schema identified by the `GlobalElementSchemaID`.

Replace subclause 8.5.3 content by the following subclauses 8.5.3.1, 8.5.3.2 and 8.5.3.3:

8.5.3.1 Attributes realization

The attributes of an element are decoded with the following rules:

— All the attributes definitions are collected from the super types of the complex type,

— All the attribute definitions defined as FIXED in the schema are suppressed,

— If existing, the "anyAttribute" definition is suppressed,

— All the attribute definitions are lexicographically ordered using their expanded name.

A sequence automaton is generated and used to decode the attributes. As a result, attributes are decoded by set of consecutive patterns. These patterns are composed of two components:

— an attribute flag, which defines the presence or the absence of an optional attribute,

— an attribute value decoded as defined in subclause 8.4.7.

The 1-bit attribute flag is only present for optional attributes. It is equal to 0 if the attribute is not coded or 1 if the attribute is coded. It is not present for mandatory attributes.

If an "anyAttribute" was present (and suppressed), the `AnyAttributeDecoding` procedure (see 8.5.3.2) shall be triggered after the decoding of all other attributes.

8.5.3.2 AnyAttribute Decoding

8.5.3.2.1 Syntax

AnyAttributeDecoding() {	**Number of bits**	**Mnemonic**
AnyAttributePresence	1	bslbf
If (AnyAttributePresenceFlag) {		
NumberOfAnyAttributes	5+	vluimsbf5
for (i=0;i< NumberOfAnyAttributes;i++) {		
SingleAnyAttributeDecoding()		
}		
}		
}		

8.5.3.2.2 Semantics

Name	*Definition*
AnyAttributePresence	indicates the presence of at least one attribute matched by the anyAttribute wildcard.
NumberOfAnyAttributes	indicates the number of attributes matched by the anyAttribute wildcard.

8.5.3.3 Single AnyAttribute Decoding

8.5.3.3.1 Syntax

SingleAnyAttributeDecoding() {	**Number of bits**	**Mnemonic**
GlobalAttributeSchemaID	ceil(log2(`NumberOfSchemas` + `NumberOfAdditionalSchemas`))	uimsbf
GlobalAttributeID	ceil(log2(number of attributes defined in schema `SchemaID`))	
if (inPayloadDecoding()) {		
SimpleType(theGlobalAttributeType)		
}		
}		

8.5.3.3.2 Semantics

Name	*Definition*
GlobalAttributeSchemaID	Identifies the schema in which the global attribute is defined. Its value is the index of the URI in the `SchemaURI` array defined in 7.2 (optionally extended with the list of additional schemas).
GlobalAttributeID	Identifies the global attribute amongst all the attributes defined in `GlobalAttributeSchemaID`. Global attributes code words are assigned in lexicographical order of their expanded names.
inPayloadDecoding()	returns true if the `AnyElementDecoding` procedure has been triggered from a payload decoding procedure.
SimpleType()	see subclause 8.4.7.
theGlobalAttributeType	The type of the global attribute identified by the `GlobalAttributeID` defined in the schema identified by the `GlobalAttributeSchemaID`.

Add the following clause (Clause 9):

9 Advanced optimised decoders

9.1 Overview

The simple types of an XML document are decoded by a set of datatype decoders which can be the default ones, simple or advanced optimised decoders. This subclause describes the advanced optimised decoder decoding mechanisms, more specifically:

— How the advanced optimised decoders are initialized and mapped to data types;

— How the decoder should manage the set of active advanced optimised decoders;

— How the advanced optimised decoders should behave through a logical interface.

9.2 Decoder behaviour

9.2.1 Advanced optimised decoder mapping and parameters

During the decoding of a BiM bitstream, the decoder maintains 3 tables :

— the `AdvancedOptimisedDecoderTypeURI[]` which contains all the advanced optimised decoders that can be instantiated by the decoder. This table is initialised at decoder initialisation phase and remain identical for the entire binary description stream.

— the `AdvancedOptimisedDecoderInstances[]` which contains all instances of advanced optimised decoders currently available for decoding. This table can be reinitialised before each fragment update unit according to the value of the `OptimisedDecodersReparameterization` field defined in subclause 7.4.2.

— The type-decoder mapping table where all schema types are associated to one or several optimised decoders. This table can be reinitialised before each fragment update unit according to the value of the `OptimisedDecodersReparameterization` field defined in subclause 7.4.2.

Both the `AdvancedOptimisedDecoderInstances` and the type-decoder mapping table can be redefined during binary description stream life time. The default set of advanced optimised decoder instances and mapping is defined by the `DecoderInit`. The decoder can switch to this initial configuration or define new one before the decoding of each fragment update unit as defined in subclause 7.4.2.

Advanced optimised decoder type ID	Optimised decoder type
0	Mpeg7:ZLib
1	Mpeg7:UniformQuantizer

Figure AMD1-12 - Example of the `AdvancedOptimisedDecoderTypeURI` table

Advanced optimised decoder instance ID	Advanced optimised decoder instance type	Advanced optimised decoder instance parameters
0	0 (Mpeg7:ZLib)	none
1	1 (Mpeg7:UniformQuantizer)	(0,1,8)
2	1 (Mpeg7:UniformQuantizer)	(-1,1,16)

Figure AMD1-13 - Example of the `AdvancedOptimisedDecoderInstances` table

Type identification			Advanced optimised decoder instance ID
Schema ID	Type ID	*equiv type name*	
1	1	*xsd:string*	0
1	2	*mpeg7:floatVector*	1, 2
1	3	*xsd:float*	default, 2
2	…	…	…

Figure AMD1-14 - Example of the type-decoder mapping table

9.2.2 Logical interface of an advanced optimised decoder

An advanced optimised decoder shall support the following operations:

Methods	*Definition*
readParameters()	Reads the optimised decoder parameters in the bitstream.
decode()	Decodes a value.

In addition, a contextual advanced optimised decoder shall support the following operation:

Methods	*Definition*
reset()	Reset the optimised decoder if it is contextual. This operation shall be performed after the decoding of each fragment update unit, before the decoding process enters a skippable subtree and after it exits a skippable subtree.

9.3 Advanced Optimised Decoder Initialization

9.3.1 Syntax

AdvancedOptimisedDecodersConfig () {		
*/** Definition of the optimised decoder instances table **/*		
NumOfAdvancedOptimisedDecoderInstances	8+	vluimsbf8
for (i=0; i< NumOfAdvancedOptimisedDecoderInstances; i++) {		
AdvancedOptimisedDecoderInstantiationLength	8+	
/** AdvancedOptimisedDecoderInstantiation **/		
AdvancedOptimisedDecoderInstances[i].Type	*ceil(log2(NumOfAdvancedOptimisedDecoders))*	bslbf
AdvancedOptimisedDecoderInstances[i].Params()		
nextByteBoundary()		
/** AdvancedOptimisedDecoderInstantiation end**/		
}		
*/** definition of the type-decoder mappings **/*		
NumOfMappings	8+	vluimsbf8
for (k=0; k< NumOfMappings;k++) {		
*/** the advanced optimized decoders in the mapping **/*		
PreserveDefaultDecoderInMapping	1	bslbf
Reserved	7	bslbf
NumOfAdvancedOptimisedDecodersInMapping[k]	8+	vluimsbf8
for (i=0; i< NumOfOptimisedDecodersInMapping [k]; i++) {		
AdvancedOptimisedDecoderInstanceID[k][i]	ceil(log2(NumOfAdvancedOptimisedDecoderInstances))	uimsbf
}		
*/** the types on which the optimised decoders are mapped **/*		
NumOfTypesInMapping[k]	8+	vluimsbf8
for (i=0; i< NumOfTypesInMapping[k]; i++) {		
SchemaID	ceil(log2 (NumberOfSchemas + NumberOfAdditionalSchemas))	uimsbf
TypeIdentificationCode[k][i]	8+	vluimsbf8
}		
}		
}		

9.3.2 Semantics

Name	*Definition*
NumOfAdvancedOptimisedDecoderInstances	Indicates the total number of optimised decoder instances present in the optimised decoder instances table.
AdvancedOptimisedDecoderInstantiationLength	Indicates the length in bytes of the decoder instantiation excluding the length.
AdvancedOptimisedDecoderInstances[i].Type	Indicates the advanced optimised decoder type that is instantiated. It refers to a type identified in the `AdvancedOptimisedDecoderTypeURI` table.
AdvancedOptimisedDecoderInstances[i].Params	Parameters associated to a optimised decoder. This field is dependent of the optimised decoder and can have a null size.
NumOfMappings	Indicates the number of mappings between sets of types and sets of optimised decoders.
PreserveDefaultDecoderInMapping	Indicates that the the types contained in the mapping referred by the index k can be decoded by their default decoder. This default decoder is always referred by the index 0 in the set of optimised decoders associated to each data type.
NumOfAdvancedOptimisedDecodersInMapping [k]	Indicates the number of optimised decoders that are associated to teh data types contained in the mapping referred by the index k.
AdvancedOptimisedDecoderInstanceID[k][j]	Indicates the instance to be added to the mapping referred by the index k. It is an index to the `AdvancedOptimisedDecoderInstances[]` table.
NumOfTypesInMapping [k]	Indicates the number of data types which accept the optimised decoders defined in the mapping.
SchemaID	Identifies the schema from the list of `schemaURI`s transmitted in the `DecoderInit` (optionally extended by a list of additional schemas) in which the type subject to the mapping is defined.
TypeIdentificationCode[k][j]	Selects one data type from the set of all data types contained in the schema with index `SchemaIndex`. The syntax and semantics of `TypeIdentificationCode[k][j]` is the same as of the type identification code defined in subclause 7.6.5.4 except that here it is represented using vluimsbf8. The `TypeIdentificationCode[k][j]` assumes the “anyType” as base type. The advanced optimised decoder instances present in the mapping are associated to this type and all the subtypes of this type defined in the schemaID. For instance if the type "string" is mapped with an optimised decoder, all inherited types from string will be bound to the same instance of the optimised decoder. In order to bind an inherited subtype with another optimised decoder, this has to be made explicit in another mapping. Once again, all subtypes of this subtype, will refer to the latter mapping.

9.4 Advanced Optimised Decoder Classification scheme

The advanced optimised decoders defined in this specification are uniquely identified by the following classification scheme:

```
<ClassificationScheme
       uri="urn:mpeg:mpeg7:systems:SystemsAdvancedOptimisedDecodersCS:2004">
   <Term termID="1">
       <Name xml:lang="en">Zlib</Name>
       <Definition   xml:lang="en">A   contextual   advanced   optimised   decoder
dedicated to the compression of strings</Definition>
   </Term>
   <Term termID="2">
       <Name xml:lang="en">UniformQuantizer</Name>
       <Definition xml:lang="en">An advanced optimised decoder to compress large
series    of    floating    point    numbers    using    uniform    quantization
technique</Definition>
   </Term>
   <Term termID="3">
       <Name xml:lang="en">NonUniformQuantizer</Name>
       <Definition xml:lang="en"> An advanced optimised decoder to compress large
series  of  floating  point  numbers  using  non-uniform  quantization  technique
</Definition>
   </Term>
</ClassificationScheme>
```

9.5 UniformQuantizer advanced optimised decoder

9.5.1 Overview of scalar quantization (informative)

A scalar quantization is a mathematical operation which maps a set of consecutive, non overlapping intervals of real values (a partition) to a set of integers (the labels) where each interval is labeled with a unique number. Therefore a quantization operation is always associated with loss of data. For this reason the quantization operation is also referred to as a kind of lossy compression in contrast to lossless compression such as e.g. zip, run length coding, Huffmann codes, and so on. However, this loss can be made arbitrarily small by choosing an appropriate number of bits to encode the integer value (representing the interval label) and by the choice of the characteristics of the quantizier. Depending on the mapping function, two types of quantizers are distinguished, namely uniform and non-uniform quantizers. In fact uniform quantizers can be considered as a special case of non-uniform quantizers.

The mapping is the process of finding the quantization interval into which the real value to be quantized fits. This interval label is then encoded with a certain limited number of bits. The reverse operation, called inverse quantization or reconstruction, restitutes a real value according to the interval label. These processes are controlled by two parameter sets: the first gives the boundaries of the non-overlapping intervals (a partition) and the second one the reconstruction values, called output levels. In order to yield minimal quantization error the probability density function (PDF) of the signal to be quantized must either be known or must be estimated based on a representative set of data. With uniform quantizers the error becomes minimal only if the input signal exhibits an even probability distribution over the numerical range to be covered.

9.5.2 Decoders parameters

9.5.2.1 syntax

UniformQuantizerParameters() {	**Number of bits**	**Mnemonic**
sgn(v_{min})	1	bslbf
abs(v_{min})	5+	vluimsbf5
sgn(v_{max})	1	bslbf
abs(v_{max})	5+	vluimsbf5
nbits-1	5	uimsbf
}		

9.5.2.2 Semantics

Name	*Definition*
sgn(v_{min})	Sign of v_{min}. If v_{min} is positive ; 0, else 1.
abs(v_{min})	Absolute value of v_{min}.
sgn(v_{max})	Sign of v_{max}. If v_{max} is positive ; 0, else 1.
abs(v_{max})	Absolute value of v_{max}.
nbits	Number of bits used to encode quantized numbers. `nbits` can take the values from 1 to 32. But the field `nbits`-1 will be written.

