INTERNATIONAL STANDARD

ISO/IEC 23001-1

First edition
2006-04-01

Information technology — MPEG systems technologies —

Part 1: Binary MPEG format for XML

Technologies de l'information — Technologies des systèmes MPEG —

Partie 1: Format binaire de MPEG pour XML

Reference number
ISO/IEC 23001-1:2006(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

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 2.

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 23001-1 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 23001 consists of the following parts, under the general title *Information technology — MPEG systems technologies*:

— *Part 1: Binary MPEG format for XML*

Introduction

This International Standard provides a standardized set of generic technologies for encoding XML documents. It addresses a broad spectrum of applications and requirements by providing generic methods for transmitting and compressing XML documents.

Part 1 – Binary Format for XML: specifies the tools for preparing XML documents for efficient transport and storage and for compressing XML documents.

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

The ISO and IEC take no position concerning the evidence, validity and scope of this patent right.

The holder of this patent right has assured ISO and IEC that he is willing to negotiate licences under reasonable and non-discriminatory terms and conditions with applicants throughout the world. In this respect, the statement of the holder of this patent right is registered with the ISO and IEC. Information may be obtained from the companies listed in Annex C.

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

Information technology — MPEG systems technologies —

Part 1:
Binary MPEG format for XML

1 Scope

This part of ISO/IEC 23001 provides a standardized set of technologies for encoding XML documents. It addresses a broad spectrum of applications and requirements by providing a generic method for transmitting and compressing XML documents.

This part of ISO/IEC 23001 specifies system level functionalities for the communication of XML documents. It provides a specification which will:

— enable the development of ISO/IEC 23001-1 receiving sub-systems, called ISO/IEC 23001-1 Terminal, or Terminal in short, to receive and assemble possibly partitioned and compressed XML documents

— provide rules for the preparation of XML documents for efficient transport and storage.

The decoding process within the ISO/IEC 23001-1 Terminal is normative. The rules mentioned provide guidance for the preparation and encoding of XML documents without leading to a unique encoded representation of such documents.

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

- ISO/IEC 10646:2003, *Information technology — Universal Multiple-Octet Coded Character Set (UCS)*

Note: The UTF-8 encoding scheme is described in Annex R of ISO/IEC 10646-1:1993, published as Amendment 2 of ISO/IEC 10646-1:1993.

- XML, *Extensible Markup Language (XML) 1.0,* October 2000.
- *XML Schema, W3C Recommendation*, 2 May 2001.
- *XML Schema Part 0: Primer,* W3C Recommendation, 2 May 2001.
- *XML Schema Part 1: Structures*, W3C Recommendation, 2 May 2001.
- *XML Schema Part 2: Datatypes*, W3C Recommendation 2 May 2001.
- XPath, *XML Path Language,* W3C Recommendation, 16 November 1999.
- *Namespaces in XML*, W3C Recommendation, 14 January 1999.

Note: These documents are maintained by the W3C (http://www.w3.org). The relevant documents can be obtained as follows:

- *Extensible Markup Language (XML) 1.0 (Second Edition),*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

- *RFC 2396, Uniform Resource Identifiers (URI): Generic Syntax.*
- *RFC 1950, ZLIB Compressed Data Format Specification version 3.3.*
- *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 and documents model, this part of ISO/IEC 23001 uses constructs specified in XML Schema, 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 XML Schema. 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 document stream retrieved by the decoder is specified in Clause 6, Clause 7, and Clause 8. Each data item in the binary document 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 6, 7 and 8 implement a complete decoding procedure. In particular, the binary syntax parsing in this specification assumes a correct and error-free binary document stream. Handling of erroneous binary document 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 document stream until but not including the next byte-aligned position in the binary document 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 document 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 document 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 document stream, for example to a byte boundary. The value of each of these bits in the binary document stream shall be “1”.

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

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

3.2.3
advanced optimised decoder
An optimised decoder used to decode a simple type. Advanced optimised decoders parameters and their mappings to types can be modified during binary document stream lifetime.

3.2.4
advanced optimised decoder instance
An advanced optimised decoder initialised and ready to be used for the decoding of some data types.

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

3.2.5
advanced optimised decoder instances table
A table of all the advanced optimised decoders available at a certain instant in time.

3.2.6
advanced optimised decoder parameters
The parameters of an advanced optimised decoder.

3.2.7
advanced optimised decoder type
The type, identified by a URI, of an advanced optimised decoder.

3.2.8
application
An abstraction of any entity that makes use of the decoded document stream.

3.2.9
binary access unit
An access unit in binary format as specified in Clause 6 and 7.

3.2.10
binary document stream
A concatenation of binary access units as specified in Clause 6 and 7.

3.2.11
binary format document tree
The internal binary decoder model.

3.2.12
byte-aligned
A bit in a binary document stream is byte-aligned if its position is a multiple of 8-bits from the first bit in the binary document stream.

3.2.13
composition time
The point in time when a specific access unit becomes known to the application.

3.2.14
content particle
A particle is a term in the XML Schema grammar for element content, consisting of an element declaration, a wildcard or a model group, together with occurrence constraints. Refers to XML SCHEMA.

3.2.15
context mode
Information in the fragment update context specifying how to interpret the subsequent context path information.

3.2.16
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.

3.2.17
context path
Information that identifies and locates the context node and the operand node in the current document tree.

3.2.18
contextual optimised decoder
"An optimised decoder which behavior is dependent on the current context of the decoding.

Note - For instance, the ZLib optimised decoder (see Clause 8) is a contextual optimised decoder.

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

3.2.19
contextual optimised decoder reset
An operation that resets the optimised decoder to put it in a defined initial state. All contextual information is discarded.

3.2.20
current context node
The starting node for the context path in case of relative addressing.

3.2.21
current document
The document that is conveyed by the initial document and all access units up to a given composition time.

3.2.22
current document tree
The XML document tree that represents the current document.

3.2.23
deferred fragment reference
A fragment reference that can be resolved at any time by the application using the terminal.

3.2.24
deferred node
A node which is present in the document tree at encoder side and for which the following is true: No part of that node has been sent to the decoder but the existence of that node has been signalled to the decoder.

3.2.25
delivery layer
An abstraction of any underlying transport or storage functionality.

3.2.26
derived type
A type defined by the derivation of an other type.

3.2.28
document
Short term for a structured XML document.

3.2.29
document composer
An entity that reconstitutes the current document tree from the fragment update units.

3.2.30
document fragment
A contiguous part of a document attached at a single node. Using the representation model of a document tree, the document fragment is represented by a sub-tree of the document tree.

3.2.31
document fragment reference
A reference to a document fragment.

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

3.2.32
document stream
The ordered concatenation of either binary or textual access units conveying a single, possibly time-variant, document.

3.2.33
document tree
A model that is used throughout this specification in order to represent documents. A document tree consists of nodes, which represent elements or attributes of an XML document. Each node may have zero, one or more child nodes. Simple content are considered as child nodes in Clause 6 of the specification.

3.2.34
effective content particle
The particle of a complexType used for the validation process.

3.2.35
fixed optimised decoder
An optimised decoder used to decode either a complex type or a simple type. Fixed optimised decoders are set up at decoder initialisation phase and their mapping to types can't be modified during binary document stream lifetime.

3.2.36
fragment reference
short term for document fragment reference.

3.2.37
fragment reference format
An encoding format of fragment references.

3.2.38
fragment reference marker
A specific information used to describe a deferred fragment reference, which is present within the current document tree. It consists of a fragment reference, the name and type of the top most element of the referenced fragment.

3.2.39
fragment reference resolver
An entity that is capable of resolving the fragment reference provided in the fragment update payload.

3.2.40
fragment update command
A command within a fragment update unit expressing the type of modification to be applied to the part of the current document tree that is identified by the associated fragment update context.

3.2.41
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.

3.2.42
fragment update context
Information in a fragment update unit that specifies on which node in the current document 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.

3.2.43
fragment update decoder parameters
Configuration parameters conveyed in the DecoderInit (see 6.2) that are required to specify the decoding process of the fragment update decoder.

3.2.44
fragment update payload
Information in a fragment update unit that conveys the information which is added to the current document or which replaces a part of the current document.

3.2.45
fragment update payload decoder
The entity that decodes the fragment update payload information of the fragment update.

3.2.46
fragment update unit
Information in an access unit, conveying a document or a portion thereof. Fragment update units provide the means to modify the current document. They are nominally composed of a fragment update command, a fragment update context and a fragment update payload.

3.2.47
initial document
A document that initialises the current document tree without conveying it to the application (see 5.3). The initial document is part of the DecoderInit (see 6.2).

3.2.48
initial schema
The schema that is known by the decoder before the decoding process starts.

3.2.49
initialisation extractor
An entity that de-multiplexes the DecoderInit (see 6.2), resulting in its components initial document, fragment update decoder parameters and schema URI.

