INTERNATIONAL STANDARD

ISO/IEC 15938-1

First edition
2002-07-01

Information technology — Multimedia content description interface —

Part 1: **Systems**

Technologies de l'information — Interface de description du contenu multimédia —

Partie 1: Systèmes

Reference number
ISO/IEC 15938-1:2002(E)

Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 3.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

ISO/IEC 15938-1 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

ISO/IEC 15938 consists of the following parts, under the general title *Information technology — Multimedia content description interface*:

— *Part 1: Systems*

— *Part 2: Description definition language*

— *Part 3: Visual*

— *Part 4: Audio*

— *Part 5: Multimedia description schemes*

— *Part 6: Reference software*

— *Part 7: Conformance testing*

— *Part 8: Extraction and use of MPEG-7 descriptions*

Annexes A to C of this part of ISO/IEC 15938 are for information only.

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.

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(..)	$ceil(x) = \begin{cases} \mathrm{int}(x) + 1 & x \geq 0 \\ \mathrm{int}(x) & x < 0 \end{cases}$
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.

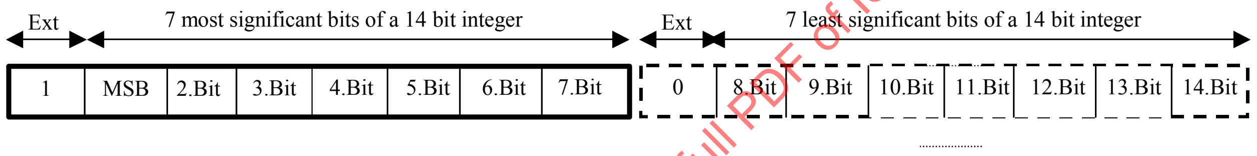
Ext
7 most significant bits of a 14 bit integer
Ext
7 least significant bits of a 14 bit integer
1
MSB
2.Bit
3.Bit
4.Bit
5.Bit
6.Bit
7.Bit
0
8.Bit
9.Bit
10.Bit
11.Bit
12.Bit
13.Bit
14.Bit

Figure 1 — Informative example for the vluimsbf8 data type

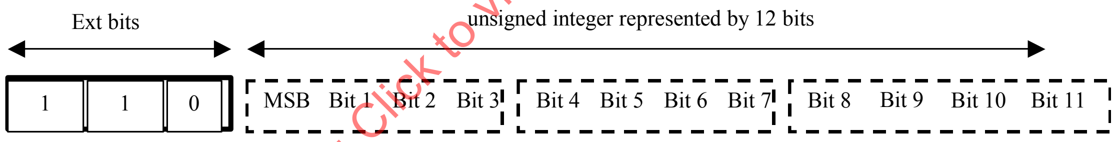

Figure 2 — Informative example for the vluimsbf5 data type

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.