9.5.3 Decoding process

9.5.3.1 Syntax

UniformQuantizerDecode(parameters) {	**Number of bits**	**Mnemonic**
vq	`nbits`	uimsbf
}		

9.5.3.1.1 Semantics

Name	*Definition*
vq	An `nbits` bits field defining the quantized value v_q. The restituted decoded value $\overline{v}$ is : $$\overline{v} = v_{\min} + v_q \frac{v_{\max} - v_{\min}}{2^{nbits} - 1}$$

9.6 NonUniformQuantizer optimized decoder

9.6.1 Overview of scalar quantization (informative)

A scalar quantization is a mathematical operation which maps a set of consecutive, non overlapping intervals of real values (a partition) to a set of integers (the labels) where each interval is labelled with a unique number. Therefore a quantization operation is always associated with loss of data. For this reason the quantization operation is also referred to as a lossy compression in contrast to lossless compression such as e.g. zip, run length coding, Huffmann codes, and so on. However, this loss can be made arbitrarily small by choosing an appropriate number of bits to encode the integer value (representing the interval label) and by the choice of the characteristics of the quantizer. Depending on this characteristics, two types of quantizers are distinguished, namely uniform and non-uniform quantizers. In fact, uniform quantizers can be considered as a special case of non-uniform quantizers.

The forward quantization process consists of finding the quantization interval into which the real value to be quantized fits. This interval label is then encoded with a certain limited number of bits. The reverse operation, called inverse quantization or reconstruction, restitutes a real value according to the interval label. These processes are controlled by two parameter sets: the first gives the boundaries of the non-overlapping intervals (partitions) and the second one the reconstruction values, called output levels. In order to yield minimal quantization error, the probability density function (PDF) of the signal to be quantized must either be known or estimated from a representative set of data.

The general scheme for the non-uniform quantization is defined by the explicit interval boundaries and the corresponding output levels. The algorithm for the quantization part comprises of finding the interval in which the input value resides. The reconstruction is performed through a table lookup where the output level is retrieved from the quantized value which serves as a table index. It is known that this form of quantization yields the minimum error when the sets of parameters (interval borders and output levels) are derived from the signal’s PDF. One algorithm for finding such parameter sets (characteristic) is the well known Lloyd-Max quantizer. If the signal to be quantized is a sequence of vectors (e.g. containing data derived from different frequency bands in each vector component), each vector component may have different statistical properties and thus require different quantizer characteristics for optimum representation.

9.6.2 Decoder parameters

9.6.2.1 Syntax

NonUniformQuantizerParameters() {	**Number of bits**	**Mnemonic**
nBitsData-1	5+	vluimsbf5
nCharacteristics	5	uimsbf
/** uniformQuantizerParameters **/**		
sgn(vmin)	1	bslbf
abs(vmin)	5+	vluimsbf5
sgn(vmax)	1	bslbf
abs(vmax)	5+	vluimsbf5
nbitsCharacteristics-1	5	uimsbf
/** uniformQuantizerParameters end **/		
for (j=0; j<nCharacteristics ; j++) {		
for (k=0; k<2$^{\wedge nbitsData}$; k++) {		
quantMatrix[j][k]	nbitsCharacteristics	uimsbf
}		
}		
}		

9.6.2.2 Semantics

Name	*Definition*
nBitsData	Indicates the number of bits used for the encoding of the data.
nCharacteristics	Indicates the total number of individual quantizer characteristics that are defined for representing the data. This number is usually chosen as 1 (for a series of simple scalar values) or as the number of vector components (for a series of vectors).
uniformQuantizerParameters	For each of the quantizer characteristics, the corresponding set of output levels is transmitted by means of a uniformly quantized representation. The uniformQuantizerParameters hold the definition information for this representation.
sgn(v_{min})	Sign of v_{min}. If v_{min} is positive ; 0, else 1.
abs(v_{min})	Absolute value of v_{min}.
sgn(v_{max})	Sign of v_{max}. If v_{max} is positive ; 0, else 1.
abs(v_{max})	Absolute value of v_{max}.
nBitsCharacteristics	Number of bits used to encode quantized numbers.
quantMatrix[j][k]	This Matrix holds the output levels (index k) for each of the quantizer characteristics (index j) for the binary payload data.

9.6.3 Decoding process

9.6.3.1 Syntax

NonUniformQuantizerDecode(parameters) {	*Number of bits*	*Mnemonic*
vqIndex	nBitsData	uimsbf
}		

9.6.3.2 Semantics

Name	*Definition*
vqIndex	The restituted decoded value $\bar{v}$ is computed in the following way: $$\bar{v}=v_{\min}+v_q\frac{v_{\max}-v_{\min}}{2^{nbits}-1}$$ Where v_q = quantMatrix[c][vqIndex] The quantizer characteristics c used to represent / reconstruct each transmitted value is determined by the order of the transmitted values. The first value uses characteristics c=0, the second value c=1 and so forth, incrementing by one and falling back to 0, once c=nCharacteristics has been reached (i.e. “increment by one, modulo nCharacteristics).

9.7 Zlib advanced optimised decoder

9.7.1 Overview

The ZLib decoder is a contextual optimised decoder which doesn't have any specific parameters. It uses the Zlib encoding scheme as defined in RFC 1950.

9.7.2 Decoding process

9.7.2.1 Syntax

String ZLibDecoder() {	**Number of bits**	**Mnemonic**
resultString=""		
tempChar = GiveNextCharInBuffer();		
while (tempChar != separatorChar){		
resultString = concat(resultString, tempChar)		
tempChar = GiveNextCharInBuffer()		
}		
return resultString		
}		

Char GiveNextCharInBuffer() {	**Number of bits**	**Mnemonic**
If isEmpty (charsBuffer){		
ZLibChunkLength	8+	vluimsbf8
ZLibChunk	8* ZLibChunkLength	bslbf
charsBuffer = inflate(ZLibChunk)		
}		
return nextChar(charsBuffer)		
}		

9.7.2.2 Semantics

If the ZLib codec is asked to deliver a string to the BiM framework, either:

— the current ZLib buffer is empty : the ZLib decoder reads a compressed chunk in the bitstream, and decompresses it into the current ZLib buffer,

— the buffer is not empty : the ZLib decoders delivers the first string of the buffer.

Strings are separated by a separatorChar. If a string is encoded over two or more chunks, the decoder decompresses all the needed chunks and returns the concatenation of all characters extracted from the buffers until the separator character.

In order to obtain the next string, the decoder reads the charsBuffer until it gets a separatorChar. If the charsBuffer is totally consumed before reaching a separatorChar, the charsBuffer is refilled by decompressing the ZLib compressed chunk available in the bitstream.

Name	*Definition*
resultString	This is a field representing the string expected.
tempChar	This is a field representing the character read in the charsBuffer.
separatorChar	This is a constant representing the separatorChar. It is equal to 0x00.
ZLibChunkLength	Indicates the size in bytes of the ZLibChunk. A value of zero is forbidden.
ZLibChunk	This is an UTF-8 representation of a compressed list of characters. The compression algorithm used is ZLib, in default compression mode.
inflate(buffer)	This function is a part of the ZLib API. It decompresses a ZLib buffer.
nextChar(charsBuffer)	This function returns the first character of the charsBuffer and removes it from the charsBuffer.
isEmpty (charsBuffer)	This function returns true if the charsBuffer is empty, otherwise false.
concat(aString, aCharacter)	This function returns the concatenation of aString and aCharacter.
charsBuffer	This is a local buffer of decompressed characters. It contains a list of strings separated by a separatorChar.

9.7.3 Encoding process (informative)

At the encoding phase, the fixed-size buffer is fed with strings. When the buffer is full, it is compressed and the resulting compressed chunk of data is placed in the expected position of the first string compressed in this buffer. Figure AMD1-15 represents a regular BiM binary stream, without the ZLib codec. Strings (in gray) are dispatched along the entire bitstream. Figure AMD1-16 represents only the two phases of the ZLib codec encoding process : strings are first gathered on the fly into a fixed-size buffer, and then, this buffer is compressed by the ZLib algorithm, into a compressed chunk, which may contains several strings. The location of the compressed chunk ensures that during the decoding process, when a string is required, either the ZLib codecs will read a compressed chunk, decompress it and delivers the string from the decompressed buffer or if its decompressed buffer isn't empty, it delivers the string from its decompressed buffer.

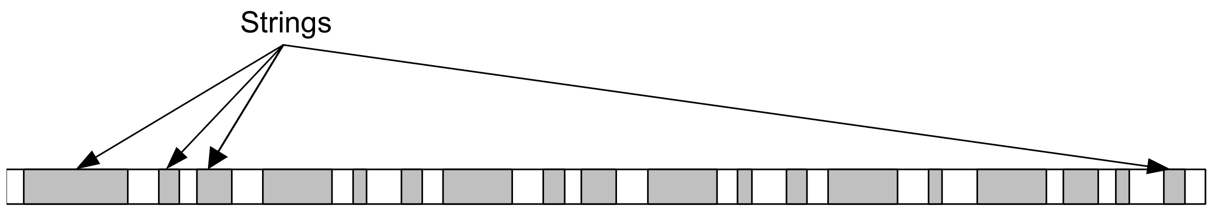


Figure AMD1-15 - Regular BiM bitstream

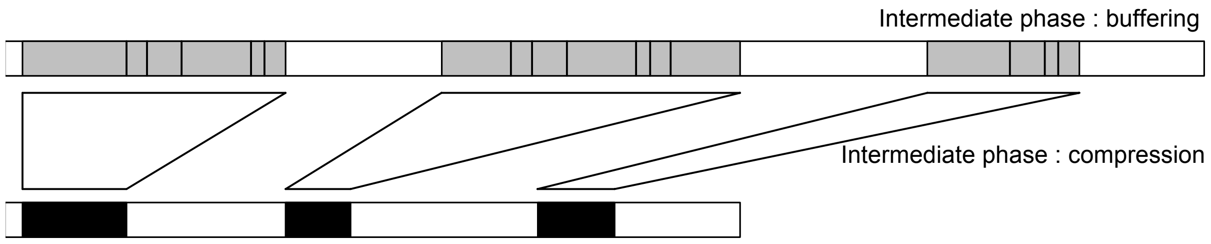

Intermediate phase : buffering
Intermediate phase : compression

Figure AMD1-16 - BiM bitstream using Zlib encoded value

In the bibliography annex, add the following reference:

ZLIB Compressed Data Format Specification Version 3.3, RFC 1950, P. Deutsch, J. Gailly, May 1996, http://www.ietf.org/rfc/rfc1950.txt

AMENDMENT 2: Fast access extension

This document preserves the sectioning of ISO/IEC 15938-1. The text and figures given below are additions and/or modifications to those corresponding sections in ISO/IEC 15938-1. All figures and tables shall be renumbered due to the addition of several figures and tables.

Add the following definitions to subclause 3.2 (keep alphabetical order), then renumber all definitions in subclause 3.2:

path index key
value representing the path to the element to be indexed/located, and the relative path to the fields to be keyed/searched.

value index key
set of encoded field values to be keyed/searched.

index stream
set of Index Access Units which together form the whole of the indexing data,

index decoder init
initialisation data for an index stream.

index access unit
index access unit header and associated structures forming a logical unit of access.

index access unit header
list of structures contained within this Index Access Unit

path index
structure allowing path index key to value index reference lookup.

value index
structure allowing value index key to value sub index reference lookup.

value sub-index
structure allowing value index key to BiM stream reference lookup.

node reference
reference from one node to another within a list or B-Tree structure.

data repository reference
reference to data entry within the binary or string data repository structures.

BiM stream reference
reference to a BiM encoded fragment within a BiM stream.

local access unit
default Access Unit associated with a Path Index.

local BiM stream reference
reference to a BiM encoded fragment contained within the local access unit.

remote BiM stream reference
reference to a BiM encoded fragment contained within the BiM stream, where the access unit is specified by an access unit ID.

value index reference
reference to a value index structure.

value sub-index reference
reference to a sub value index structure.

position codes reference
reference to a position codes entry within a position codes structure.

position code
location of an XML element within its parent element.

position codes
set of position code values for all elements within a context path.

BTree
binary decision tree, where each node can have multiple keys.

BTree order
specifies the maximum number of child nodes of a node in a BTree.

indexed element
XML element to which an index refers.

BiM stream reference format
specifies the format of the BiM stream reference.

value encoding
specifies how data has been encoded in value index key.

Add the following subclause 5.9.1:

5.9.1 General Description

Using the ISO/IEC 15938-1 index encoding, only fragments of the description that are of immediate interest to the terminal can be selectively acquired and combined with the current description tree. The terminal can search the index information to determine which fragments contain a node at a given location which has a related node with a given value within the description. Additionally the terminal may search for fragments containing nodes which have related node values falling within a given range.

The index information can also be compiled to allow the terminal to search for fragments containing nodes with multiple given related node locations, and respective values, within the description. This can allow the terminal to perform searches with multiple conditions, without needing to consolidate multiple result sets. As the indexing stream is optional a stream may consist of either

A DecoderInit and a description stream

A DecoderInit, an IndexDecoderInit, an index stream, and a description stream.