3.2.51
non-deferred fragment reference
A fragment reference that shall be resolved by the terminal at the composition time of the access unit containing the fragment reference.

3.2.52
operand node
The node in the binary format document 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.

3.2.53
optimised decoder
A decoder associated to a type and dedicated to certain encoding methods better suited than the generic ones.

3.2.54
optimised decoder mapping
An association between a type and a set of optimised decoders.

3.2.55
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 XML Schema.

3.2.56
schema resolver
An entity that is capable of resolving the schema identification provided in the DecoderInit (see 6.2), and to possibly retrieve the specified schemas.

3.2.57
schema update unit
Information in an access unit, conveying a schema or a portion thereof. Schema update units provide the means to modify the current decoder schema knowledge.

3.2.58
schema URI
A URI that uniquely identifies a schema.

3.2.59
schema valid
A document that is schema valid satisfies the constraints embodied in the Schema to which it should conform.

3.2.60
selector node
The parent node of the topmost node of a document tree. It artificially extends the document tree to allow the addressing of the topmost node.

3.2.61
skippable subtree
A subtree of an XML document that the decoder is permitted not to decode.

3.2.62
super type
The parent of a type in its type hierarchy.

3.2.63
systems layer
An abstraction of the tools and processes specified in this specification.

3.2.64
terminal
The entity that makes use of a coded representation of a document.

3.2.67
topmost node
The node specified by the first element in the document, instantiating one of the global elements declared in the schema.

3.2.68
type codec
Synonym to optimised decoder.

3.2.69
type hierarchy
The hierarchy of type derivations.

3.2.70
validation
The process of parsing an XML document to determine whether it satisfies the constraints embodied in the Schema to which it should conform.

3.2.71
XML Schema parser
An application that is capable of validating document schemes (content and structure) and descriptor data types against their schema definition.

4 Symbols and abbreviated terms

4.1 Abbreviations

AU	Access Unit
BiM	Binary format for document streams
D	Descriptor
DL	Delivery Layer
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
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UTF	Universal Character Set Transformation Formats
XML	Extensible Markup Language
XPath	XML Path Language
MSB	Most Significant Bit
SU	Schema Update
SUU	Schema Update Unit

4.2 Mathematical operators

The mathematical operators used to describe this specification 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 (x)	denotes the smallest integer larger than or equal to x.
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 specification. 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.
vlurmsbf5	Variable length code unsigned rational number, most significant bit first. The first n bits (Ext) which are ‘1’ except of the nth bit which is ‘0’, indicate that the rational number R in the interval 0≤R<1 is encoded by n times 4 bits. The ith bit of the n times 4 bits representing the rational number corresponds to a value of 2^-i. Thus the (n+1)st bit of the vlurmsbf5 code word (which corresponds to the MSB of the rational number) represents a value of ½, the (n+2)nd bit of the vlurmsbf5 code word represents a value of ¼., and so forth. An example for this type is shown in Figure 3. Note - Comparing two rational numbers A and B represented by a vlurmsbf5 code word can be done by comparing bit by bit the rational numbers starting from their respective MSBs. Then the rational number A is bigger if there is a ‘1’ bit at a position at which there is a ‘0’ for B. A is also bigger if there is a ‘1’ bit at a position which is not present for B and when A is longer than B.

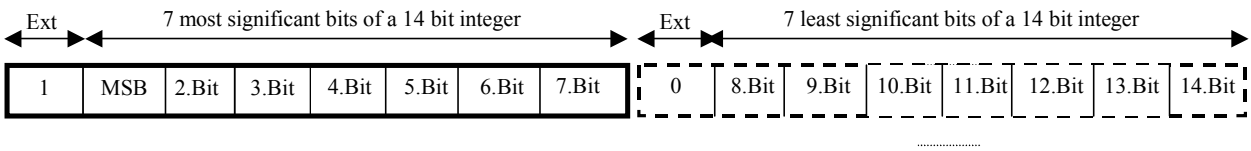

Figure 1 — Informative example for the vluimsbf8 data type

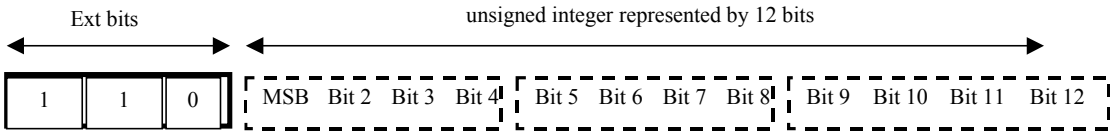

Figure 2 — Informative example for the vluimsbf5 data type

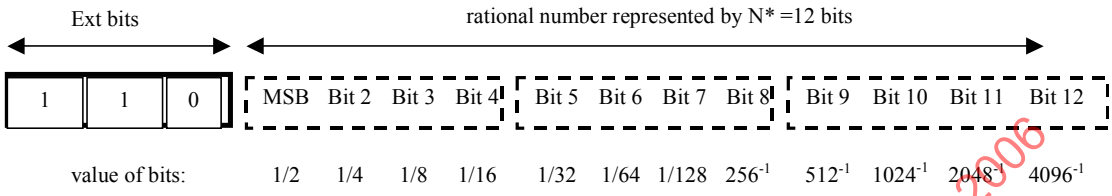

Figure 3 — Informative example for the vlurmsbf5 data type

5 System architecture

5.1 Terminal architecture

ISO/IEC 23001-1 provides the means to represent coded XML documents. The entity that makes use of such coded representations of XML documents is generically referred to as the “ISO/IEC 23001-1 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 23001-1 terminal, its components, and their operation. The architecture of such a terminal is depicted in Figure 4. The following subclauses introduce the tools specified in this specification.

In Figure 4, there are three main layers outlined: the application, the normative systems layer, and the delivery layer. ISO/IEC 23001-1 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 document. This specification does make specific assumptions about the delivery layer, and those assumptions are outlined in subclause 5.5.4. The systems layer 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 4, 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 document streams

An ISO/IEC 23001-1 terminal consumes document streams and outputs a – potentially dynamic – representation of the document called the current document tree. Document 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 document tree corresponding to a specific access unit becomes known to the application. The timing information shall be carried by the delivery layer (see subclause 5.6). The current document tree shall be schema-valid after processing each access unit.

A document stream consists of binary access units is termed a binary document stream and is processed by a binary decoder (see subclause 5.2.2 and Clauses 6 and 7).

5.2.2 Principles of the binary decoder

Using the ISO/IEC 23001-1 generic method for binary encoding, called BiM, a document (nominally in a textual XML form) can be compressed, partitioned, streamed, and reconstructed at terminal side. The reconstructed XML document will not be byte-equivalent to the original document. 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 schema components (XML elements, types and attributes). BiM defines two methods to address schema components.

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

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

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

To further improve compression, BiM allows the association of specific codecs to specific data types instead of using the generic mechanisms defined in Clause 7. These encoding schemes can be optimised with the full knowledge of the properties of that data type.

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

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

5.3 Sequence of events during decoder initialisation

The decoder set-up is signalled by the initialisation extractor receiving a binary `DecoderInit` (specified in 6.2). 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 document stream, which is also received from the delivery layer. The component parts of the document stream are discussed in subclause 5.4.