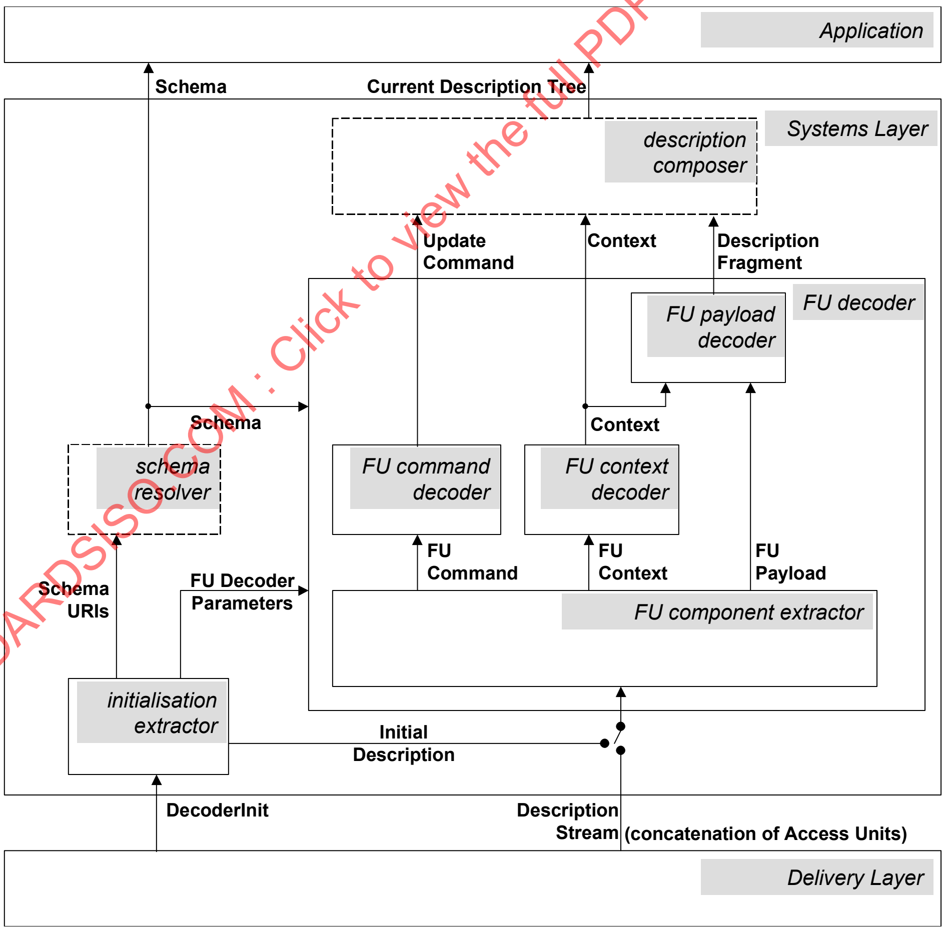

Figure 3 — Terminal Architecture
Dashed boxes in the systems layer are non-normative. FU is an abbreviation for Fragment Update.

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.

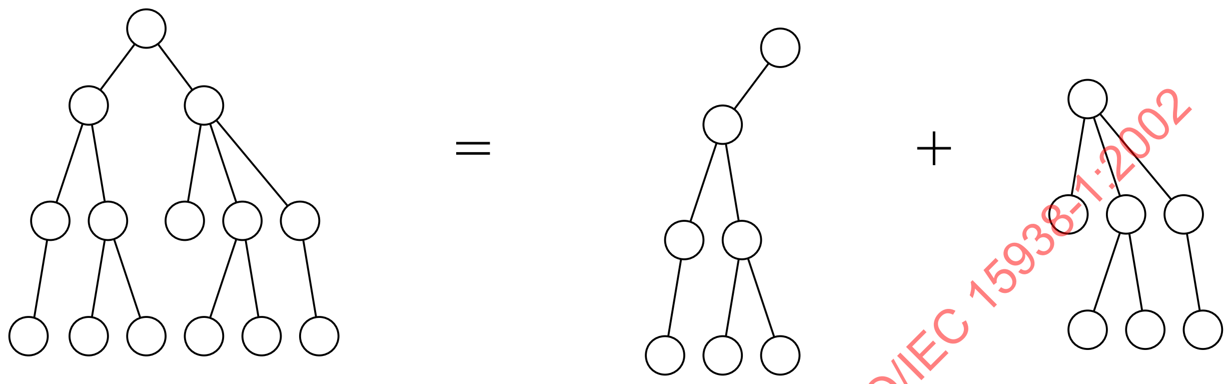

Figure 4 — Decomposition of a description into two description fragments

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.

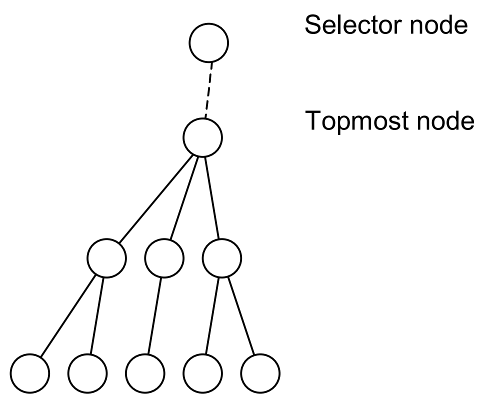

Figure 5 — Example for the tree representation of a description

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.