Before an index stream can be queried both the DecoderInit for the BiM stream to which the index stream belongs, and the IndexDecoderInit for the index stream must be acquired. However acquiring fragments from the description stream, without querying the index stream, only requires the DecoderInit to be acquired.

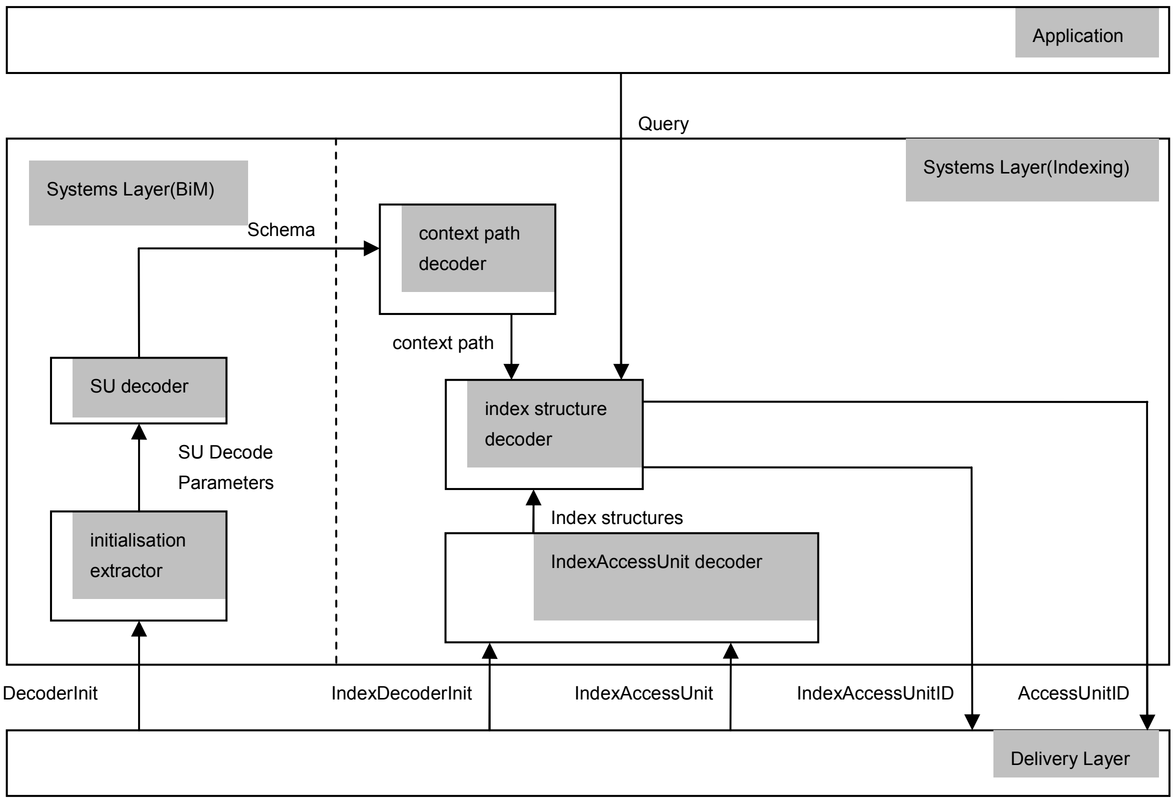


Figure Amd2.1 — Indexing Enabled Terminal Architecture Extension.

All components of Systems Layer(Indexing) section are non-normative

The Terminal Architecture for a BiM enabled terminal may be extended to support indexing, as shown in Figure Amd2.1. Figure Amd2.1 shows only the extensions to the Terminal Architecture, and not the complete architecture. The components shown in the Systems Layer(BiM) section of Figure Amd2.1 are the existing components of the standard Terminal Architecture.

Add the following subclause 5.9.2:

5.9.2 Options for multi criteria query

There are two main methods of querying for fragments when there are multiple criteria, multi-value indexing, and multi-stage indexing. These different methods are distinct and offer two complementary optimizations.

Multi-value indexing allows the whole data set to be indexed in a very compact manner. The size of the index stream and the number of comparisons of values is minimized, allowing the index to perform a multi criteria query using the smallest amount of index data and queries possible. Partitioning of indices with larger data sets allows maximum IndexAccessUnit sizes to be imposed. This is useful when the underlying transport layer has an imposed, or preferred size for a unit of access, as might be the case with network packets, or transport layer data buffers. It is usual for several IndexAccessUnits to be required to complete a query, hence the IndexAccessUnit is not independent.

Multi-stage indexing allows the data set to be sectioned into multiple smaller index stream segments. This increases both the overall size of the index stream and the number of value comparisons by a moderate amount. However, multi-stage indexing can facilitate more efficient use of resources in client terminal devices where caching of stream index data is necessary but the size of the stream index data prohibits caching all of it. Caching is desirable where either the index stream is not always available or the acquisition time of Index Access Units from the index stream is significant. In a multi-stage index the data size of each independent segment of the index stream is reduced, allowing a whole segment of the index to be cached from the index stream into memory and searched independent of the Index Stream. This is often a better optimisation than attempting to cache a portion of an index arranged as a single segment.

Multi-value indexing is intended to be used in situations where efficient index size is a priority and there are no significant restraints imposed by client terminal resources. There are two formats of multi-value indexing, composite value, and hierarchical single value indexing. In composite value indexing the values are stored as an N column table, where N is the number of related values. In hierarchical single value indexing the values are stored as an N level tree, with the tree's nodes containing 1 column tables (Lists). The composite value index is intended for use where there is unlikely to be multiple instances of the related node with the same value, whilst the hierarchical single value indexing is intended for use where there are many instances of the related node with the same value in the BiM fragments to be indexed. It is important to take care to choose the order of the related nodes carefully, as the related node with most common values placed at the highest level of the hierarchy will usually produce the smallest index stream and the minimum number of value compares when querying.

Multi-stage indexing is intended for use in situations where client terminals with limited resources must access a very large amount of BiM fragments by index, and for which there are a small number of search criteria common for most queries. The common criteria can be used to segment the index stream into a collection of index stream segments, each segment then being accessed and searched independently of other segments. This allows searches to start on an initial segment stored in a cache and then progress to the relevant follow-on segment which, if not cached, must be acquired from the index stream. The search is therefore completed with a minimum number of acquisitions and without requiring the whole multi-stage index to be cached.

Note that the method of caching and cache maintenance is implementation dependent, and not defined in this specification.

Note - Another situation where multi stage indexing can be used, is when consolidating multiple independent index streams. This may be the case if there are multiple providers of BiM fragments each with an associated index stream. Consolidation of the index streams can be achieved easily by modifying each index stream to be a second stage segment. A first stage index would then be created to associate a provider to a second stage segment within the index stream.

Change the following sentence at the end of subclause 7.1 as indicated:

Several other coding modes are initialised in the DecoderInit related to the features used by the binary description stream: the insertion of elements, the transmission of schema information, references to fragments and a fixed length context path.

And add the following sentence at the end of subclause 7.1:

The fixed length context path mechanism provides a simplified addressing of nodes for usage scenarios where only a limited number of nodes need to be addressed. This is done by a table that uniquely maps fixed length codes to full context paths.

In subclause 7.2.2, insert grey marked rows at the position indicated:

If (! NoAdvancedFeatures) {		
AdvancedFeatureFlags_Length	8+	vluimsbf8
/** FeatureFlags **/		
InsertFlag	1	bslbf
AdvancedOptimisedDecodersFlag	1	bslbf
AdditionalSchemaFlag	1	bslbf
AdditionalSchemaUpdatesOnlyFlag	1	bslbf
FragmentReferenceFlag	1	bslbf
MPCOnlyFlag	1	bslbf
HierarchyBasedSubstitutionCodingFlag	1	bslbf
ContextPathTableFlag	1	bslbf
ReservedBitsZero	FeatureFlags_Length*8-8	bslbf
/** FeatureFlags end **/		

If (**ContextPathTableFlag**) {		
ContextPathTable()		
}		
*<u>/** FUUConfig - Advanced optimised decoder framework **/</u>*		
If (AdvancedOptimisedDecodersFlag) {		

ContextPathTable {		
ContextPathTable_Length	8+	vluimsbf8
ContextPathCode_Length	8+	vluimsbf8
NumberOfContextPaths	8+	vluimsbf8
CompleteContextPathTable	1	bslbf
for(i=0;i<NumberOfContextPaths;i++){		
ContextPath_Length[i]	5+	vluimsbf5
ContextPath()[i]	ContextPath_Length[i]	
If(!CompleteContextPathTable){		
ContextPathCode[i]	ContextPathCode_Length	bslbf
}		
}		
nextByteBoundary()		
}		

In subclause 7.2.3, insert,

ContextPathTableFlag	Signals the presence of a context path table in the decoder init.

ContextPathTable_Length	Defines the number of bytes used for the indication of the ContextPathTable. Note – This length provides a simple framework to skip the table.
ContextPathCode_Length	Signals the length of the context path codes in number of bits.
NumberOfContextPaths	Signals the number of ContextPaths contained in the ContextPathTable.
CompleteContextPathTable	Signals if the ContextPathTable is complete and ordered according to the assignment of ContextPathCodes. If CompleteContextPathTable is set to ‘1’ the ContextPathCodes are assigned in the order the ContextPaths are specified in the ContextPathTable starting from ‘1’. If CompleteContextPathTable is set to ‘0’ the ContextPathCodes are assigned explicitly. The ContextPathCode ‘0’ is reserved.
ContextPath_Length	Signals the number of bits used for the following ContextPath[i]()
ContextPath[i]()	Signals the ContextPath as specified in subclause 7.6.2 with the following restrictions: - ContextModeCode is set to ‘001’ - PositionCode() is an empty bitfield
ContextPathCode[i]	Signals the ContextPathCode of .ContextPath[i]

In subclause 7.6.2, insert grey marked rows at the position indicated:

FragmentUpdateContext () {	**Number of bits**	**Mnemonic**
SchemaID	ceil(log2(NumberOfSchemas))	uimsbf
ContextModeCode	3	bslbf
If (ContextModeCode==’101’){		
ContextPathCode	ContextPathCode_Length	bslbf
for (i=0; i < TBC_Counter(ContextPathCode); i++) {		
PositionCode()		
}		
}		
else {		
ContextPath()		
}		
}		

In subclause 7.6.4, insert grey marked rows at the position indicated:

Code	**Context Mode**
…	
101	Navigate in “Absolute addressing mode” from the selector node to the node specified by the Context Path signaled by the ContextPathCode.
110-111	Reserved

Add the following clause 10:

10 Indexing

10.1 Overview

The index is provided to support fast random access into a BiM stream. The index allows access to the FUU’s, within the BiM stream, containing a desired XML node, either element or attribute, based on the specification of one or more related node, element or attribute, values. For instance, determining FUU’s which contain “Car” elements, who’s “Color” attribute is “Red”. The search specification uses the context path of the desired XML node, and the relative context paths of the related nodes, this specification is termed the PathIndexKey. This allows a flexible PathIndexKey, which is capable of indexing complex type elements, simple type elements, and attributes according to one or more criteria.

This section gives a general overview of the structure and functionality of the index, and how it is accessed to arrive at a resultant set of FUU’s matching a given search criteria.

The index is composed of two parts, the Path Index, and the Value Index. The Path Index allows the particular Value Index relating to the PathIndexKey, specified in the search criteria, to be located. The Value Index allows the set of BiM stream references which contain the desired XML node, with values for the PathIndexKey's related nodes meeting the search criteria. The indexing technologies specified here, allow a Value Index to be partitioned into smaller Value Sub Index structures. This partitioning allows a Value Index to contain an unlimited number of entries without increasing the resource requirements, in particular working memory, of the client.

The data structures in this specification allow for the Value Index to be created with the values for the PathIndexKey's related nodes, to be represented as a hierarchical index, or as a consolidated index. The hierarchical index can offer much smaller index structures, and faster searches, if the PathIndexKey's related nodes values contain significant repetition, as this allows repeated values to be grouped together and entered into the Value Index only once. If there is not significant repetition, then the Value Index can be represented in a consolidated index, which allows all of the values for the PathIndexKey's related nodes to be consolidated and represented within a single index structure. A search on the Value Index will result in a set of zero or more BiM stream references. The BiM Stream references locate the FUUs which contain the desired XML nodes with related node values satisfying the search criteria. The BiM Stream references can optionally specify the position of the desired XML node within each FUU, in addition to the FUU reference itself.