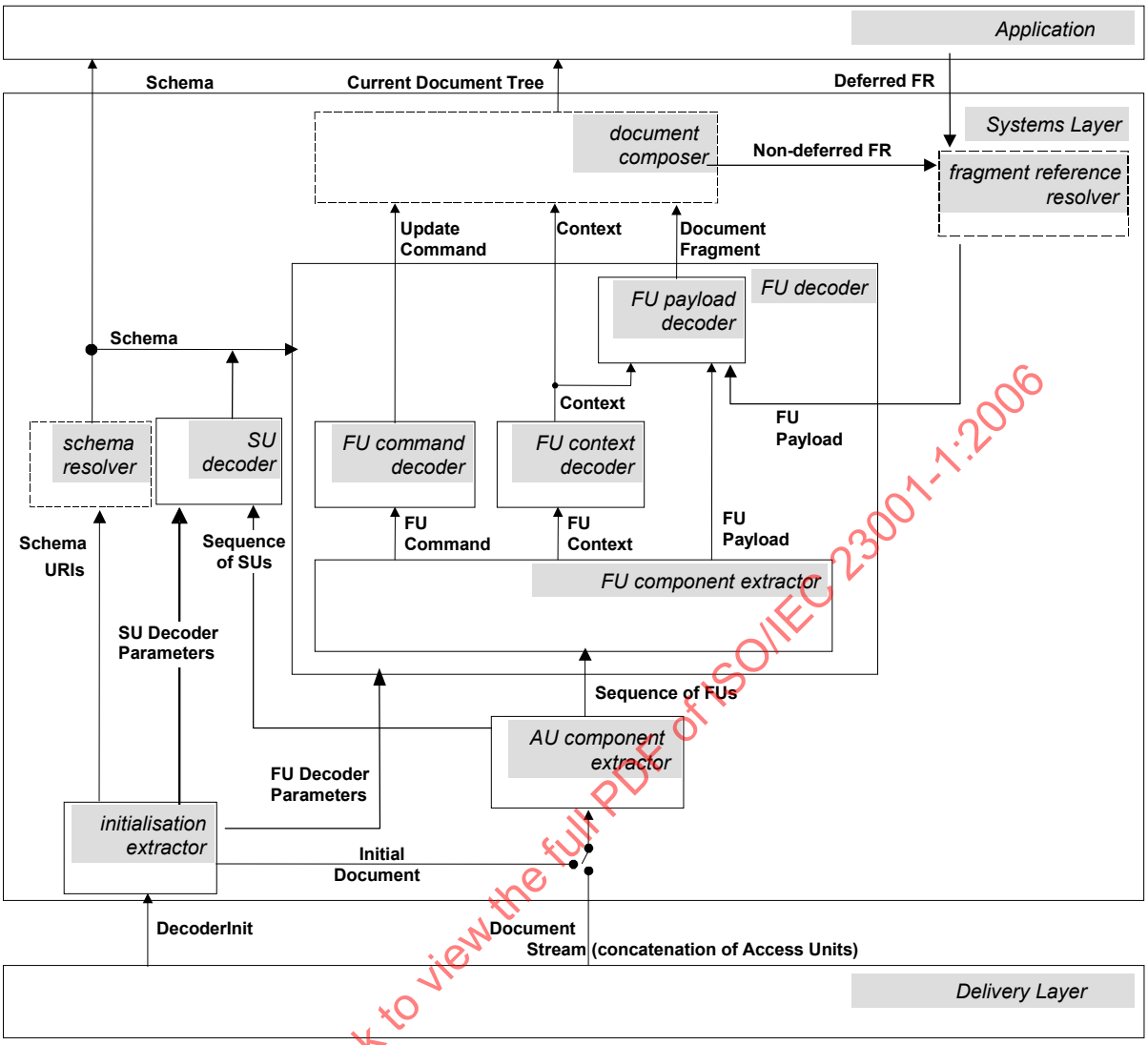

Figure 4 — Terminal Architecture.
Dashed boxes in the systems layer are non-normative. FU is an abbreviation for Fragment Update.

The `DecoderInit` contains a list of URIs that identifies schemas, miscellaneous parameters to configure the decoder (FU Decoder Parameters and SU Decoder Parameters, in Figure 4), and an initial document. There shall be only one `DecoderInit` per document stream. The list of URIs (Schema URIs, in Figure 4) is passed to a schema resolver that associates the URIs with schemas to be passed into the fragment update decoder (see subclause 6.2). 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 6 and 7 and by any textual XML Schema 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 document stream shall be ignored (i.e., “skipped” and not processed).

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

5.4 Decoder behaviour

The document 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.”

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

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

- a namespace identifier,
- a set of code tables to represent global elements, global types and global attributes,
- a binary encoded schema carrying the schema components definitions.

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

Some further constraints are applied to the acquisition of schema update units, notably to ensure that a decoder will not break in case of a missed schema update unit. A specific schema update unit, the so-called first schema update unit, contains initialization information and shall be acquired by the decoder before any use of a received definition. The decoder behavior in case of such missed schema update units is not normative. A transmitted schema definition shall not change during binary document stream lifetime and there shall not be two schema identifiers associated to the same namespace. Finally, all the optimised decoders associated to existing types are immediately applied to all types they derive from in accordance to the rules defined for the optimized decoders in Clause 8.

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

Fragment update units are 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 document tree);
- a fragment update context that identifies the data type in a given schema document, and points to the location in the current document tree where the fragment update command applies; and
- a fragment update payload conveying either the coded document fragment (extracted out of the original document) to be added or replaced, or a reference to it.

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 document composer. The decoded fragment update context information (‘context’ in Figure 4) is passed along to both the document composer and the fragment update payload decoder. The fragment update payload decoder embodies the BiM Payload decoder (Clause 7), which decodes a fragment update payload (aided by context information) to yield a document fragment (see Figure 4).

The corresponding update command and context are processed by the non-normative document composer, which either places the document fragment extracted out of the original document or a reference to it received from the fragment update payload decoder at the appropriate node of the current document tree, or sends a reconstruction event containing this information to the application. If the payload consists of a fragment reference, depending on its nature, the referenced fragment is either immediately acquired (non-deferred

fragment reference) or its acquisition is left to the application (deferred fragment references). In case of a deferred fragment reference, a fragment reference marker is available to the application to help further acquisition. This marker consists of the fragment reference itself, the name and type of the top most element of the referenced fragment. The fragment reference marker is added to the current document tree at the location defined by the fragment update context.

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

5.5 Issues in encoding documents

5.5.1 Fragmenting documents

A document stream serves to convey an XML document, 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 document may be possible and it is out of scope of this specification to define such decompositions. Figure 5 illustrates an example of a document, consisting of a number of nodes, that is broken into two document fragments.

If multiple document fragments corresponding to a specific node of the document are sent (e.g., a node is replaced) then the previous data within the nodes of the document represented by that document fragment become unavailable to the terminal. Replacing a single node of the document 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 document is outside the scope of this specification.

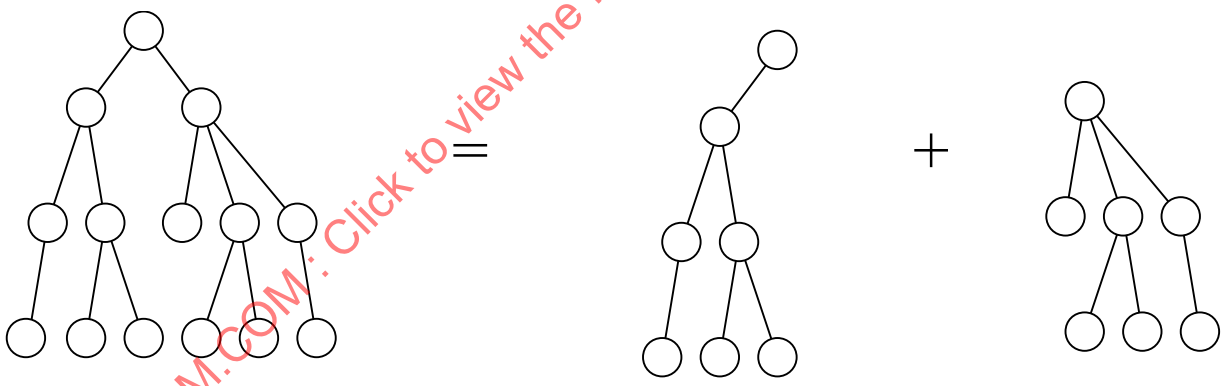

Figure 5 — Decomposition of a document into two document fragments

5.5.2 Deferred nodes, fragment references and their use

With BiM, there exists the possibility for the encoder to indicate that a node in the current document 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 document 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 document 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 or on application demand 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 document stream.

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

5.5.3 Managing schema version compatibility with ISO/IEC 23001-1

It is very conceivable that a given schema will be updated during its lifetime. Therefore, ISO/IEC 23001-1 provides, with some constraints, interoperability between different versions of ISO/IEC 23001-1 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 document 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 document 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 6 and 7. 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 document related to different schema in separate chunks of the binary document 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 23001-1 “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 document stream.

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

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 document 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 Characteristics of the delivery layer

The delivery layer is an abstraction that includes functionalities for the synchronization, framing and multiplexing of document streams with other data streams. Document streams may be delivered independently or together with the described multimedia content. No specific delivery layer is specified or mandated by ISO/IEC 23001-1.

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 document 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 document stream nor is it relevant for the usage of the reconstructed document. The composition time is understood to be “best effort,” and the order of decoding AUs, if prescribed by the

producer of the document, shall be preserved. Note, however, that this in no way precludes time related information to be present within the document.

A delivery layer (DL) suitable for conveying ISO/IEC 23001-1 document streams shall have the following properties:

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

— The DL shall provide a mechanism by which at least one entry point to the document 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 document streams, the DL shall provide a suitable random access mechanism.

— The DL shall provide delineation of the access units within the document 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 document 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 subclause 6.2) 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 document 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.

5.7 Decoding of Fragment References

5.7.1 Decoding of Non-Deferred Fragment References

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

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

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

5.7.2 Decoding of Deferred Fragment References

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

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

6 Binary format- BiM

6.1 Overview

The following subclauses specify the syntax elements and associated semantics of the binary format for ISO/IEC 23001-1 documents, abbreviated BiM. The binary DecoderInit (6.2), the binary access unit (6.3), the binary fragment update unit (6.4) and its constituent parts, the binary fragment update command (6.5), binary fragment update context (6.6) and binary schema update unit (6.7) are covered by Clause 6. The specification of the binary fragment update payload follows in Clause 7.