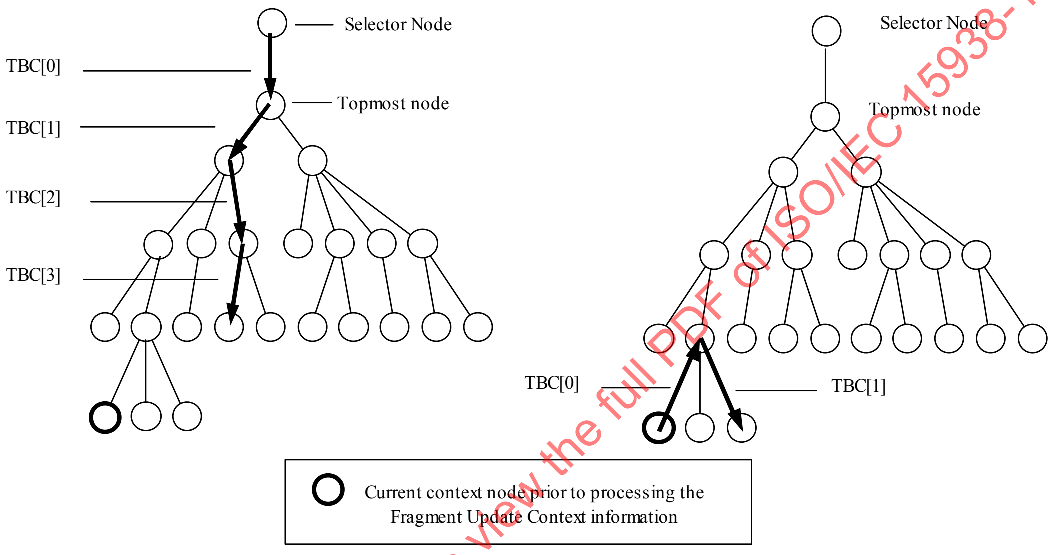

Figure 6 — Absolute (left) and relative context path example

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.

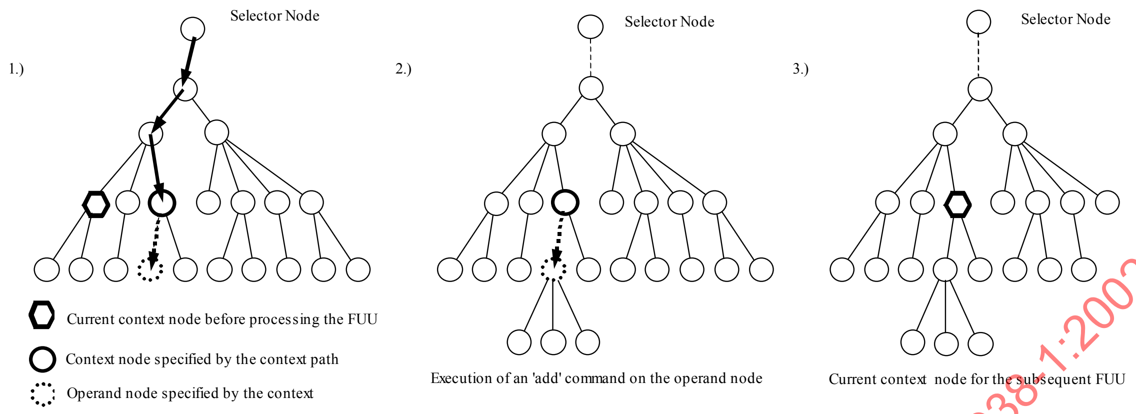
1.)
Selector Node
Current context node before processing the FUU
Context node specified by the context path
Operand node specified by the context
2.)
Selector Node
Execution of an 'add' command on the operand node
3.)
Selector Node
Current context node for the subsequent FUU

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.