To demonstrate the searching process, consider the example where a simple, single criteria, search is being made for a "Picture" node, whose related node, "Subject", has a value of "Winter"

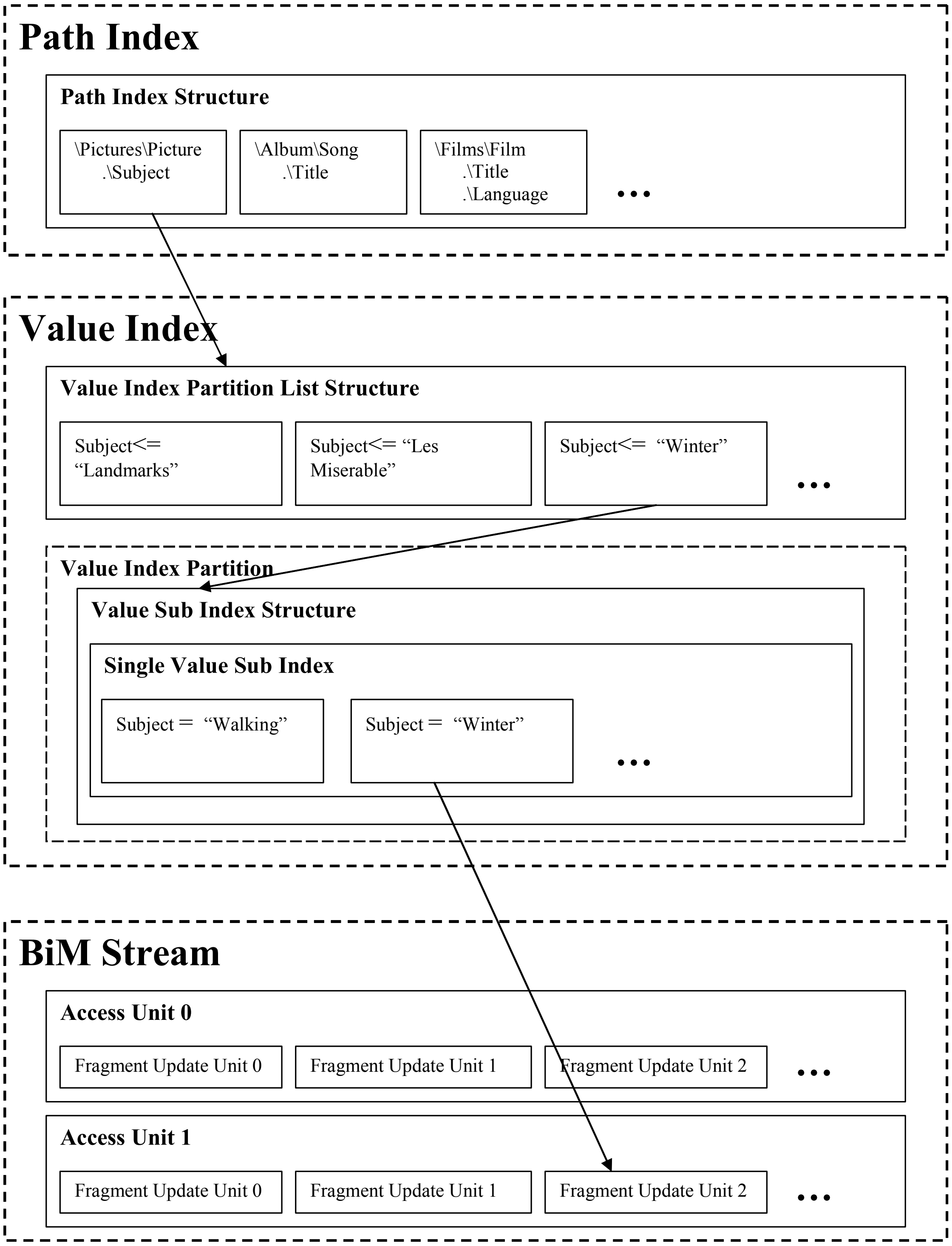


Figure Amd2.2 — Block diagram of the BiM Index structure

The Path Index is first searched to locate the relevant Value Index. This is achieved by scanning the Path Index structure. Once the Value Index has been determined, its Value Index Partition List structure is used to determine which of the Value Index partitions will contain the "Subject" being searched for, in this case "Winter". Now the Value Sub Index structure can be accessed, which in this case contains a Single Value Index, for the values of "Subject" in this partition. These values are then searched to determine the BiM Stream references which match the search criteria.

The next example explains how the Compound Value Index can be used to search for the “Film” element with two related elements “Title” and “Language”

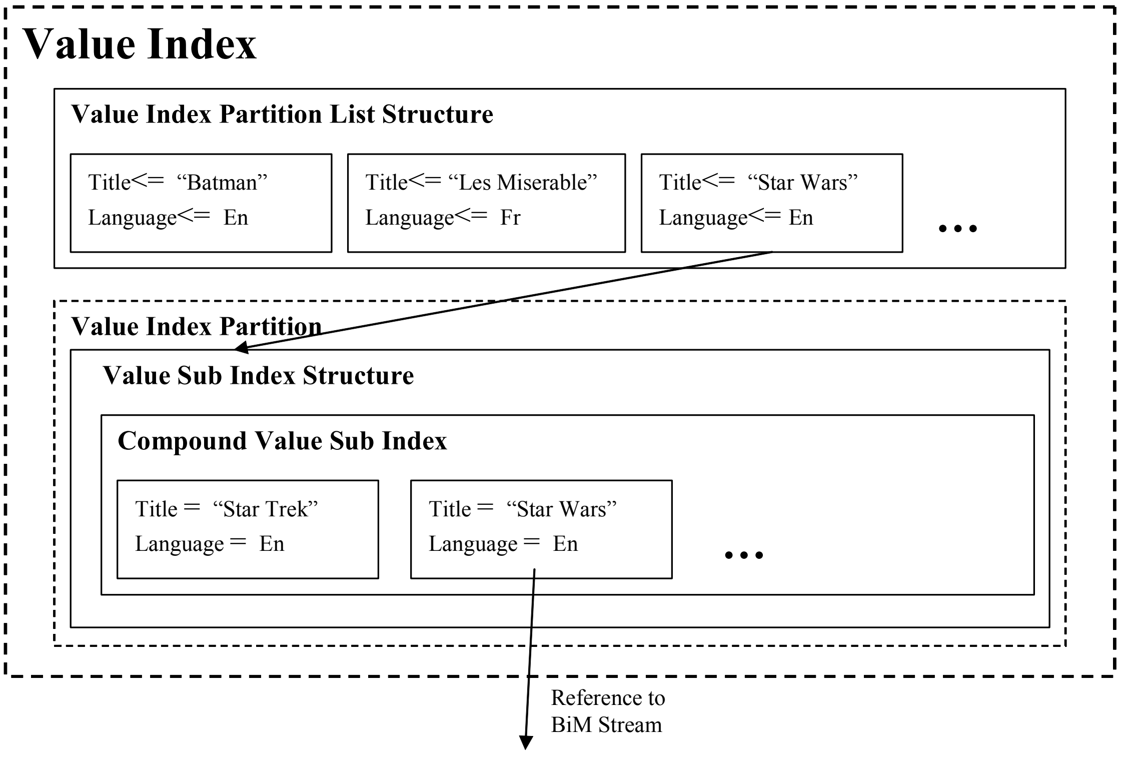


Figure Amd2.3 — Block diagram of the compound value index structure

The process is the same as in the first example, except that the Compound Value Index contains values for both related nodes.

The final example, is the same as the previous example, except that a hierarchy of Single Value Index entries has been used rather then the Compound Value Index.

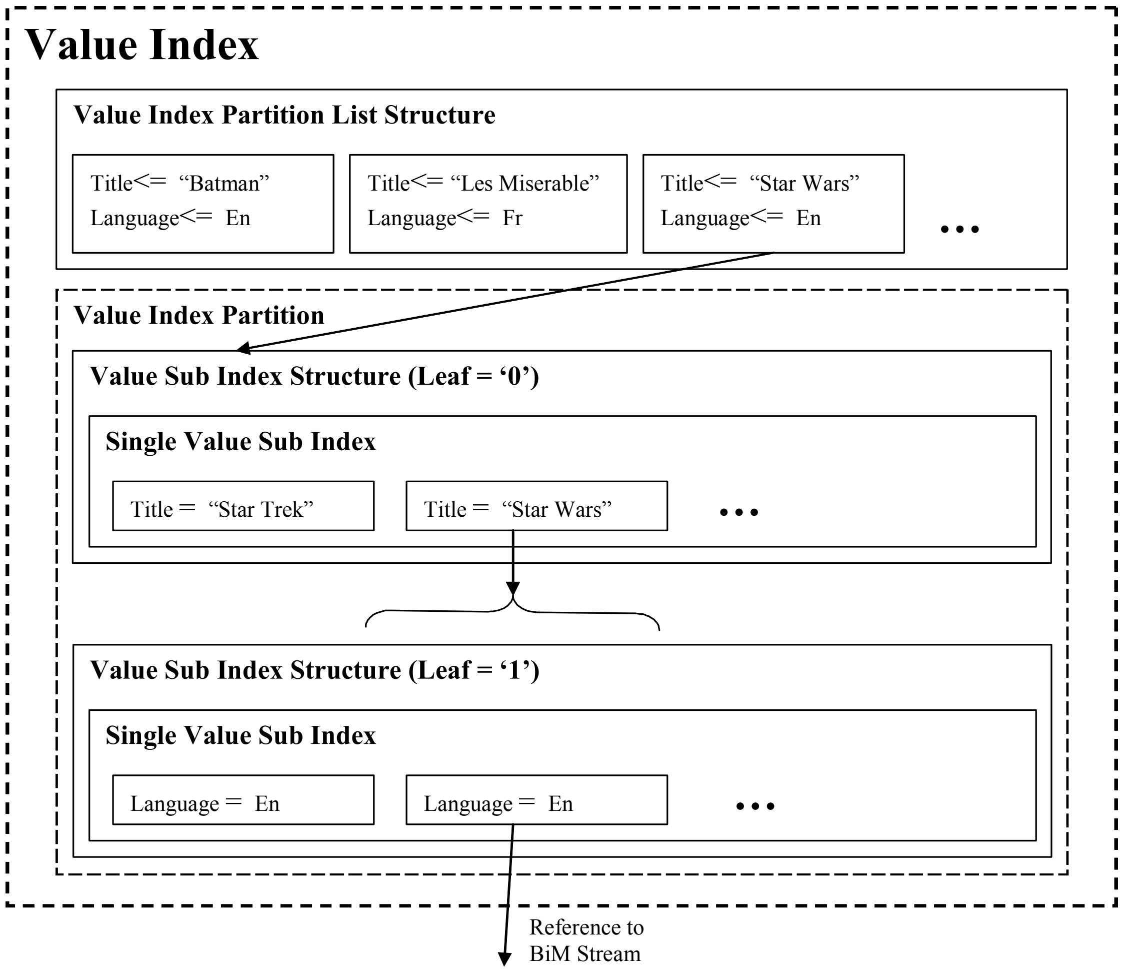


Figure Amd2.4 — Block diagram of the hierarchical single field index structure

This shows that the first related node, "Title" is searched first, but instead returning the BiM stream references, it returns a range to search in the child Sub Value Index. The child is then searched for the correct value for the "Language" element to determine the BiM stream references.

10.2 Characteristics of the delivery layer

The delivery layer is an abstraction that includes functionalities for the synchronization, framing and multiplexing of indexing streams with other data streams. Index streams may be delivered independently or together with the associated Description Stream. No specific delivery layer is specified or mandated by ISO/IEC 15938.

A delivery layer (DL) suitable for conveying ISO/IEC 15938 index streams shall have the following properties in addition to the properties defined for decoding of description streams:

— The DL shall provide a mechanism to communicate an index stream from its producer to the terminal.

— The DL shall provide a mechanism by which a random access point to the index stream can be identified.

— The DL shall provide a suitable random access mechanism allowing access to an IndexAccessUnit by use of a 16 bit IndexAccessUnit identifier.

— The DL shall provide a default 16 bit IndexAccessUnit identifier for each PathIndex in the index stream.

— The DL shall provide a mechanism by which a random access point to the description stream can be identified.

— The DL shall provide a suitable random access mechanism allowing access to an Access Unit by use of a 16 bit Access Unit identifier.

— The DL shall provide delineation of the index access units within the index stream, i.e., IndexAccessUnit boundaries shall be preserved end-to-end.

— The DL shall preserve the order of IndexAccessUnits on delivery to the terminal, if the producer of the index stream has established such an order.

— The DL shall provide either error-free index access units to the terminal or an indication that an error occurred.

— The DL shall provide a means to deliver the `DecoderInit` information (see subclauses 6.2 and 7.2) and the IndexDecoderInit information (see subclause 10.3 to the terminal before any index access unit decoding occurs.

— The DL shall provide signalling of the association of an index stream to a description stream.

— If an application requires index access units to be of equal or restricted lengths, it shall be the responsibility of the DL to provide that functionality transparently to the systems layer.

Note - The 16 bit Access Unit ID is independent of the 16 bit Index access Unit ID.

10.3 IndexDecoderInit

10.3.1 Overview

The Index`DecoderInit` specified in this subclause is used to configure parameters required for the decoding of the index access units. There is only one Index`DecoderInit` associated with one index stream.

Main components of the Index`DecoderInit` are an indication of the profile and level of the associated index stream.

Both the DecoderInit for the description stream and the IndexDecoderInit for the index stream must be acquired prior to decoding Index Access Units.