Identifying schema components in the BiM framework

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

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

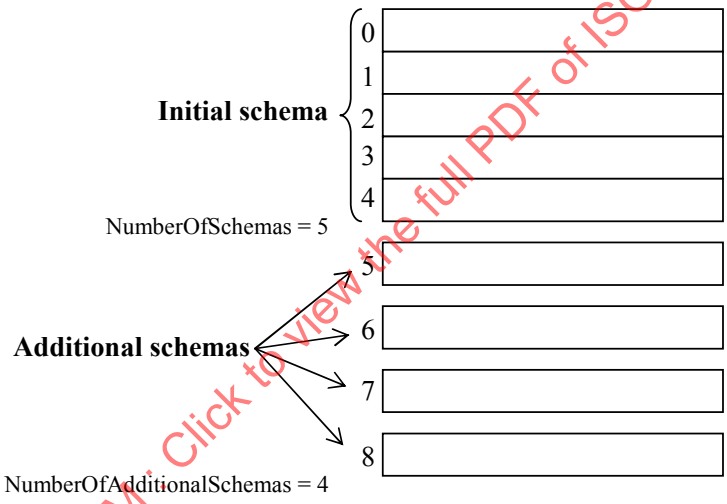

Figure 6 — Addressing the initial schema and the additional schemas

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

6.2 Binary DecoderInit

6.2.1 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 document stream.

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

An optimised type codec specifies an encoding for a data type not using the generic mechanism specified in Clause 7 but optimised with full knowledge of the properties of that data type. There are two kinds of optimised type codec (or optimised decoders). A fixed optimised decoder associates a specific encoding scheme to a type of the schema (complex as well as simple) and this association is fixed for the entire stream. An advanced optimised decoder associates a specific encoding scheme to any simple type and this association can be changed during the transmission of the bitstream. Moreover, several advanced optimised decoders can be associated to a single type and can accept parameters. Some fixed optimised type codecs are specified in ISO/IEC 15938-3. Some advanced optimised decoders are defined in Clause 8.

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

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

The fixed length context path mechanism provides a simplified addressing of nodes for usage scenarios where only a limited number of nodes need to be addressed. This is done by a table that uniquely maps fixed length codes to full context paths.

6.2.2 Syntax

DecoderInit () {	**Number of bits**	**Mnemonic**
SystemsProfileLevelIndication	8+	vluimsbf8
UnitSizeCode	3	bslbf
NoAdvancedFeatures	1	bslbf
ReservedBits	4	bslbf
If (! NoAdvancedFeatures) {		
AdvancedFeatureFlags_Length	8+	vluimsbf8
/** FeatureFlags **/		
InsertFlag	1	bslbf
AdvancedOptimisedDecodersFlag	1	bslbf
AdditionalSchemaFlag	1	bslbf
AdditionalSchemaUpdatesOnlyFlag	1	bslbf
FragmentReferenceFlag	1	bslbf
MPCOnlyFlag	1	bslbf
HierarchyBasedSubstitutionCodingFlag	1	bslbf
ContextPathTableFlag	1	bslbf
ReservedBitsZero	FeatureFlags_Length*8-8	bslbf
/** FeatureFlags end **/		
}		
*/** Start FUUConfig **/*		

If (! AdditionalSchemaUpdatesOnlyFlag) {		
NumberOfSchemas	8+	vluimsbf8
for (k=0; k< NumberOfSchemas; k++) {		
SchemaURI_Length[k]	8+	vluimsbf8
SchemaURI[k]	8* SchemaURI_Length[k]	bslbf
LocationHint_Length[k]	8+	vluimsbf8
LocationHint[k]	8* LocationHint_Length[k]	bslbf
NumberOfTypeCodecs[k]	8+	vluimsbf8
for (i=0; i< NumberOfTypeCodecs[k]; i++) {		
TypeCodecURI_Length[k][i]	8+	vluimsbf8
TypeCodecURI[k][i]	8* TypeCodecURI_Length[k][i]	bslbf
NumberOfTypes[k][i]	8+	vluimsbf8
for (j=0; j< NumberOfTypes[k][i]; j++) {		
TypeIdentificationCode[k][i][j]	8+	vluimsbf8
}		
}		
}		
If (**ContextPathTableFlag**) {		
ContextPathTable()		
}		
*/** FUUConfig - Advanced optimised decoder framework **/*		
If (AdvancedOptimisedDecodersFlag) {		
NumOfAdvancedOptimisedDecoderTypes	8+	vluimsbf8
for (i=0; i< NumOfAdvancedOptimisedDecoderTypes; i++) {		
AdvancedOptimisedDecoderTypeURI_Length[i]	8+	vluimsbf8
AdvancedOptimisedDecoderTypeURI[i]	8* AdvancedOptimisedDecoderTypeURI_Length[i]	bslbf
}		
AdvancedOptimisedDecodersConfig ()		
}		
*/** FUUConfig - Fragment reference framework **/*		
If (FragmentReferenceFlag) {		
NumOfSupportedFragmentReferenceFormat	8	uimsbf
for (i=0;i< NumOfSupportedFragmentReferenceFormat;i++) {		
SupportedFragmentReferenceFormat[i]	8	blsbf
}		

}		
}		
/** *end FUUConfig* **/		
If (AdditionalSchemaFlag) {		
AdditionalSchemaConfig ()		
}		
/** *Initial document* **/		
If (!AdditionalSchemaUpdateOnlyFlag) {		
InitialDocument_Length	8+	vluimsbf8
InitialDocument()		
}		
}		

ContextPathTable {		
ContextPathTable_Length	8+	vluimsbf8
ContextPathCode_Length	8+	vluimsbf8
NumberOfContextPaths	8+	vluimsbf8
CompleteContextPathTable	1	bslbf
for(i=0;i<NumberOfContextPaths;i++){		
ContextPath_Length[i]	5+	vluimsbf5
ContextPath()[i]	ContextPath_Length[i]	
If(!CompleteContextPathTable){		
ContextPathCode[i]	ContextPathCode_Length	bslbf
}		
}		
nextByteBoundary()		
}		

Semantics

Name	*Definition*
SystemsProfileLevelIndication	Indicates the profile and level as defined in ISO/IEC 23001-1 to which the document 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 7.
NumberOfSchemas	Indicates the number of schemas on which the document stream is based. These schemas compose the initial schema. 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 document 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 document. The `SchemaID` (see 6.6.3 and 7.4.4) as well as `SchemaIDOfSubstitution` (see 7.4.3) 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. 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 document stream. The syntax and semantics of `TypeIdentificationCode[k][i][j]` is the same as of the type identification code defined in subclause 6.6.5.4 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.
NumOfAdvancedOptimisedDecoderTypes	Defines the number of advanced optimised decoder types that are necessary to properly decode the binary document stream.
AdvancedOptimisedDecoderTypeURI_Length[i]	Indicates the size in bytes of the `AdvancedOptimisedDecoderTypeURI[i]` syntax element.
AdvancedOptimisedDecoderTypeURI[i]	Defines the UTF-8 representation of the URI referencing the advanced optimised decoder type with index i.
AdvancedOptimisedDecodersConfig()	See subclause 8.2.
NumOfSupportedFragmentReferenceFormat	Specifies the number of fragment reference format that shall be supported by the decoder.
SupportedFragmentReferenceFormat[i]	Specifies the i^{th} fragment reference format, according to Table 3, that shall be supported by the decoder. The `SupportedFragmentReferenceFormat[0]` indicates the default fragment reference format.
AdditionalSchemaConfig()	See subclause 6.2.3.
AdditionalSchemaUpdatesOnlyFlag	Signals that the document stream contains only additional schema updates i.e. no fragment update units. The AdditionalSchemaFlag shall be set to true when this flag is set to true.
FragmentReferenceFlag	Signals that fragment references are supported.
MPCOnlyFlag	Signals that position codes in the fragment update context are encoded in MPC mode only.
HierarchyBasedSubstitutionCodingFlag	Signals that element substitution codes are computed taking into account their substitution hierachy. If additional schemas are supported (i.e. AdditionalSchemaFlag==true) this flag shall be set to true.
ContextPathTableFlag	Signals the presence of a context path table in the decoderInit.

ContextPathTable_Length	Defines the number of bytes used for the indication of the ContextPathTable. Note – This length provides a simple framework to skip the table.
ContextPathCode_Length	Signals the length of the context path codes in number of bits.
NumberOfContextPaths	Signals the number of ContextPaths contained in the ContextPathTable.