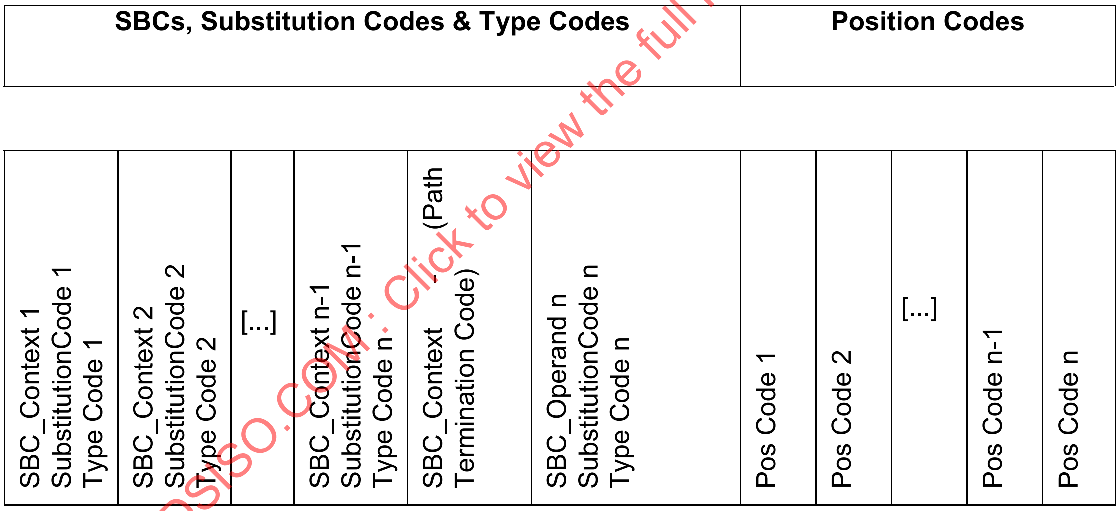
SBCs, Substitution Codes & Type Codes
Position Codes
SBC_Context 1 SubstitutionCode 1 Type Code 1
SBC_Context 2 SubstitutionCode 2 Type Code 2
[...]
SBC_Context n-1 SubstitutionCode n-1 Type Code n-
SBC_Context _(Path Termination Code)
SBC_Operand n SubstitutionCode n Type Code n
Pos Code 1
Pos Code 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))”.

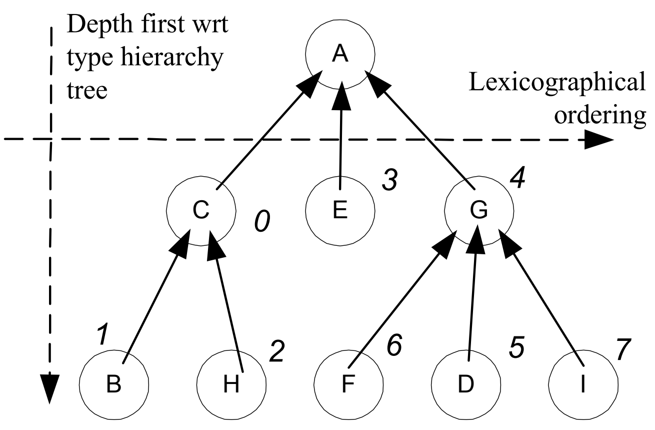

Figure 9 — Example for the Type Identification Code assignment for the types derived from A

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 i^{th} 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 i^{th} 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.