10.3.2 Syntax

IndexDecoderInit () {	**Number of bits**	**Mnemonic**
SystemsIndexProfileLevelIndication	8+	vluimsbf8
}		

10.3.3 Semantics

Name	*Definition*
SystemsIndexProfileLevelIndication	Indicates the profile and level as defined in ISO/IEC 15938-1 to which the description stream conforms. Table Amd2.1 lists the indices and the corresponding profile and level.

Table Amd2.1 — Index Table for SystemsIndexProfileLevelIndication

Index	*Systems Profile and Level*
0	no profile specified
1 – 127	Reserved for ISO Use

10.4 Index Access Unit

10.4.1 Overview

The Index Access Unit id used to collect multiple structures together into a logical unit. For instance all the data for structures belonging to a single Index Access Unit, must be contained in the Data Repository structure within the same Index Access Unit. The grouping of structure into an Index Access Unit would normally be determined by what is simplest and logical for the encoding process, but may also be limited by the underlying transport.

It is the responsibility of the transport layer to provide the retrieval of Index Access Units from their 16 bit Index Access unit id.

Before the Index Access Unit can be interpreted the Decoder Init and the Index Decoder Init must be acquired.

10.4.2 Syntax

IndexAccessUnit(){	**No. of Bits**	**Mnemonic**
IndexAccessUnitHeader(){		
num_structures	8	uimsbf
for(j = 0; j < num_structures; j++){		
structure_type	8	uimsbf
structure_id	8	uimsbf
structrue_ptr	24	uimsbf
structure_length	24	uimsbf
}		
}		
for(j = 0; j < **num_structures**; j++) {		
structure[j]()		
}		
}		

10.4.3 Semantics

Name	*Definition*
num_structures	number of structures within Index Access Unit.
structure_type	identifies type of structure, such as data repository. See subclause 10.4.4.
structure_id	stores index id/sub index id, context dependent on structure_type. see subclause 10.4.5.
structure_ptr	bytes offset from start of Index Access Unit.
structure_length	length in bytes of structure

10.4.4 structure_type assignments

Value	*Description*
0x00	Index Configuration Structure (see subclause 10.5)
0x01	Reserved
0x02	Data Repository Structure (see subclause 10.6)
0x03	Path Index Structure (see subclause 10.7)
0x04	Value Index Structure (see subclause 10.8)
0x05	Value Sub-Index Structure (see subclause 10.9)
0x06	BiMStreamReferences Structure (see subclause 10.12)
0x07	Reserved
0x08	Position Codes Structure (see subclause 10.6)
0x09-0xDF	Reserved
0xE0-0xFF	User Defined

10.4.5 structure_type and their matching valid structure_id

structure_type	*structure_id*	*Description*
0x00	0x00-0xFF	Used to identify Path Index Structure to which this configuration relates.
0x01	0x00-0xFF	User Defined

0x02	0x00	Data Repository Structure of type strings (see subclause 10.6.3)
0x02	0x01	Data Repository Structure of type binary data (see subclause 10.6.4)
0x02	0x02-0xFF	Reserved
0x03	0x00	Root Index. This is the Path Index to start a search of a stand alone or hierarchical index (see subclause 10.7)
0x03	0x01-0xFF	Used to identify hierarchical child index (see subclause 10.7)
0x04	0x00-0xFF	Used to identify a specific instance of a value index structure, within an Index Access Unit (see subclause 10.8)
0x05	0x00-0xFF	Used to identify a specific instance of a value sub-index structure, within an Index Access Unit (see subclause 10.9)
0x06	0x00-0xFF	Used to identify a specific instance of a BiMStreamReference Structure (see subclause 10.12)
0x07	0x00-0xFF	Reserved
0x08	0x00-0xFF	Reserved
0x09-0xDF	0x00-0xFF	Reserved
0xE0-0xFF	0x00-0xFF	User Defined

Structure types whose structure_id is 'Reserved' shall set structure_id to 0xFF.

10.5 IndexConfiguration

10.5.1 Overview

This structure contains the configuration parameters associated with the index whose PathIndex structure resides within the same IndexAccessUnit and has the same structure_id.

If this structure is not present within the IndexAccessUnit, the following default values shall be used,

PathIndexKey_format	0x00
BTree_order	0x00
global_value_index_config_flag	'0'
LocalAccessUnitID	Defined by underlying transport layer

The remaining fields will not be referenced within the index, as global_value_index_config_flag is zero.

10.5.2 Syntax

IndexConfiguration() {	**No. of Bits**	**Mnemonic**
PathIndexKey_format	8	uimsbf
BTree_order	8	uimsbf
overlapping_Partitions	1	bslbf
CompoundValueSubIndices	1	bslbf
partition_list	1	bslbf
reserved	4	bslbf
global_value_index_config_flag	1	bslbf
BiMStreamReference_format	8	uimsbf
LocalAccessUnitID	16	uimsbf
}		

10.5.3 Semantics

Name	*Definition*
PathIndexKey_format	Specifies the format used for the PathIndexKey entries. See table below.
BTree_order	The order of the BTree, defined as the number of node references per node. If the order is 1, then the BTree is equivalent to an ordered list, as it only has a right hand branch. A value of 0x0 signals an unordered list.
overlapping_Partitions	Indicates that a range of ValueIndexKeys found within a ValueIndexPartition may overlap those within another ValueIndexPartition structure.
CompoundValueSubIndices	Indicates that the single layer encoding format has been used within the ValueSubIndex structures. Sublause 10.10
partition_list	If 1, indicates that there is a partition list of SubValueIndices, If 0, There is only one SubValueIndex, which is contained inline within the ValueIndex structure.
reserved	Bits reserved for future use, set to '1'.
global_value_index_config_flag	If 1, indicates that the following global overlapping_SubValueIndices, single_layer_SubValueIndices, and BiMStreamReference_format values should be used for all value index structures within this index stream.
BiMStreamReference_format	Specifies the format of the BiMStreamReference. (see subclause 10.8.3)
LocalAccessUnitID	The Access Unit id of the Local Access Unit to which Local BiMStreamReferences refer.

PathIndexKey_format	**Format**
0x00	PathIndexKey_literal (see subclause 10.7.5)
0x01	PathIndexKey_context_path (see subclause 10.7.7)
0x02-0xFF	Reserved

10.6 Data Repository

10.6.1 Overview

The Data Repository forms the base structure, used to hold string data and binary data. All references to the data repository are local. i.e. from within the same Index Access Unit. The type of data, which the data repository carries, is indicated by the structures associated structure_id.

10.6.2 Syntax

DataRepository() {	**No. of Bits**	**Mnemonic**
if(structure_id == 0x00) {		
string_repository()		
}		
else if(structure_id == 0x01) {		
binary_repository()		
}		
else {		
Reserved		
}		
}		

10.6.3 string_repository

10.6.3.1 Overview

The string repository is used to hold all strings used by structures within the same Index Access Unit.

There shall only ever be one string repository per Index Access Unit. References to this repository are always local (that is, from the same Index Access Unit). Support is provided for identifying the string encoding system, to enable the use of non ASCII base character sets. The use of length fields or termination values are dependent on the string encoding used.

10.6.3.2 Syntax

string_repository() {	**No. of Bits**	**Mnemonic**
encoding_type	8	uimsbf
for(i=0; i<strings_count; i++) {		
for(j=0; j<string(i).length; j++) {		
string_character	8+	bslbf
}		
string_terminator	8+	bslbf
}		
}		

10.6.3.3 Semantics

Name	*Definition*
encoding_type	An 8 bit field used to define the character encoding system, according to section 10.6.3.4.

10.6.3.4 Character Encoding and their termination values

encoding_type	*Description*	*Termination Value*
0x00	7 bit ASCII (ISO/IEC 10646-1 [1])	0x00
0x01	UTF-8	0x00
0x02	UTF-16	0x0000
0x03	GB2312	0x0000
0x04	EUC-KS	0x0000
0x05	EUC-JP	0x0000
0x06	Shift_JIS	0x0000
0x07-0xDF	Reserved	Undefined
0xE0-0xFF	User Defined	User Defined

10.6.4 binary_repository

10.6.4.1 Overview

The encoding of data in the binary repository is defined at the point of reference. Each item of data must either have a length explicitly encoded within it, or a length implicitly understood by the decoder (i.e. fixed length). No provision is made to define the data length within the binary data repository structure.

All entries shall be byte aligned.

There shall only ever be one binary data repository within a single Index Access Unit.

10.6.4.2 Syntax

binary_repository() {	**No. of Bits**	**Mnemonic**
for(i=0; i<value_count; i++) {		
for(j=0; j< length; j++) {		
value_byte	8	bslbf
}		
}		
}		

10.6.4.3 Semantics

Name	*Definition*
`value_byte`	A byte of binary value data

10.7 PathIndex

10.7.1 Overview

A path index structure capable of supporting, unordered lists, ordered lists, and b-trees is desirable, as each is optimal for different applications. However it is not desirable to implement multiple path index structure handlers in every decoder, and so a multipurpose structure is defined. This structure can be parsed by the same structure decoder whether it is list or b-tree, with minimal additional overhead in the decoder.

10.7.2 Syntax

PathIndex() {	**No. of Bits**	**Mnemonic**
for(int j=0; j<num_nodes; j++) {		
PathIndexNode[k]()		
}		
}		

10.7.3 PathIndexNode

10.7.3.1 Overview

The PathIndexNode represents the encoding of a single node within the PathIndex. Each PathIndexNode may contain one or more PathIndexKeys, depending on the BTree order.

10.7.3.2 Syntax

PathIndexNode() {	**No. of Bits**	**Mnemonic**
node_reference	8+	vluimsbf8
if(BTree_order > 1) {		
number_of_entries	ceil($\log^2$ BTree_order-1)	bslbf
}		
for(int k=0; k<number_of_entries; k++) {		
PathIndexKey[k] ()		
nextByteBoundary()		
if(BTree_order > 1) {		
node_reference[k]	8+	vluimsbf8
}		
ValueIndex_reference [k]()		
}		
}		

10.7.3.3 Semantics

Name	*Definition*
node_reference	A byte offset to the next sibling or child node within the path index. If the PathIndexKey to be located is less than that of the nodes PathIndexKey, then the preceding node_reference provides a link to the next level that should be searched within the B-Tree. If the index you are trying to locate is greater than the last index_key within the node then the last node reference provides a link to the next level that should be searched. If the node_reference is set to 0x00, then the bottom of the B-Tree has been reached and so the item can not be found within the index list.
number_of_entries	defines the number of keys within this index node. As the index node cannot have 0 keys the value is the number of keys -1.
PathIndexKey	This is the key to be compared against the PathIndexKey to be located. Entries will always be in increasing order.

10.7.4 PathIndexKey

10.7.4.1 Overview

The path index key is used by the client to identify and locate a Value Index for a query it wishes to perform. The path index key identifies the paths of nodes which have been indexed, and the paths of the values used to index the node.

10.7.4.2 Syntax

PathIndexKey () {	**No. of Bits**	**Mnemonic**
if(**PathIndexKey_format** == 0x00) {		
PathIndexKey_litteral ()		
} else if (**PathIndexKey_format** == 0x01) {		
PathIndexKey_context_path ()		
} else {		
undefined		
}		
}		

10.7.5 PathIndexKey_literal

10.7.5.1 Overview

The use of literals to identify indexed nodes and value nodes allows a value index to be located by the use of well known literals. This allows low end clients which have only predetermined searching capabilities fixed within their software to locate index paths via a simple 16 bit number. The PathIndexKey_literal also allows flexibility for client and server to use an application specific alternative as a key within the path index.

10.7.5.2 Syntax

PathIndexKey_literal () {	No. of Bits	Mnemonic
PathIndexKey_literal_value ()		
num_value_nodes	8	uimsbf
for(k = 0; k < **num_value_nodes**; k++) {		
PathIndexKey_literal_value ()		
value_encoding	16	uimsbf
}		
}		

10.7.5.3 Semantics

Name	*Definition*
num_value_nodes	The number of value nodes.
value_encoding	Signals the method of encoding used for the index key value. (subclause 10.7.5.4)

10.7.5.4 Value_encoding

10.7.5.4.1 Interpretation

value_encoding	**value encoding interpretation**
0x0000 – 0x00FF	Field is a 16 bit offset in bytes from the start of the string repository structure.
0x0100 – 0x01FF	Field contains an inline 2-byte value.
0x0200 – 0x0201 0x0300 0x0401	Field contains an inline 4-byte value.
0x0204 - 0x0206	Field contains an inline 1-byte value.
0x0202 - 0x0203	Field is a 16 bit offset in bytes from the start of the binary data repository.
0x0302 0x0400	Field contains an inline 8-byte value.
0x0204 – 0x02FF 0x0402 – 0x04FF	Undefined.
0x0500 – 0xFFFF	Reserved for future use.