CompleteContextPathTable	Signals if the ContextPathTable is complete and ordered according to the assignment of ContextPathCodes. If CompleteContextPathTable is set to '1' the ContextPathCodes are assigned in the order the ContextPaths as specified in the ContextPathTable starting from '1'. If CompleteContextPathTable is set to '0' the ContextPathCodes are assigned explicitly. The ContextPathCode '0' is reserved.
ContextPath_Length	Signals the number of bits used for the following ContextPath[i]()
ContextPath[i]()	Signals the ContextPath as specified in subclause 6.6.2 with the following restrictions: - ContextModeCode is set to '001' - PositionCode() is an empty bitfield
ContextPathCode[i]	Signals the ContextPathCode of ContextPath[i]

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

Table 3 — Fragment Reference Formats

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

6.2.3 Syntax of AdditionalSchemaConfig

AdditionalSchemaConfig () {	**Number of bits**	**Mnemonic**
NumberOfAdditionalSchemas	8+	vluimsbf8
NumberOfKnownAdditionalSchemas	8+	vluimsbf8
for (int t=1;t<NumberOfKnownAdditionalSchemas-1;t++){		
KnownAdditionalSchemaID	8+	
AdditionalSchemaURI_Length[KnownAdditionalSchemaID]	8+	vluimsbf8
AdditionalSchemaURI[KnownAdditionalSchemaID]	8* AdditionalSchemaURI_Length[KnownAdditionalSchemaID]	bslbf
BinaryLocationHint_Length[KnownAdditionalSchemaID]	8+	vluimsbf8
BinaryLocationHint[KnownAdditionalSchemaID]	8*BinaryLocationHint_Length[KnownAdditionalSchemaID]	bslbf
NumberOfTypeCodecs[KnownAdditionalSchemaID]	8+	vluimsbf8
for (i=0; i< NumberOfTypeCodecs[KnownAdditionalSchemaID]; i++) {		
TypeCodecURI_Length[KnownAdditionalSchemaID][i]	8+	vluimsbf8
TypeCodecURI[KnownAdditionalSchemaID][i]	8* TypeCodecURI_Length[KnownAdditionalSchemaID][i]	bslbf
NumberOfTypes[KnownAdditionalSchemaID][i]	8+	vluimsbf8
for (j=0; j< NumberOfTypes[KnownAdditionalSchemaID][i]; j++) {		
TypeIdentificationCode[KnownAdditionalSchemaID][i][j]	8+	vluimsbf8
}		
}		
ExternallyCastableTypeTable(KnownAdditionalSchemaID)		
ExternallySubstitutableElementTable(KnownAdditionalSchemaID)		
}		
SchemaEncodingMethod	8	blsbf
ExternallyCastableTypeTable(InitialSchema)		
ExternallySubstitutableElementTable(InitialSchema)		
ReservedBitsZero	7	blsbf
}		

Name	*Definition*
NumberOfAdditionalSchemas	Indicates the number of schemas that can be transmitted and that are not declared in the list of `schemaURI`. If additional schemas are not supported, this value is set to zero. If additional schemas are supported then the value of GlobalSchemaID = `NumberOfSchemas` is reserved for a virtual namespace for attributes that do not belong to any namespace. The value of GlobalSchemaID = `NumberOfSchemas + NumberOfAdditionalSchemas – 1` is reserved for ISO usage.
NumberOfKnownAdditionalSchemas	Indicates the number of additional schemas that are known to be updated in the bitstream.
KnownAdditionalSchemaID	Identifies a schema known to be updated in the bitstream. This identifier shall only address an additional schema. The value KnownAdditionalSchemaID = `NumberOfSchemas` shall be reserved for attributes that do not belong to any namespace. The value of KnownAdditionalSchemaID = `NumberOfSchemas + NumberOfAdditionalSchemas – 1` shall be reserved for ISO Use.
AdditionalSchemaURI_Length[KnownAdditionalSchemaID]	Indicates the size in bytes of the `AdditionalSchemaURI[KnownAdditionalSchemaID]` length. A value of zero is forbidden.
AdditionalSchemaURI[KnownAdditionalSchemaID]	Indicates the UTF-8 representation of the URI of the additional schema identified by `KnownAdditionalSchemaID`. Note – This field allows to identify some of the additional schemas that are expected to be updated. This information allows one decoder not to monitor the schema updates for which it already knows the schema.
BinaryLocationHint_Length[KnownAdditionalSchemaID]	Indicates the size in bytes of the `BinaryLocationHint_Length[KnownAdditionalSchemaID]`. A value of zero indicates that for the schema that is referenced by the index `KnownAdditionalSchemaID` there is no binary encoded schema available.
BinaryLocationHint[KnownAdditionalSchemaID]	This is the UTF-8 representation of the URI to unambiguously reference the location of the binary encoded schema that is referenced by the index `KnownAdditionalSchemaID`. The schema can be fetched by the schema resolver and is then received as a document stream composed only of schema update units i.e. for which the `SchemaOnlyFlag` is set to true.
NumberOfTypeCodecs[KnownAdditionalSchemaID]	see `NumberOfTypeCodecs[k]` in 6.2.2.
TypeCodecURI_Length[KnownAdditionalSchemaID]	see `TypeCodecURI_Length[k]` in 6.2.2.
TypeCodecURI[KnownAdditionalSchemaID][i]	see `TypeCodecURI[k][i]` in 6.2.2.
NumberOfTypes[KnownAdditionalSchemaID][i]	see `NumberOfTypes[k][i]` in 6.2.2.

TypeIdentificationCode[KnownAdditionalSchemaID][i][j]	see `TypeIdentificationCode[k][i][j]` in 6.2.2.
SchemaEncodingMethod	Indicates the encoding method of the schema update units.
ExternalCastableTypeTable	Defines the types of the initial schema that are externally castable as defined in subclause 6.7.5.4 and 6.7.5.5.
ExternalSubstitutableElementTable	Defines the elements of the initial schema that are substitutable as defined in subclause 6.7.6.4 and 6.7.6.5.

Table 4 — Schema encoding method

SchemaEncodingMethod	***definition***
0	ISO reserved
1	BiM encoded schema as described in subclause 6.7.8
2-224	ISO reserved
225-255	Private use

6.3 Binary Access Unit

6.3.1 Overview

A binary access unit is composed of one or more binary fragment update units that represent one or more document fragments. Therefore, an access unit may convey updates for several distinct parts of a document simultaneously. 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 document composer in the order specified within the access unit. Syntax and semantics of a fragment update unit are described in subclause 6.4.

6.3.2 Syntax

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

6.3.3 Semantics

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

6.4 Binary Fragment Update Unit

6.4.1 Overview

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

The topmost node is the node corresponding to the first element in the document. 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 7 shows an example document tree.

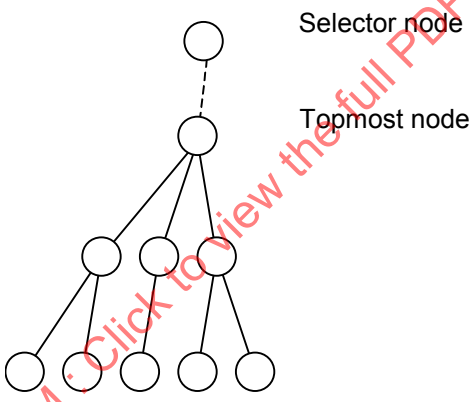

Figure 7 — Example for the tree representation of a document

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

The binary format document 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 document 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 document 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 document tree is defined, immediately after the decoding of any AU, as the set of instantiated nodes in the binary format document 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 document tree, i.e. if a document fragment shall be added, replaced or deleted or if the complete binary format document tree shall be reset;

— the fragment update context signals on which node in the binary format document 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 document 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.

6.4.2 Syntax

FragmentUpdateUnit () {	**Number of bits**	**Mnemonic**
FUU_Length	8+	vluimsbf8
FragmentUpdateCommand	4	bslbf
If (AdvancedOptimisedDecodersFlag){		
OptimisedDecoderReparameterization	2	bslbf
if (OptimisedDecoderReparameterization == '00') {		
AdvancedOptimisedDecodersConfig ()		
}		
}		
if (FragmentUpdateCommand != ‘0100’) {		
/ ‘0100’ corresponds to “Reset” */*		
FragmentUpdateContext()		
if (FragmentUpdateCommand != ‘0011’) {		
/ ‘0011’ corresponds to “DeleteContent” */*		
for (i=0;i<NumberOfFragmentPayloads;i++) {		
FragmentUpdatePayload(startType)		
}		
}		
}		
nextByteBoundary()		
}		

6.4.3 Semantics

Name	*Definition*
FUU_Length	Indicates the length in bytes of the remainder of this fragment update unit (excluding the `FUU_Length` syntax element).
OptimisedDecodersReparameterization	This 2-bit flag signals if the parameters of the optimised decoders shall be updated. It can take the following values: — '00' – the optimised decoder instance table and mappings shall be redefined; — '01' – the optimised decoder instance table and mappings shall not be redefined; — '10' – the optimised decoder instance table and mappings are reset to the default table and mappings defined in the `DecoderInit`; '11' – reserved.
AdvancedOptimisedDecodersConfig()	See subclause 8.2.
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 7.3.1. Its value is derived from the Operand_TBC in the `FragmentUpdateContext` as specified in 6.6.
NumberOfFragmentPayloads	The value of this internal variable is derived from `FragmentUpdateContext` as specified in 6.6.
FragmentUpdatePayload()	See 7.3.1

6.5 Binary Fragment Update Command

The `FragmentUpdateCommand` code word specifies the command that shall be executed on the binary format document tree. Table 5 defines the code words and the semantics of the fragment update command.