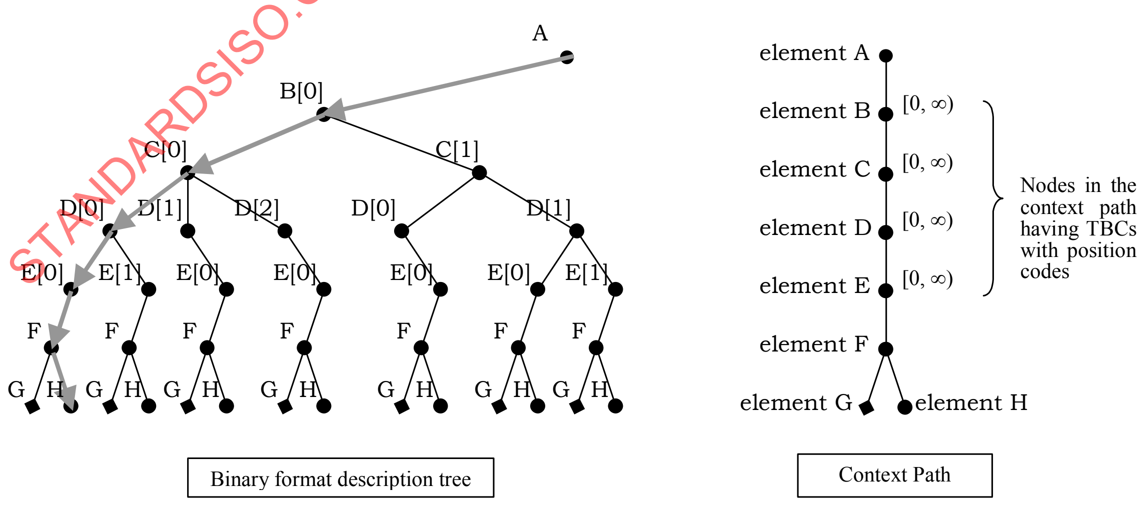

Figure 11 — Example for multiple-occurrence nodes in a context path

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.

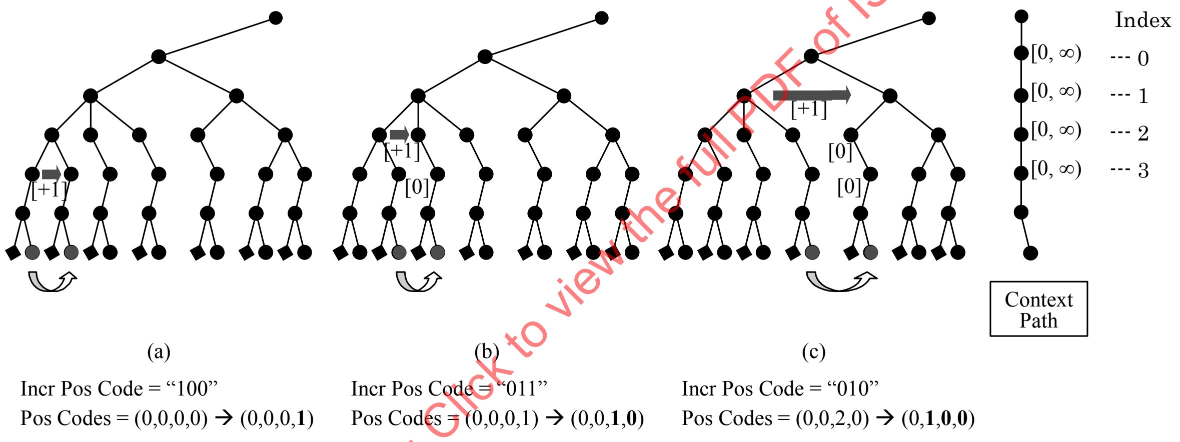

Figure 12 — Indicated Positions using Incremental Position Codes

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.

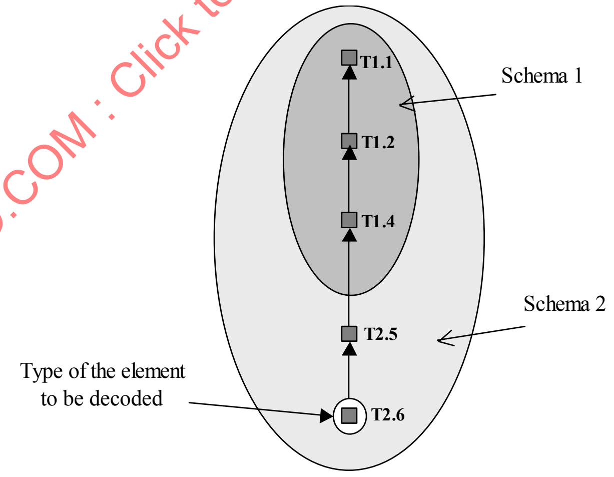

Figure 13 — Example of a type hierarchy defined across versions

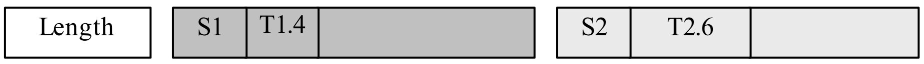

Figure 14 — Example of a forward compatible encoding

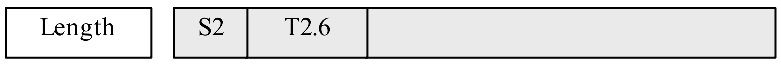

Figure 15 — Example of a non forward compatible encoding

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.