10.7.5.4.2 Respective Sizes

value_encoding	**Description**	**Encoding**	**Size in bits**
0x0000	string type	Null-terminated string	variable (8+)
0x0001 – 0x00FF	Reserved for custom string types		
0x0100	signed short	two’s complement – Big Endian	16

0x0101	unsigned short	unsigned binary – Big Endian	16
0x0102 – 0x01FF	Reserved for custom 2 byte types		16
0x0200	signed long	two's complement – Big Endian	32
0x0201	unsigned long	unsigned binary – Big Endian	32
0x0202	variable length signed integer	One bit represents sign (0: positive, 1:negative), followed by abs(value) using vluimsbf5	variable (6+)
0x0203	variable length unsigned integer	vluismbf8	variable (8+)
0x0204	boolean	0:False 1:True	8
0x0205	signed byte	Two's complement	8
0x0206	unsigned byte	unsigned binary	8
0x0207 – 0x02FF	Reserved for custom integer types		
0x0300	signed float	IEEE standard 754-1985 – Big Endian	32
0x0301	reserved		
0x0302	signed double	IEEE standard 754-1985 – Big Endian	64
0x0303 – 0x03FF	reserved		
0x0400	dateTime	Modified Julian Date and Milliseconds (as defined in subclause 10.5.4.4)	64
0x0401	date	Modified Julian Date (as defined in subclause 10.5.4.5)	32
0x0402 – 0x04FF	Reserved for custom binary formats.		
0x0500 – 0xFFFF	Reserved for future use		

10.7.5.5 dateTime Codec

The XML Schema primitive is used widely, and so a specific codec has been designed for representing date time fields.

Times shall be based on GMT, with no provision provided for maintaining the local time offset information. Any requirements to localise time values shall be performed by the receiving terminal.

The dateTime primitive is represented as an 8-byte unsigned integer number (Big-Endian), Days are represented using the first 4 bytes using Modified Julian Date. Time is represented using the last 4 bytes expressed as the number of elapsed milliseconds since 00:00:00 hours.

The origin for the Modified Julian Date shall be Midnight 17th November 1858.

Example dates:

Date	**Modified Julian Date**
1st April 1980	44 330
30th January 2000	51 573
1st March 2001	51 969

Example dateTimes:

dateTime value	**Encoded value**
1980-04-01T02:00:00Z	`0x0000AD2A006DDD00`
2000-01-30T12:10:01Z	`0x0000C975029C59A8`
2001-03-01T00:00:00Z	`0x0000CB0100000000`

10.7.5.6 date Codec

The XML Schema primitive simple type date describes a date within the Gregorian calendar. Within the XML the date takes the form of a string as defined by ISO/IEC 8601.

The XML Schema date primitive shall be represented as a 4-byte unsigned integer (Big-Endian). It shall contain the number of days using the Modified Julian Date format, as described in subclause 10.7.5.5.

10.7.6 PathIndexKey_literal_value

10.7.6.1 Overview

Literal values for the PathIndexKey are used where the encoder and decoder have additional knowledge about the context paths used within the index.

An example is where indexed nodes are always aligned with fragments, and the fragments use a limited set of context paths. In this instance the context paths used for the fragments, and hence the indexed node, are assigned a number which is known to the encoder and the decoder. This number can then be used in place of the context path.

Another instance where literal keys are useful is where a fragment is to be indexed based on data not contained in the source instance document. For instance an index of fragments which have been changed in the last day could be generated.

The literal index key places a requirement for both the encoder and decoder to understand the meaning of the literal key. Otherwise the decoder will not be able to use the index. Any index based on an unknown literal key does not prevent the decoder from using other value Indices specified within the same path index.

10.7.6.2 Syntax

PathIndexKey_literal_value () {	**No. of Bits**	**Mnemonic**
literal_type	16	uimsbf
if(**literal_type** < 0x8000) {		
UserDefined_literal	(Lower 15 bits of **literal_type**)	
} else if (**literal_type** < 0xFF00) {		
UserDefined_inlined	8*(**literal_type** & 0xFF)	
} else if (**literal_type** < 0xFFFD) {		
reserved	16	uimsbf
} else if (**literal_type** == 0xFFFE) {		
context_path_length	8+	vluimsbf8
context_path	variable	uimsbf
} else if (**literal_type** == 0xFFFF) {		
indexed_node_xpath_ptr	16	uimsbf
}		
}		

10.7.6.3 Semantics

Name	*Definition*
literal_type	The type of literal value encoding
UserDefined_literal	15 bit literal value of user defined significance. Value is lower 15 bits of **literal_type**.
UserDefined_inlined	variable length inlined used defined data
context_path_length	The length of the index root context path in bits.
context_path	This is a variable length field, which identifies the index root element context path. This is encoded as a context path using the Context Path syntax as defined in subclause 7.6.5 in document ISO/IEC 15938-1:2002, with position codes normalized to 1.
indexed_node_xpath_ptr	Pointer to a W3C XPath expression within the string repository

10.7.7 PathIndexKey_context_path

10.7.7.1 Overview

The PathIndexKey_context_path allows a client device to determine and use value indices without prior knowledge of what the Path Index is likely to contain. This provides a very flexible index to be generated by a server, and still be decoded by a client.

In contrast with the literal path index key, the context path does not require any external definitions to be known by the encoder or decoder, other than the XML schema.

10.7.7.2 Syntax

PathIndexKey_context_path () {	**No. of Bits**	**Mnemonic**
indexed_node_context_path_length	8+	vluimsbf8
indexed_node_context_path	variable	uimsbf
do {		
valuenode_indicator	1	bslbf
if(**valuenode_indicator** == ‘1’) {		
valuenode_context_path_length	8+	vluimsbf8
valuenode_context_path	variable	uimsbf
value_encoding	16	uimsbf
}		
} while(**valuenode_indicator** == ‘1’)		
if(num_valuenodes() == 0)		
{		
value_encoding	16	uimsbf
}		
}		

10.7.7.3 Semantics

Name	*Definition*
indexed_node_context_path_length	The length of the index root context path in bits.
indexed_node_context_path	This is a variable length field, which identifies the index root element context path. This is encoded as a context path using the Context Path syntax as defined in subclause 7.6.5 in document ISO/IEC 15938-1:2002. If position code information is present within the context path, it shall be ignored.
valuenode_indicator	A '1' indicates another value node follows A ‘0’ indicates no more value nodes follow (End of list)
valuenode_context_path_length	The length valuenode_context_path in bits.
valuenode_context_path	This is a variable length field, which identifies the context path of the value node. This is encoded as a relative context path using the Context Path syntax as defined in subclause 7.6.5 in document ISO/IEC 15938-1:2002. If position code information is present within the context path, it shall be ignored.
value_encoding	Signals the method of encoding used for the index key value. (subclause 10.7.5.4)
num_valuenodes()	Return the number of value nodes defined in the preceding key list

Note: If no value nodes are defined in the key list, then the indexed node must be an element of a simple type, or an attribute, and the value of this node is used as the key value. If value nodes are defined in the key list, then the indexed nodes must be elements of simple type, or attributes, and the values of these nodes are used as the key values, In the case where value nodes are defined in the key list, the indexed node may be any element or attribute, and the value of the indexed node is not used within the key.

10.7.8 ValueIndex_reference

10.7.8.1 Overview

The ValueIndex_reference is used to specify the Index Access Unit and structure_id of the referenced ValueIndex structure.

10.7.8.2 Syntax

ValueIndex_reference () {	**No. of Bits**	**Mnemonic**
IndexAccessUnit_identifier	16	uimsbf
ValueIndex_identifier	8	uimsbf
}		

10.7.8.3 Semantics

Name	*Definition*
IndexAccessUnit_identifier	The ID of the Index Access Unit containing the referenced Value Index structure.
ValueIndex_identifier	The ID of the Value Index structure within the Index Access Unit. This is carried in the structure_id field of the Index Access Unit header.

10.8 ValueIndex

10.8.1 Overview

The ValueIndex structure is the top level of an index. It provides a list of all ValueSubIndexReference fields and the ranges of ValueIndexKeys that they contain. When considering a classic indexing system it is normal for there not to be any overlaps in the range of ValueIndexKeys to be found within a given set of sub indexes. This is to minimise the amount of searching required to find a particular value.

Having overlapping ValueSubIndex structures can lead to sequential searching of ValueSubIndex structures, introducing an associated decrease in performance. However in some circumstances it may be desirable to allow this, to simplify index compilation or transmission.

In the case of overlapping ValueSubIndexReferences they shall be declared within the index structure in order of search priority. Where the first declared ValueSubIndexReferences, which may contain the set of required ValueIndexKeys, has the highest priority.

10.8.2 Value Ordering

The ordering of index entries within an index is dependent on a field's primitive XML schema simple type. In the case of strings the order may be dependent on the selected language, and not necessarily in alphanumeric order.

Table Amd2.2 — Defined index order for primitive simple types

Simple Type	Ordering
string	All strings shall be ordered in increasing Lexicographical order. Lexicographical ordering is language dependent, and may not be alphanumeric.
anyURI	Increasing alphanumeric order.
boolean	‘False’ precedes ‘True’
NMTOKEN	Increasing binary representation order
gYear	Increasing numeric value
integer	Increasing numeric value with negative values first
date	Increasing date value
nonNegativeInteger	Increasing numeric (binary) value
positiveInteger	Increasing numeric (binary) value

dateTime	Increasing dateTime (binary) value
duration	Increasing duration (binary)
float	Increasing numeric value (negative values first)
double	Increasing numeric value (negative values first)

Given high_ValueIndexKey, (a_1, a_2, ..., a_n) and (b_1, b_2, ..., b_n), of two arbitrary ValueSubIndices among the ValueSubIndices list, the sorting of ValueSubIndices is determined as follows:

(a_1, a_2, ..., a_n) is larger than (b_1, b_2, ..., b_n) if and only if there exists an integer i ($0 \le i \le n-1$) such that for every $j (0 \le j \le i-1)$, $a_j = b_j$ and $a_i > b_i$.

(a_1, a_2, ..., a_n) is smaller than (b_1, b_2, ..., b_n) if and only if there exists an integer i ($0 \le i \le n-1$) such that for every $j (0 \le j \le i-1)$, $a_j = b_j$ and $a_i < b_i$.

(a_1, a_2, ..., a_n) is equal to (b_1, b_2, ..., b_n) if and only if for every $i (1 \le i \le n)$, $a_i = b_i$.

Specifically, within the ValueIndexPartitionList() structure, if there is no overlapping between ValueSubIndices, for all j between 0 and ValueSubIndex_count-1 (high_ValueIndexKey[j,0], …, high_ValueIndexKey[j,k]) is smaller than (high_ValueIndexKey[j+1,0], …, high_ValueIndexKey[j+1,k])

"j" is the ValueSubIndex identifier
"k" is the value node identifier

This function high_ValueIndexKey[j,k] takes its value according to the loop defined in the ValueIndexPartitionList() table.

10.8.3 Syntax

ValueIndexPartitionList() {	**No. of Bits**	**Mnemonic**
if (!**global_value_index_config_flag**){		
overlapping_Partitions	1	bslbf
CompoundValueSubIndices	1	bslbf
partition_list	1	bslbf
reserved	5	bslbf
BiMStreamReference_format	8	uimsbf
}		
if(**partition_list** == ‘1’) {		
for (j=0; j<**ValueSubIndex_count**, j++) {		
for(k=0; k<**num_valuenodes**; k++) {		

if (**overlapping_Partitions** == ‘1’) {		
low_ValueIndexKey[j][k]	field encoding dependent	uimsbf
}		
high_ValueIndexKey[j][k]	field encoding dependent	uimsbf
}		
ValueSubIndexReference[j]()		
}		
} else {		
ValueSubIndex()		
}		
}		

10.8.3 Semantics

Name	*Definition*
overlapping_Partitions	When set to ‘1’, indicates that one or more of the value sub indices which form this value index, overlap with respect to the range of values found within the sub index. Where sub indices overlap, the sub indices are declared in descending order of search priority. When set to ‘0’, indicates that the sub indices do not overlap, and the declared sub indices are ordered in ascending order.
CompoundValueSubIndices	indicates the data structures used within the corresponding ValueSubIndex structures to represent keys with multiple values. When set to ‘1’ it indicates that all values for a given index entry are declared together in a single CompoundValueSubIndex structure. When set to ‘0’ it indicates that each value of a key is contained within a separate SingleValueSubIndex structure.
partition_list	If 1, indicates that there is a partition list of SubValueIndices, If 0, There is only one SubValueIndex, which is contained inline within the ValueIndex structure.
BiMStreamReference_format	Identifies the format and interpretation of the BiMStreamReference field which is used within the ValueSubIndex (leaf field). See Table Amd2.3 — BiMStreamReference formats
low_ValueIndexKey	The lowest value that can be referenced by an entry in a given value index partition. The lowest value signalled in low_ValueIndexKey may not be the lowest ValueIndexKey actually present in the given value index partition, it merely indicates that the referenced value index partition structure may contain entries with ValueIndexKeys in the given range. The size and type of encoding used and the interpretation of the low_ValueIndexKey are defined by the value_encoding within the PathIndex structure.
high_ValueIndexKey	The highest ValueIndexKey that can be referenced by the given value index partition. The highest value signalled in high_ValueSubIndex may not be the highest value actually present in the given fragment, it merely indicates that the referenced value index partition structure may contain entries with ValueIndexKeys in the given range. The size and type of encoding used and the interpretation of the high_ValueIndexKey are defined by value_encoding within the PathIndex structure.