Table 5 — Code Table of fragment update commands

Code Word	Command Name	Specification
0000	---	Reserved
0001	AddContent	Add the document fragment contained in the fragment update payload at the node indicated by the operand node (see 6.6). 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 document tree this is equivalent to either appending or inserting the corresponding nodes.
0010	ReplaceContent	Replace the document fragment at the node indicated by the operand node with document 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 document tree this is equivalent to replacing the corresponding node.
0011	DeleteContent	Delete the document 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 document tree this is equivalent to deleting the corresponding node.
0100	Reset	Reset the complete binary format document tree, i.e. revert all nodes in the binary format document tree to “not instantiated” and decode the `InitialDocument` conveyed in the `DecoderInit`. After processing a reset command the current context is set to the topmost node of the current document tree. Note This is equivalent to deleting the complete current document tree.
0101 – 1111	---	Reserved

6.6 Binary Fragment Update Context

6.6.1 Overview

The fragment update context specifies on which node of the binary format document 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 document 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 8).

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 6.6.5.

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 6.6.5.2),

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 6.6.5.3),

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 6.6.5.4), and

4) a Position Code which is used if multiple occurrences of the element addressed by the SBC are possible (see 6.6.5.5).

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 8 shows the principle of absolute and relative addressing.

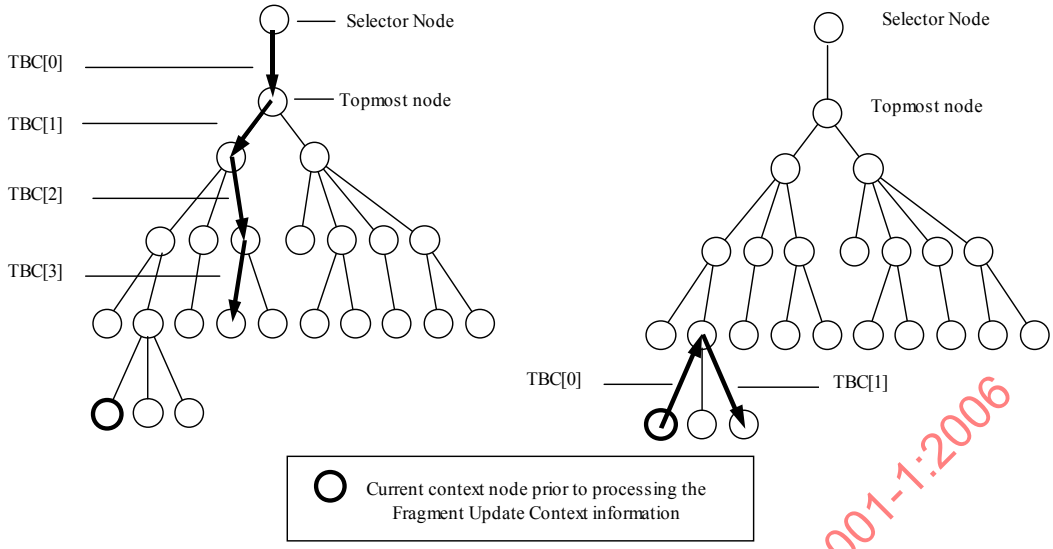

Figure 8 — Absolute (left) and relative context path example

A `ContextModeCode` (6.6.4) 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 6.6.4 and 6.6.5.6.

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 document tree and move upwards to the parent node. The OperandTBC table additionally contains also the references to the attributes and either to the elements of simpleType or to a simple content, but does not contain the Path Termination Code nor the reference to the parent node. Furthermore, in the OperandTBC table one TBC is reserved for User Data Extension as defined in subclause 6.6.5.2. In case of a mixed content model the OperandTBC table also contains a reference to the character data that may appear between the elements. Example TBC tables are shown in Table 6 and Table 7.

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 9.

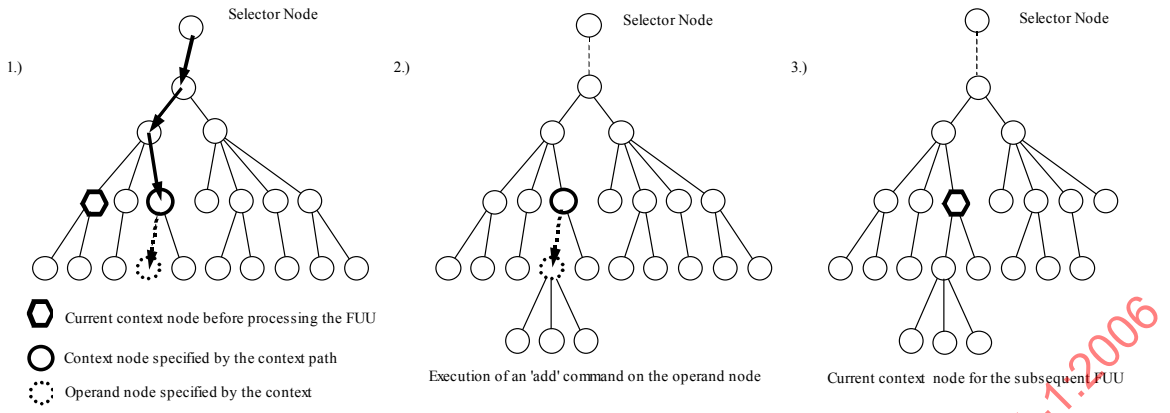

Figure 9 — Example of Context node and operand node during the execution of a fragment update unit

The ContextTBC and OperandTBC tables are generated automatically from the schema as specified in 6.6.5 and, hence, do not appear in this specification. Table 6 and Table 7 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. In this example the content model of the complex type definition is not 'mixed'.

Table 6 — 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 7 — Example of an 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 document tree when using a relative addressing mode.

In order to support efficient searching and filtering the document 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 10.