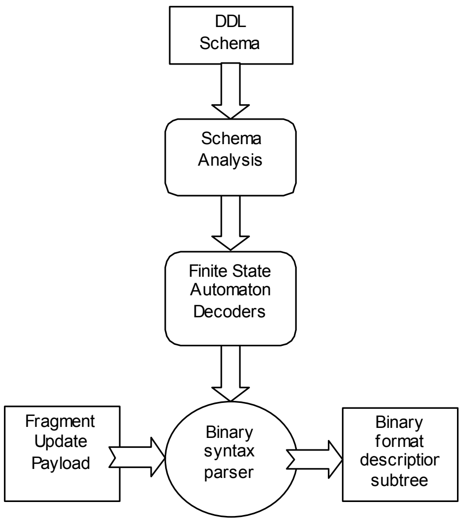

Figure 16 — The complex content decoding process

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.

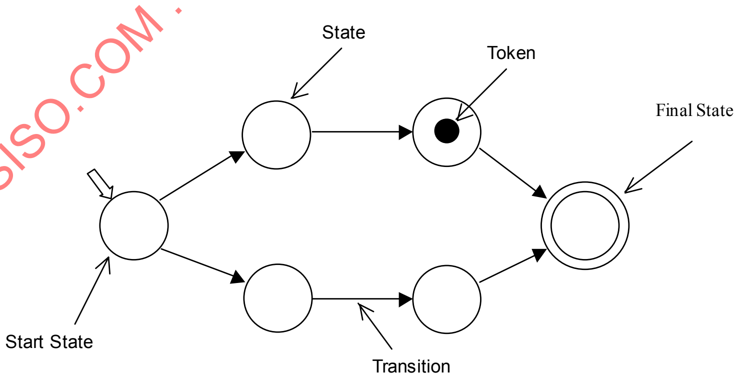

Figure 17 — Example of an automaton

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.

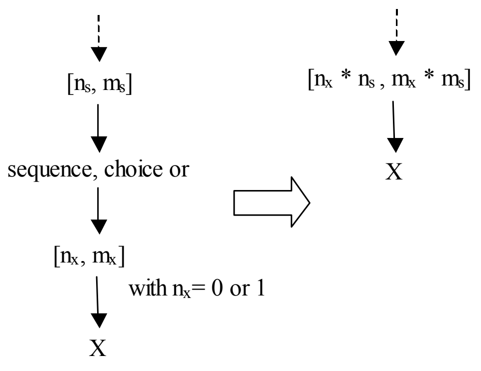

Figure 18 — Group simplification rule

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.

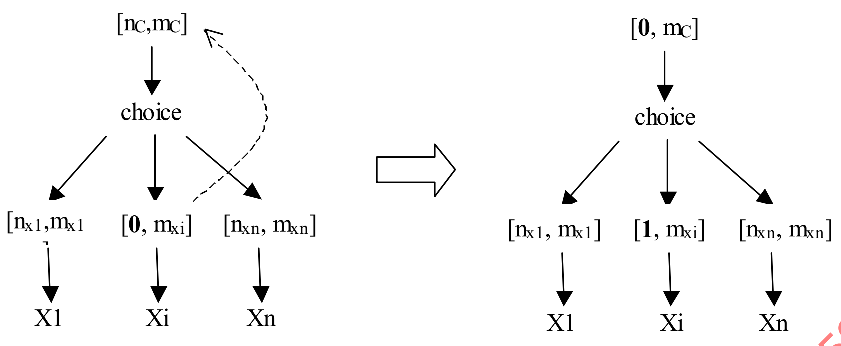

Figure 19 — Empty choice simplification rule

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.