Table Amd2.3 — BiMStreamReference formats

Value	Meaning
0x00	local_BiMStreamReference (see subclause 10.14.3)
0x01	remote_BiMStreamReference (see subclause 10.14.2)
0x02	local_BiMStreamReference_with_position (see subclause 10.14.5)
0x03	remote_BiMStreamReference_with_position (see subclause 10.14.4)
0x04	PathIndexReference (see subclause 10.14.6)
0x05 – 0x3F	reserved
0x40 – 0x7F	User Private
0x80	local_BiMStreamReference_vl (see subclause 10.14.8)
0x81	remote_BiMStreamReference_vl (see subclause 10.14.6)
0x82	local_BiMStreamReference_with_position_vl (see subclause 10.14.10)
0x83	remote_BiMStreamReference_with_position_vl (see subclause 10.14.9)
0x84	PathIndexReference_vl (see subclause 10.15.5)
0x85 – 0xDF	reserved
0xE0 – 0xFF	User Private

It should be noted that the high_ValueIndexKey for all but the first value node may be lower than the previous high_ValueIndexKey sub index entry. This is caused when there is a difference in the value of the parent value node.

For example if we have an index keyed on channel & event time nodes, we could have a set of sub indexes with the following ranges:

Sub index 1 – channel high_ValueIndexKey= ‘3’, event time high_ValueIndexKey= ‘12:00’

Sub index 2 – channel high_ValueIndexKey= ‘4’ event time high_ValueIndexKey= ‘09:00’

Where the index uses CompoundValueSubIndeces, the ordering of the high_ValueIndexKey shall match that defined for the index within the PathIndex structure.

When defining the range of values that a particular value index partition shall cover, sufficient space should be left to enable the addition of further index entries without impacting other value index partitions. For example if a ValueSubIndex can hold a maximum of say 64K entries, it is recommended that the range of current entries should equal around half to two thirds the space. This leaves plenty of room for additional entries without having to changing the way in which the value index is split into value index partitions.

10.8.5 ValueSubIndexReference

10.8.5.1 Overview

The ValueSubIndex_reference is used to specify the Index Access Unit and structure_id of the referenced ValueSubIndex structure.

10.8.5.2 Syntax

ValueSubIndexReference () {	**No. of Bits**	**Mnemonic**
ValueSubIndex_IndexAccessUnitID	16	uimsbf
ValueSubIndex_identifier	8	uimsbf
}		

10.8.5.3 Semantics

Name	*Definition*
ValueSubIndex_IndexAccessUnitID	The id of the Index Access Unit carrying the first ValueSubIndex of the described value index partition.
ValueSubIndex_identifier	This field identifies the ValueSubIndex structure instance containing the described value sub index. This value is carried within the structure_id field of the Index Access Unit header.

10.9 ValueSubIndex

10.9.1 Overview

A value sub index provides references to fragments, which contain values within the range specified for this value sub index. The structure supports indexes with both single and multiple value keys. In the case of indices with multiple value keys, the syntax provides two methods:

- SingleLayer CompoundValues - All values define together within a single ValueSubIndex.
- MultiLayer SingleValues- Each ValueSubIndex indexes a single value of a key.

10.9.1.1 SingleLayer CompoundValue SubIndex

Single Layer Structures provide a simple mechanism for describing multiple value key indices. As each entry in the structure can be decoded one by one in a straightforward manner, this structure would be preferred in a situation where the received index data need to be reorganised in the receiver before its use. Note that the index data can be restructured inside the receiver according to its own storage method and query processing policy. For example, a receiver may want to reorganise one of the received indices in its own B-tree index.

In addition, the Single Layer Structure provides an efficient mechanism for representing multiple value indexes, where there is typically a one to one mapping e.g. <surname, givenname>.

10.9.1.2 Multi Layer SingleValue SubIndex

Multi Layer Structures provide an efficient mechanism for describing multiple value indexes with common single value indexes. This is achieved with the use of multiple SingleValueSubIndex structures, where each structure is used to describe one layer of a multiple value index, (layer is equal to a key field of a multi field index).

Each index entry within the SingleValueSubIndex, point to further SingleValueSubIndex structures (except for the leaf field), which contain index entries of the next layer.

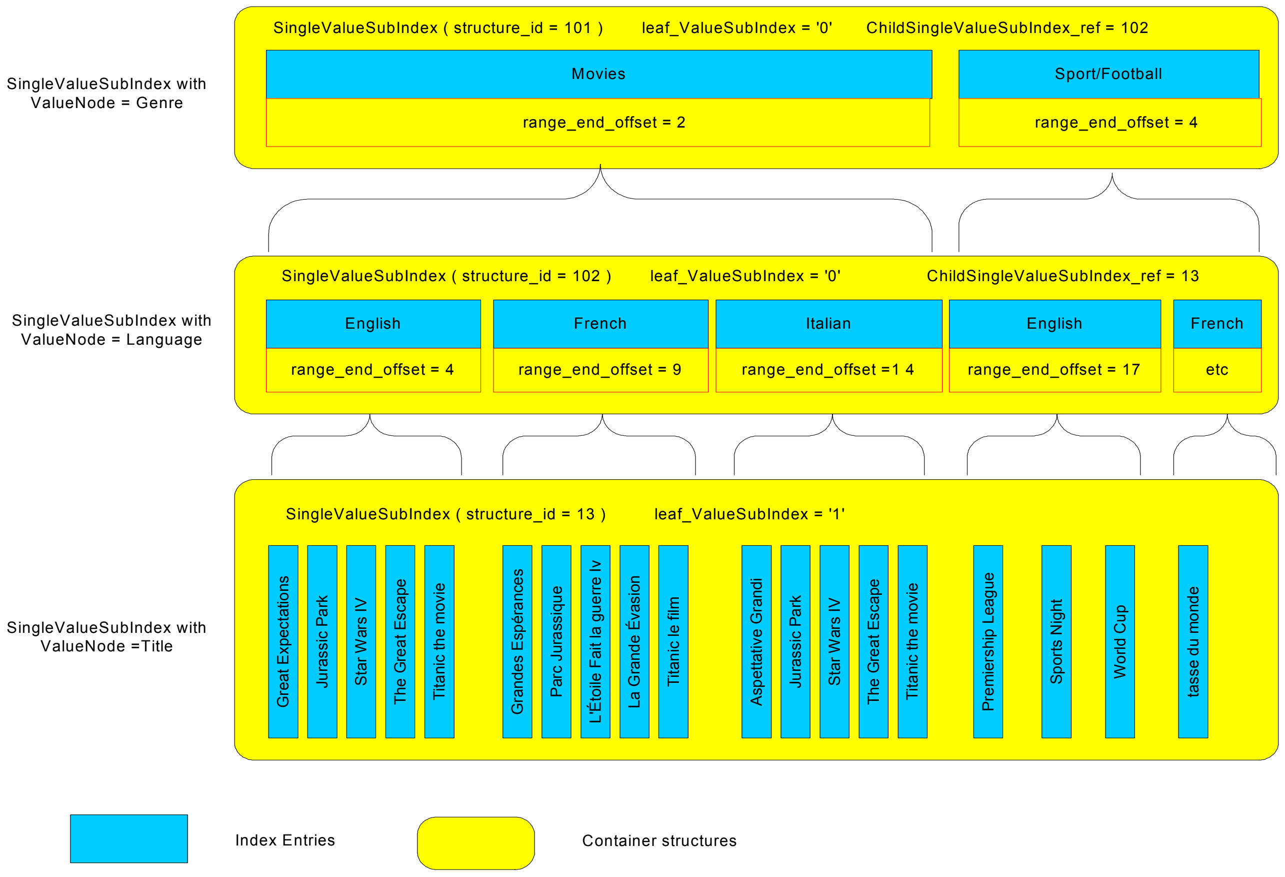


Figure Amd2.5 — Example ValueSubIndex structure (using multi layer syntax) for an index with 3 key fields (Genre, Language, & Title)

The ValueSubIndex structure is formed of two parts:

- ValueSubIndex_header.
- ValueSubIndex_entries.

The ValueSubIndex_header defines how the ValueSubIndex_entries sub structure should be interpreted, and indirectly defines the size of each index entry.

All entries within the ValueSubIndex_entries sub structure are ordered in ascending order. All entries are also of a fixed size, which enables the sub structure to be efficiently searched using a binary search algorithm.

The number of entries within the structure is not explicitly defined, but can be inferred as follows:

num_entries = (structure_length - sizeof(ValueSubIndex_header))/sizeof(ValueSubIndex_entry)

It should be noted that the syntax used within SingleValueSubIndex structures is not always common across all sub indices. Therefore the header of each SingleValueSubIndex should be parsed to infer the syntax used within a given instance.

10.9.2 Syntax

ValueSubIndex() {	**No. of Bits**	**Mnemonic**
ValueSubIndex_header()		
if(**CompoundValueSubIndices** == ‘0’) {		
SingleValueSubIndex ()		
} else {		
CompoundValueSubIndex ()		
}		
}		

10.9.3 Semantics

Name	*Definition*
CompoundValueSubIndices	This value is obtained from the value index which referenced this value sub-index.

10.9.4 ValueSubIndex_header

10.9.4.1 Overview

Given ValueIndexKeys, ($a_1, a_2, ..., a_n$) and ($b_1, b_2, ..., b_n$), of two CompoundValueIndex_entries, the order between the two entries is determined as follows:

($a_1, a_2, ..., a_n$) is larger than ($b_1, b_2, ..., b_n$) if and only if there exists an integer i ($0 \le i \le n$-1) such that for every j($0 \le j \le i$-1), $a_j = b_j$ and $a_i > b_i$.

($a_1, a_2, ..., a_n$) is smaller than ($b_1, b_2, ..., b_n$) if and only if there exists an integer i ($0 \le i \le n$-1) such that for every j($0 \le j \le i$-1), $a_j = b_j$ and $a_i < b_i$.

($a_1, a_2, ..., a_n$) is equal to ($b_1, b_2, ..., b_n$) if and only if for every i($1 \le i \le n$), $a_i = b_i$.

Specifically, within the ValueSubIndex() structure, for all j between 0 and num_entries-1 (ValueIndexKey[j,0], …, ValueIndexKey[j,k]) is smaller than (ValueIndexKey[j+1,0], …, ValueIndexKey[j+1,k])

10.9.4.2 Syntax

ValueSubIndex_header () {	**No. of Bits**	**Mnemonic**
leaf_ValueSubIndex	1	bslbf
multiple_BiMStreamReferences	1	bslbf
reserved	6	bslbf
}		

10.9.4.3 Semantics

Name	*Definition*
leaf_ValueSubIndex	This shall be set to '1' when the ValueSubIndex carries the leaf field of an index (last indexed field). Which indicates that the structure contains references to fragments, and not to further ValueSubIndex structures. This field is only used within multi layer value sub indexes. When a single layer sub index is being described this flag shall be ignored.
multiple_BiMStreamReferences	A flag which when set to '1' indicates that there are potentially multiple referenced fragments which have the same set of ValueIndexKeys. This provides a more bandwidth efficient mechanism, when multiple fragments have the same set of ValueIndexKeys. The actual BiMStreamReferences are carried in a separate structure within an Index Access Unit, and an offset is used to reference the set of relevant BiMStreamReferences within the structure. When the flag is set to '0' it indicates that BiMStreamReference are defined inline.

10.10 CompoundValueSubIndex

10.10.1 Overview

The CompoundValueSubIndex allows a set of BiMStreamReferences to be addressed as a set of one or more ValueIndexKeys. The compound value sub-index groups the values together in a flat index. This is beneficial if the different data values are uncorrelated.

10.10.2 Syntax

CompoundValueSubIndex () {	**No. of Bits**	**Mnemonic**
ValueSubIndex_entries {		
for (j=0; j<num_entries; j++) {		
for(f=0; f<num_fields; f++) {		
ValueIndexKey	value encoding dependent	**uimsbf**
}		
if(multiple_BiMStreamReferences == '1') {		
BiMStreamReference_end_offset	**16**	**uimsbf**
}		
else {		
BiMStreamReference()		
}		
}		
}		
}		

10.10.3 Semantics

Name	*Definition*
ValueIndexKey	The value of the ValueIndexKey of the referenced fragment. The size and meaning of this field depends on the value of the value_encoding member of the relevant PathIndex structure. The values of the ValueIndexKey must be within the range given for this value index partition.
BiMStreamReference_end_offset	When the multiple_BiMStreamReferences flag is set to '1' in the ValueSubIndex_header this field is used to indicate the inclusive end offset within the BiMStreamReferences structure where the set of valid references can be found. The format of these BiMStreamReferences is defined by the BiMStream_reference_format declared within the PathIndex structure.

The BiMStreamReference_start_offset is implicit from the previous entry within the ValueSubIndex, as follows.