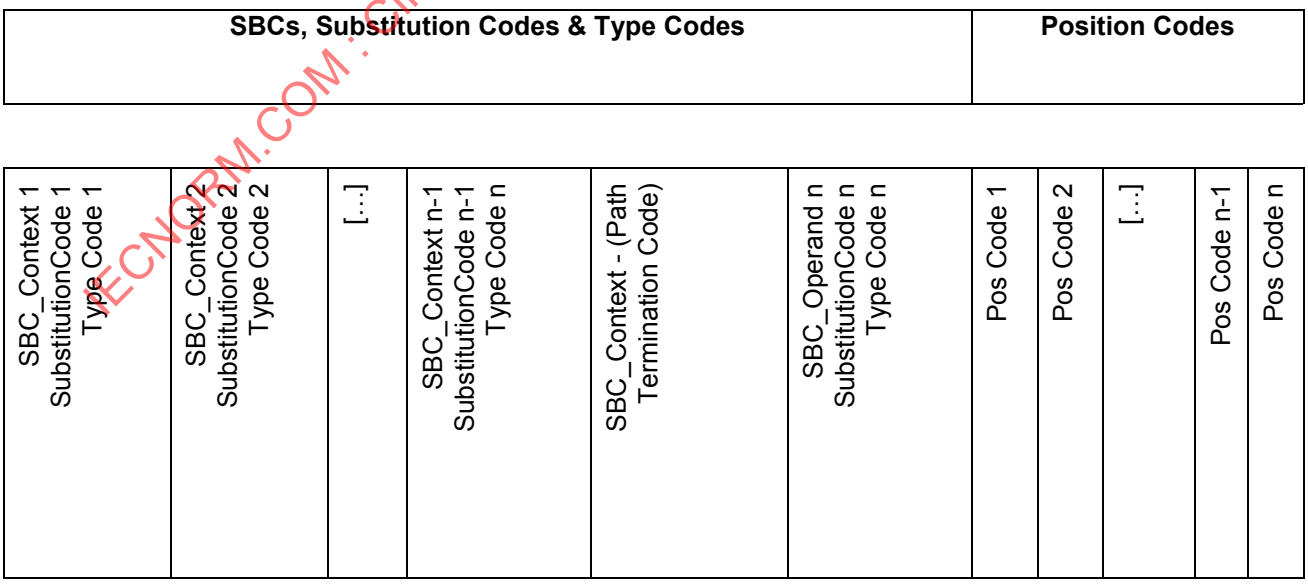

Figure 10 — Example of the structure of a Context Path

6.6.2 Syntax

FragmentUpdateContext () {	**Number of bits**	**Mnemonic**
SchemaID	ceil(log2(`NumberOfSchemas + NumberOfAdditionalSchemas`))	uimsbf
ContextModeCode	3	bslbf
If (ContextModeCode==’101’) {		
ContextPathCode	ContextPathCode_Length	bslbf
for (i=0; i < TBC_Counter(ContextPathCode); i++) {		
PositionCode()		
}		
}		
else {		
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 */*		
If (AdditionalSchemaFlag) {		
SchemaIDOfSBC_Context_Selector	ceil(log2(`NumberOfSchemas + NumberOfAdditionalSchemas`))	uimsbf
Extended_SBC_Context_Selector	ceil(log2(number of global elements in `SchemaIDOfSBC_Context_Selector`))	bslbf
} else {		
SBC_Context_Selector	ceil(log2(number of global elements +1))	bslbf
}		
PathTypeCode()		
}		
else {		
SBC_Context	ceil(log2(number of child elements of complexType + 2))	bslbf

If (SBC_Context == “SBC_any”) {		
AnyElementDecoding ()		
} else {		
SubstitutionCode()		
}		
PathTypeCode()		
}		
TBC_Counter ++		
} while ((SBC_Context_Selector != “Path Termination Code”) && (SBC_Context != “Path Termination Code”))		
if (SBC_Context_Selector == “Path Termination Code”)) {		
If (AdditionalSchemaFlag) {		
SchemaIDOfSBC_Operand_Selector	ceil(log2(`NumberOfSchemas + NumberOfAdditionalSchemas`))	uimsbf
Extended_SBC_Operand_Selector	ceil(log2(number of global elements in `SchemaIDOfSBC_Operand_Selector`))	bslbf
} else {		
SBC_Operand_Selector	ceil(log2(number of global elements))	bslbf
}		
PathTypeCode()		
}		
else {		
SBC_Operand	ceil(log2(number of child elements + number of attributes + has_simpleContent + 1))	bslbf
if (SBC_Operand == “SBC_anyAttribute”) {		
AnyAttributeDecoding()		
}		
if (SBC_Operand == “SBC_any”) {		
AnyElementDecoding()		
}		
SubstitutionCode()		
PathTypeCode()		
}		
TBC_Counter ++		
for (i=0; i < TBC_Counter; i++) {		
PositionCode()		
}		

if ((ContextModeCode == '011') ||
(ContextModeCode == '100')) {
/ multiple fragment update payload mode*/*		

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

6.6.3 Semantics

Name	*Definition*
SchemaID	Identifies the schema (from the list of `schemaURI`s transmitted in the `DecoderInit` (optionally extended by a list of additional schemas) which is used as basis for the fragment update context coding. The `SchemaID` code word is built by sequentially addressing the list of `SchemaURI` contained in the `DecoderInit` (optionally followed by the additional schemas). The length of this field is determined by: "ceil(log2(`NumberOfSchemas`))" or "ceil(log2(`NumberOfSchemas + NumberOfAdditionalSchemas`))" depending on the presence of additional schemas. The value of this code word is the same as the variable "k" in the definition of the `SchemaURI[k]` syntax element as specified in 6.2.2 optionally extended to additional schemas. The `SchemaID` syntax element is also used for the decoding of the fragment update payload as described in subclause 7.4.4. 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 6.6.4.
ContextPath()	See 6.6.5.

Name	*Definition*
TBC_Counter	This internal variable represents the number of TBCs in the context path.
SchemaIDOfSBC_Context_Selector	Identifies the schema in which the `Extended_SBC_Context_Selector` selects a declared global element.
Extended_SBC_Context_Selector	Selects one global element of the schema referenced by `SchemaIDOfSBC_Context_Selector` using the ContextTBC table as specified in 6.6.5.2.3.
SBC_Context_Selector	Selects one global element of the schema referenced by `SchemaID` using the ContextTBC table as specified in 6.6.5.2.3.
PathTypeCode()	See 6.6.5.4.
SBC_Context	Selects one child node using the ContextTBC table as specified in 6.6.5.2.2.
AnyElementDecoding()	See 7.5.2.4.5.2 and 7.5.2.4.5.3.
SubstitutionCode()	See 6.6.5.3.
SchemaIDOfSBC_Operand_Selector	Identifies the schema in which the `Extended_SBC_Operand_Selector` selects a declared global element.
Extended_SBC_Operand_Selector	Selects one global element of the schema referenced by `SchemaIDOfSBC_Operand_Selector` using the ContextTBC table as specified in 6.6.5.2.3.
SBC_Operand_Selector	Selects one global element of the schema referenced by `SchemaID` using the table OperandTBC table as specified in 6.6.5.2.3.
SBC_Operand	Selects one child node using the OperandTBC table as specified in 6.6.5.2.2.
AnyAttributeDecoding()	See 7.5.3.2.
PositionCode()	See 6.6.5.5.
NumberOfFragmentPayloads	This internal variable represents the number of fragment update payload syntax elements present in this fragment update unit as specified in 6.6.5.6.
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 6.6.5.6.
IncrementalPositionCode	See 6.6.5.6.

6.6.4 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 8.

Table 8 — 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.
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 6.6.5.6.
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 6.6.5.6.
101	Navigate in “Absolute addressing mode” from the selector node to the node specified by the Context Path signaled by the ContextPathCode.
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 document stream shall use an absolute addressing mode (i.e. code ‘001’ or ‘011’). In addition, the first fragment update unit of the initial document shall use an absolute addressing mode.

6.6.5 Context Path

6.6.5.1 Overview

The Context Path specifies on which node in the binary fragment document 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.

6.6.5.2 Schema Branch Codes

6.6.5.2.1 Overview

A Schema Branch Code (SBC) is used to select a node as branch for the navigation in the binary format document tree. The SBCs in the ContextTBC table and in the OperandTBC table differ (as described in subclause 6.6.1 and shown in Table 7). The SBCs are assigned as specified in 6.6.5.2.2. For the special case of the selector node the SBCs are assigned as specified in 6.6.5.2.3.

6.6.5.2.2 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.

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

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. If there is an “any” element declared in the complex type then a SBC is also assigned to this element and this SBC is called “SBC_any”. If there is an “anyAttribute” declaration in the complex type then a SBC is also assigned to it and this SBC is called “SBC_anyAttribute”.

— 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. The lexicographical ordering for an “any” element and for an “anyAttribute” is done with respect to their signature as defined in subclause 7.5.2.2.4.

— 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 7.5.2.2.4.

  — 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.

6.6.5.2.3 SBC_Context_Selector and SBC_Operand_Selector

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

If the `AdditionalSchemaFlag` in the binary `DecoderInit` equals ‘0’ then

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

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

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

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

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

  — in case the `HierarchyBasedSubstitutionCodingFlag` is set to true, a depth first ordering is performed with respect to the hierarchy of element substitutions which forms one or several trees as shown in Figure 11. For elements which are siblings within the element substitution hierarchy or roots of a substitution hierarchy a lexicographical ordering is performed based on their expanded name as defined in 7.2.

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

If the `AdditionalSchemaFlag` in the binary `DecoderInit` equals ‘1’ then

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

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

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

— The SBCs are assigned sequentially to the global elements defined in the schema referred by the `SchemaIDOfSBC_Context_Selector` respectively `SchemaIDOfSBC_Operand_Selector`. No SBCs are assigned to elements imported from other namespaces into this schema. Before the assignment:

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

  — in case the `HierarchyBasedSubstitutionCodingFlag` is set to true, a depth first ordering is performed with respect to the hierarchy of element substitutions which forms one or several trees as shown in Figure 11. For elements which are siblings within the element substitution hierarchy or roots of a substitution hierarchy a lexicographical ordering is performed based on their expanded name as defined in 7.2.

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

6.6.5.3 Substitution Codes

6.6.5.3.1 Overview

In case a TBC represents a reference to an element 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`.