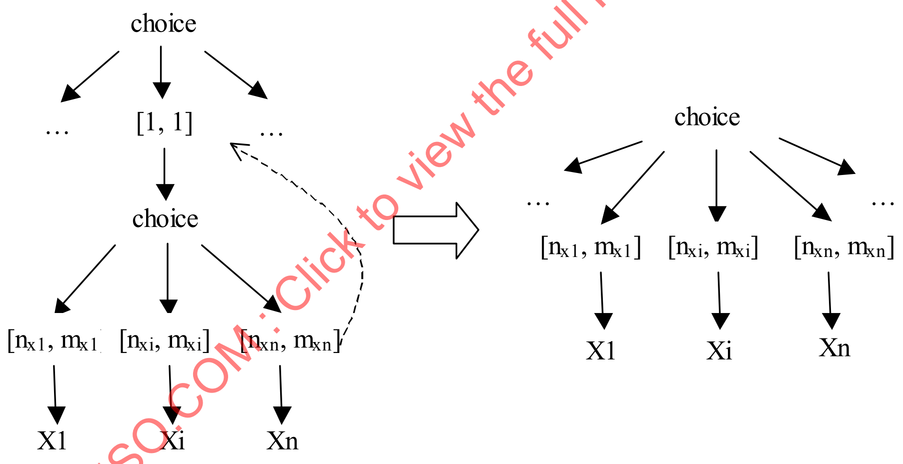

Figure 20 — Choice simplification rule

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 :

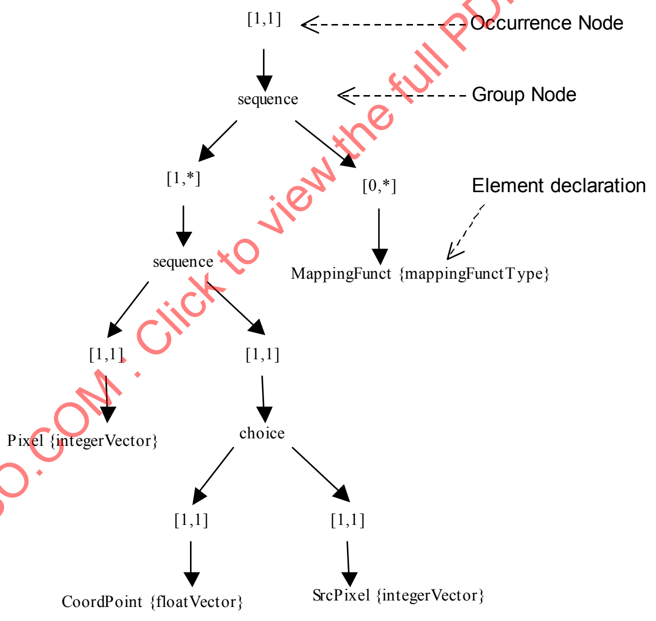

Figure 21 — Example - The syntax tree of coordinate mapping complexType

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.

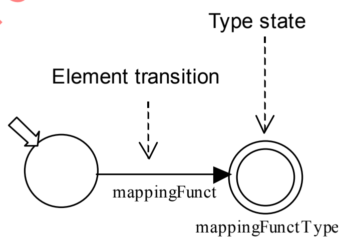

Figure 22 — Example of an element declaration node automaton

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.

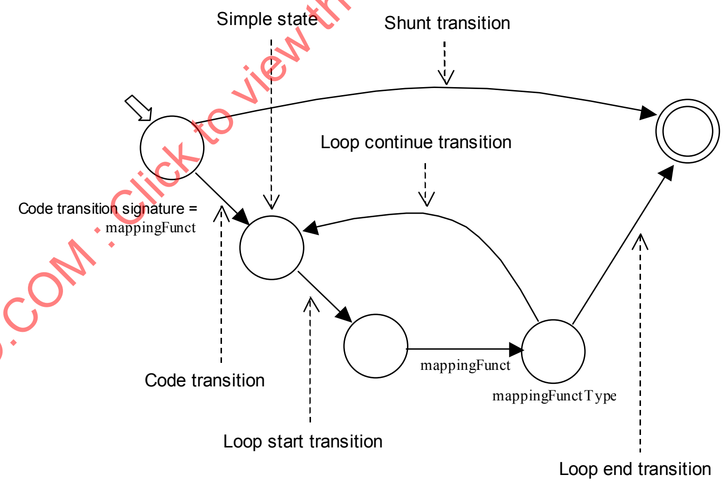

Figure 23 — Example of an occurrence node automaton

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.