- If it's the first entry within the ValueSubINdex_entries then BiMStreamReference_start_offset shall equal 0.
- If it's not the first entry, the previous entries BiMStreamReference_end_offset + 1 shall be used as the current entries inclusive BiMStreamReference_start_offset.

```
if (current index != 0) {
   BiMStreamReference_start_offset = value sub index_entries[current index-1].
BiMStreamReference_end_offset + 1;
 }else {
   BiMStreamReference_start_offset = 0;
}
```

It should be noted that for fixed size BiMStreamReference formats these references are based on BiMStreamReference entries and not byte offsets. The actual byte offset within the BiMStreamReferences structure is calculated as follows:

```
if(MostSignificantBit(BiMStreamReference_format) == '0')
{
    byte_offset = BiMStreamReference_end_offset * sizeof(BiMStreamReference());
}
else
{
    byte_offset = BiMStreamReference_end_offset;
}
```

10.11 SingleValueSubIndex

10.11.1 Overview

The SingleValueSubIndex allows a set of BiMStreamReferences to be addressed by only one ValueIndexKey. Multiple ValueIndexKey indices can however be built up hierarchically using the SingleValueSubIndex. This is beneficial if the different data values are correlated, as a hierarchical search requires less comparisons whilst searching, and can result in a much smaller structure size. This may however result in a very large structure size if the data is uncorrelated, so care must be taken when deciding which ValueSubIndex format to use.

10.11.2 Syntax

SingleValueSubIndex () {	**No. of Bits**	**Mnemonic**
if (**leaf_ValueSubIndex**='0') {		
child_ValueSubIndex_ref	8	uimsbf
}		
for (j=0; j<num_entries; j++) {		
ValueIndexKey	value encoding dependent	uimsbf
if(**leaf_ValueSubIndex** == ‘1’) {		
if(**multiple_BiMStreamReferences** == ‘1’) {		
BiMStreamReference_end_offset	16	uimsbf
}		
else {		
BiMStreamReference()		
}		
}		
else {		
range_end_offset	16	uimsbf
}		
}		
}		

10.11.3 Semantics

Name	*Definition*
child_ValueSubIndex_ref	This value identifies a further SingleValueSubIndex structure within the current Index Access Unit which holds index entries having a value of this layer equal to that defined within this sub index. The combination of this value and the range_end_offset enables you to locate a set of index entries, which have a specific value index key.
ValueIndexKey	The value of the ValueIndexKey of the referenced fragment. The size and meaning of this field depends on the value of the value_encoding member of the relevant PathIndex structure. The values of the ValueIndexKey must be within the range given for this value index partition.

BiMStreamReference_end_offset	offset of last BiMStreamReference within the BiMStreamReferences structure keyed by ValueIndexKey. BiMStreamReference_start_offset is defined implicitly in the same manor as range_start_offset.
range_end_offset	defines the set of entries within the referenced SingleValueSubIndex (a SingleValueSubIndex with its structure_id equal to child_ValueSubIndex_ref) having a value equal to that defined by the ValueIndexKey.

The range_end_offset is an inclusive offset from the start of the ValueSubIndex_entries of the target ValueSubIndex, where the range_end_offset for the set of entries which have the declared value can be found. The range_start_offset is implicit from the previous, entry within the SingleValueSubIndex, as follows.

- If it's the first entry within the ValueSubIndex_entries then range_start_offset shall equal 0.
- If it's not the first entry, the previous entries range_end_offset + 1 shall be used as the current entries inclusive range_start_offset.

```
if (current index != 0) {
   range_start_offset = value sub index_entries[current index-
1].range_end_offset + 1;
 }else {
   range_start_offset = 0;
}
```

It should be noted that these references are based on index entries and not byte offsets. So the actual byte offset within the structure is calculated as follows:

byte_offset = (range_end_offset * sizeof(ValueSubIndex_entry)) + sizeof(ValueSubIndex_header);

10.12 BiMStreamReferences structure

10.12.1 Overview

The BiMStreamReferences structure is used to carry BiMStreamReferences, which are referenced from the ValueSubIndex, where there are references with the same set of ValueIndexKeys.

There shall only ever be a maximum of one BiMStreamReferences structure within a single Index Access Unit.

10.12.2 Syntax

BiMStreamReferences() {	**No. of Bits**	**Mnemonic**
for(int i=0; i<num_references; i++) {		
BiMStreamReference()		
}		
}		

10.12.3 Semantics

Name	*Definition*
num_references	This value is inferred from the size of the structure which is declared within the IndexAccessUnit. I.e. num_references = structure_length/sizeof(BiMStreamReference());

10.13 Position Codes structure

10.13.1 Overview

The position codes structure is used to carry position code values for the index root element.

There shall only ever be a maximum of one position_codes structure within a single Index Access Unit.

10.13.2 Syntax

PositionCodes() {	**No. of Bits**	**Mnemonic**
for(int i=0; i<num_position_codes; i++) {		
position_codes ()		
}		
}		

10.13.3 Semantics

Name	*Definition*
num_position_codes	This value is inferred from the size of the structure which is declared within the IndexAccessUnit. I.e. num_position_codes = structure_length/sizeof(position_codes ());

10.14 BiMStreamReference formats

10.14.1 Overview

There are a number of defined BiMStreamReference formats to enable the referencing of fragments from an index entry.

10.14.2 remote_BiMStreamReference

10.14.2.1 Overview

When a data structure becomes quite large, or it is a requirement to be able to carousel the index at a different rate to that of the data, it is advantageous to split the index and data across a number of Index Access Units. This format provides a mechanism for an index entry to reference a fragment within another Index Access Unit.

10.14.2.2 syntax

remote_BiMStreamReference () {	**No. of Bits**	**Mnemonic**
remote_fragment_reference()		
}		

10.14.3 local_BiMStreamReference

10.14.3.1 Overview

It is quite possible to use the above method for referencing fragments within the same Index Access Unit, however it is not the most efficient way. Therefore the following method is supported.

10.14.3.2 Syntax

local_BiMStreamReference() {	**No. of Bits**	**Mnemonic**
local_fragment_reference()		
}		

10.14.4 remote_BiMStreamReference_with_position_codes

10.14.4.1 Overview

When a data structure becomes quite large, or it is a requirement to be able to carousel the index at a different rate to that of the data, it is advantageous to split the index and data across a number of Index Access Units. This format provides a mechanism for an index entry to reference a fragment within another Index Access Unit.

10.14.4.2 Syntax

remote_BiMStreamReference_with_position_codes () {	**No. of Bits**	**Mnemonic**
position_codes_reference()		
remote_fragment_reference()		
}		

10.14.5 local_BiMStreamReference_with_position_codes

10.14.5.1 Syntax

local_BiMStreamReference_with_position_codes () {	**No. of Bits**	**Mnemonic**
position_codes_reference()		
local_fragment_reference()		
}		

10.14.6 PathIndexReference

10.14.6.1 Overview

The PathIndexReference is used to reference another PathIndex structure within another IndexAccessUnit. This allows a large set of index data to be split into a number of smaller index data set.

For instance an electronic program guide's schedule index for seven days may be very large. By splitting the index up into seven data sets of one day each, the index can be represented as a parent index data set and seven child index data sets. This allows client devices with differing memory resources to cache none, part, or the entire weeks schedule index. Un-cached indices could still be acquired from the broadcast stream, albeit at a slower pace. This allows the performance of different client devices to be optimised according to their memory resources, whilst sharing the same index and BiM fragment data.

10.14.6.2 Syntax

PathIndexReference () {	**No. of Bits**	**Mnemonic**
target_index_access_unit	16	uimsbf
}		

10.14.6.3 Semantics

Name	*Definition*
target_index_access_unit	The index access unit identifier containing the target path index.

10.14.7 remote_BiMStreamReference_vl

10.14.7.1 syntax

remote_BiMStreamReference_vl () {	**No. of Bits**	**Mnemonic**
remote_fragment_reference_vl()		
}		

10.14.8 local_BiMStreamReference_vl

10.14.8.1 syntax

remote_BiMStreamReference_vl () {	**No. of Bits**	**Mnemonic**
local_fragment_reference_vl()		
}		

10.14.9 remote_BiMStreamReference_with_position_codes_vl

10.14.9.1 syntax

remote_BiMStreamReference_with_position_codes_vl () {	**No. of Bits**	**Mnemonic**
position_codes ()		
remote_fragment_reference_vl()		
}		

10.14.10 local_BiMStreamReference_with_position_codes_vl

10.14.10.1 syntax

local_BiMStreamReference_with_position_codes_vl () {	**No. of Bits**	**Mnemonic**
position_codes ()		
local_fragment_reference_vl()		
}		

10.15 Fragment References

10.15.1 local_fragment_reference

10.15.1.1 Syntax

local_fragment_reference () {	**No. of Bits**	**Mnemonic**
fragment_offset	16	uimsbf
}		

10.15.1.2 Semantics

Name	*Definition*
fragment_offset	The index into an access unit, defined by LocalAccessUnitID, where the fragment can be found. This is the position within the FUU list of the access unit, and not the byte offset.

10.15.2 local_fragment_reference_vl

10.15.2.1 Syntax

local_fragment_reference_vl () {	**No. of Bits**	**Mnemonic**
fragment_offset	8+	vluimsbf8
}		

10.15.2.2 Semantics

Name	*Definition*
fragment_offset	The index into an access unit, defined by LocalAccessUnitID, where the fragment can be found. This is the position within the FUU list of the access unit, and not the byte offset.

10.15.3 remote_fragment_reference

10.15.3.1 Syntax

remote_fragment_reference () {	**No. of Bits**	**Mnemonic**
target_access_unit	16	uimsbf
target_fragment	24	uimsbf
}		

10.15.3.2 Semantics

Name	*Definition*
target_access_unit	The access unit identifier of the access unit containing the target fragment.
target_fragment	The fragment update unit identifier that uniquely identifies a fragment within the target access unit.

10.15.4 remote_fragment_reference_vl

10.15.4.1 syntax

remote_fragment_reference_vl () {	**No. of Bits**	**Mnemonic**
target_access_unit	8+	vluimsbf8
target_fragment	8+	vluimsbf8
}		

10.15.4.2 Semantics

Name	*Definition*
target_access_unit	The access unit identifier of the access unit containing the target fragment.
target_fragment	The fragment update unit identifier which uniquely identifies a fragment within the target access unit.

10.15.5 PathIndexReference_vl

10.15.5.1 Overview

The PathIndexReference is used to reference another PathIndex structure within another IndexAccessUnit. This allows a large set of index data to be split into a number of smaller index data set, or segments.

10.15.5.2 Syntax

PathIndexReference_vl () {	**No. of Bits**	**Mnemonic**
target_index_access_unit	8+	vluimsbf8
}		

10.15.5.3 Semantics

Name	*Definition*
target_index_access_unit	The index access unit identifier containing the target path index.

10.16 Position Codes

10.16.1 Overview

Position codes use a literal value to identify a child element within its parent. Context paths can contain position codes to enable addressing of a specific element within a document. These position codes are necessary when multiple child elements can have the same name, to resolve the ambiguity. Within the indexing structures it is more efficient to code the context path's tree branch codes and position codes separately, as the tree branch codes are shared by all the fragments referenced by an index. Subclause 10.6 defines the encoding of these separated position codes.

10.16.2 position_codes_reference

10.16.2.1 Syntax

position_codes_reference() {	**No. of Bits**	**Mnemonic**
position_codes_ref	16	uimsbf
}		

10.16.3 Semantics

Name	*Definition*
position_codes_ref	Reference to an entry within the position codes structure, located within the same Index Access Unit, encoded as offset in bytes.

10.16.4 position_codes

10.16.4.1 Overview

Within a SingleValueSubIndex entry multiple BiM Stream References can be indexed by the same value. In this case the first BiM Stream References's position codes are encoded as absolute values. For subsequent BiM Stream References, only the deltas from the preceding BiM Stream Reference.

An example set of BiM Stream References and their encoding is shown below

AU = 0, FUU = 1, Position Codes = 1,1,2,1,1,4	encoded as 1,1,2,1,1,4
AU = 0, FUU = 2, Position Codes = 1,1,5,1,1,4	encoded as 0,0,3,0,0,0
AU = 0, FUU = 3, Position Codes = 1,1,10,1,1,4	encoded as 0,0,5,0,0,0
AU = 0, FUU = 5, Position Codes = 1,1,8,1,1,4	encoded as 0,0,-2,0,0,0

10.16.4.2 Syntax

position_codes () {	No. of Bits	Mnemonic
bits_per_position_code	5+	vluimsbf5
for(int j=0; j< no_of_position_codes; j++) {		
position_code	**bits_per_position_code**	
}		
0 (terminating position code)	**bits_per_position_code**	
nextByteBoundary()		
}		

10.16.5 Semantics

Name	*Definition*
bits_per_position_code	How many bits are used to signal each position code. The first entry within a set of position codes will always be an absolute value Note: if the position codes are encoded as absolute, then the position codes are treated as unsigned. e.g. 1,2,1,1 = 2 bits. The terminating zero position code is used to determine the number of codes, if the position codes are encoded as relative, then they are encoded as signed. e.g. 0,2,0,-8 = 4 bits. With relative position codes there can be zeros in the data as well as the terminating zero, so in this instance the number of position codes is determined from the absolute encoded position codes at the start of the set.
position_code	Position codes used to signal parent child relationship.

If the path index key can reference nodes of deferent depths within the document, such as a key with an XPath containing a descendent or self command (e.g. “//Name”), then BiMStreamReferenceFormats containing position codes cannot be used for that index.

This Indian Standard has been developed from Doc No.: LITD 14 (3135).

Amendments Issued Since Publication

Amendment No.	Date of Issue	Text Affected

Published by BIS, New Delhi