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

6.6.5.3.2 Syntax

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

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

6.6.5.3.3 Semantics

Name	*Definition*
substitution_possible	This is in internal flag which is derived from schema evaluation as specified in 6.6.5.3.1 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”.
external_element_substitution_possible	This internal flag indicates whether the element can be subject to a substitution occurring in an other schema than the one in which the element is defined. This flag is set by the `ExternallySubstitutableType` table defined in subclause 6.7.6.4 and 6.7.6.5.
all_element_externally_substitutable	This internal flag indicates whether every element defined in the schema of the expected element can be subject to a substitution occurring in an other schema. This flag is set by the `ExternallySubstitutableType` table defined in subclause 6.7.6.4 and 6.7.6.5.
SubstitutionFlag	Signals whether a substitution is present for the element (`SubstitutionFlag`=1).
SchemaSwitching	Indicates whether the element substitution occurs in another schema than the schema where the expected element is defined.
SchemaID	Identifies the schema in which the substitute element is defined.
GlobalSubstitutionSelect	This code identifies the substitute element in the schema of index ‘`SchemaID`’ in the `SchemaURI[k]` table. When the `HierarchyBasedSubstitutionCondingFlag` is set to false or when it is not defined in the `DecoderInit`, the code referring to the elements are assigned sequentially starting from zero after lexicographical ordering of all global elements using their expanded names as defined in subclause 7.2.

	When the HierarchyBasedSubstitutionCodingFlag is set to true, the SubstitutionSelect codes are assigned in a depth-first manner with respect to the hierarchy of element substitutions which forms on or several trees as shown in an example in Figure 11. For elements which are siblings within the element substitution hierarchy or elements which are roots of a substitution hierarchy the code words are assigned in a lexicographical order based on their expanded names. The order of elements that have a head of substitution in an other schema than the one identified by SchemaID are defined in the same relative order than if they were in the *initial schema*. The length of this field is determined by "ceil(log2(number_of_global elements in the schema identified by SchemaID))" in both cases. Note – If schema identified by SchemaID is an additional schema, the substitution select codes are computed on the set of all elements defined in the namespace identified by the SchemaID entry in the SchemaURI table of the DecoderInit.
SubstitutionSelect	This code is used as address within a substitution group where each element defined in the schema of the expected element is assigned a SubstitutionSelect code. When the HierarchyBasedSubstitutionCondingFlag is set to false or when it is not defined in the DecoderInit, the SubstitutionSelect codes referring to the elements are assigned sequentially starting from zero after lexicographical ordering of the element using their expanded names as defined in subclause 7.2. The length of this field is determined by "ceil(log2(number_of_possible_substitutes in the schema of the expected element))". In case the HierarchyBasedSubstitutionCondingFlag is true, the SubstitutionSelect codes are assigned in a depth-first manner with respect to the hierarchy of element substitutions which forms one or several trees as shown in an example in Figure 11. For elements which are siblings within the element substitution hierarchy or for elements which are roots of a substitution hierarchy the code words are assigned in a lexicographical order based on their expanded names. The element substitution code identifies the substitute element which is used in the encoded document. The length of the code word for the element substitution code is equal to "ceil(log2(number of possible_substitute in the schema of the expected element))". Note – If the schema identified by SchemaID is an additional schema, the substitution select codes are computed on the set of all elements defined in the namespace identified by the SchemaID entry in the SchemaURI table of the DecoderInit.

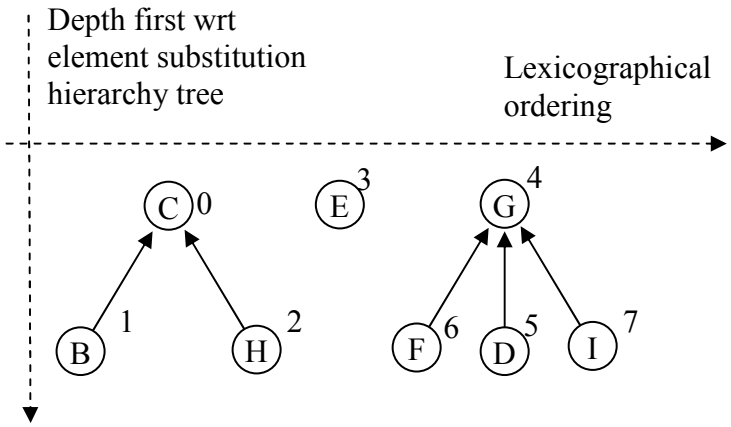

Figure 11 — Example for the Element Substitution Identification Code assignment for some elements in the hierarchy based coding mode

6.6.5.4 Type Code in the Context Path (Path_Type_Code)

6.6.5.4.1 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 document) 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.

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

6.6.5.4.2 Syntax

PathTypeCode () {	**Number of bits**	**Mnemonic**
if (type_cast_possible == 1 || external_type_cast_possible == 1 || all_type_externally_castable == 1) {		
TypeCodeFlag	1	bslbf
if ((TypeCodeFlag == 1) {		
if (external_type_cast_possible == 1 || all_type_externally_castable == 1) {		
SchemaSwitching	1	bslbf
if (SchemaSwitching) {		
SchemaID	ceil(log2(`NumberOfSchemas` + `NumberOfAdditionalSchemas`))	uimsbf

GlobalTypeIdentificationCode	ceil(log2 (number_of_global_types_in_schema_referred_by_SchemaID))	bslbf
} else {		
TypeIdentificationCode	ceil(log2(number of derived types))	bslbf
}		
} else {		
TypeIdentificationCode	ceil(log2(number of derived types))	bslbf
}		
}		
}		
}		

6.6.5.4.3 Semantics

Name	*Definition*
type_cast_possible	This internal flag which is derived from schema evaluation as specified in 6.6.5.4.1 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”.
external_type_cast_possible	Indicates whether the expected type can be subject to a type casting occurring in an other schema than the one in which the type is defined. This flag is set by the `ExternallyCastableType` table defined in subclause 6.7.5.4 and 6.7.5.5.
all_type_externally_castable	Indicates whether every type defined in the schema of the expected type can be subject to a type casting occurring in an other schema. This flag is set by the `ExternallyCastableType` table defined in subclause 6.7.5.4 and 6.7.5.5.
TypeCodeFlag	This flag indicates whether a type cast is present or not.
SchemaSwitching	Indicates whether the type cast occurs in an other schema than the schema of the expected type.
SchemaID	Identifies the schema in which the derived type element is addressed.
GlobalTypeIdentificationCode	Identifies a type defined in `SchemaIDOfDerivation` by a code word. The Type Identification Code is generated for a given type (simpleType or complexType) from the set of all types (itself being not included) including abstract types defined in the schema referenced by `SchemaID`.

	The Type Identification Codes are assigned in a depth-first manner with respect to the hierarchy of types which forms a tree as shown in an example in Figure 12. For types which are siblings within the type hierarchy the code words are assigned in a lexicographical order based on their expanded names. The Type Identification Code identifies the derived type which is used for the type cast. The length of the code word for the Type Identification Code is equal to “ceil(log2(number of types in the schema))”. The order of types that have a super type in an other schema than the one identified by `SchemaID` are defined in the same relative order than if they were in the initial schema. Note – If schema identified by `SchemaID` is an additional schema, the type codes are computed on the set of all types defined in the namespace identified by the `SchemaID` entry in the `SchemaURI` table of the `DecoderInit`.
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 12. 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))”.

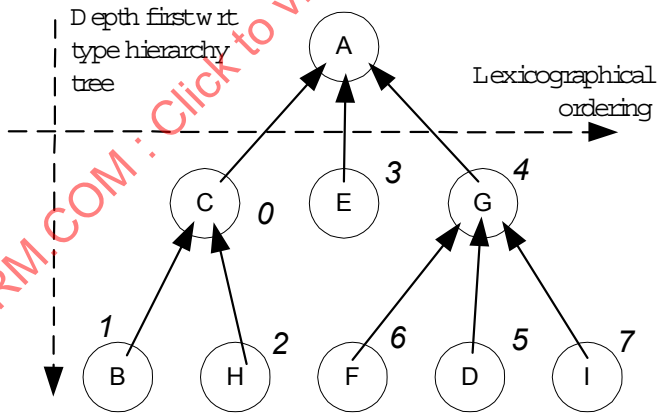

Figure 12 — Example for the Type Identification Code assignment for the types derived from A

6.6.5.5 Position Codes

6.6.5.5.1 Overview

Within a TBC a Position Code shall uniquely identify the position of a node among its sibling nodes in the binary document tree. Position Codes are distinguished in Multiple element Position Codes (MPC) and Single element Position Codes (SPC) for efficiency reasons. If the `MPCOnlyFlag` in the `DecoderInit` is set to true, MPC are always used. If the `MPCOnlyFlag` in the `DecoderInit` is set to false, then the presence of the

Position Code and the decision whether SPC or MPC are used is determined by the complexType definition: A position code is present (1) if multiple occurrences are possible for the element referenced by the SBC or (2) if the node is part of a model group declared in the corresponding complexType definition or (3) if the content model of the corresponding complexType definition is an ALL group. Also the OperandTBC of character content in a mixed content model contains a position code.

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.

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

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

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

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

In Figure 13 an example of a binary document tree of the element A including an assignment of position codes to child elements B is given. The Position Codes representing rational numbers specify the order in which the child elements B reside in the binary document tree.

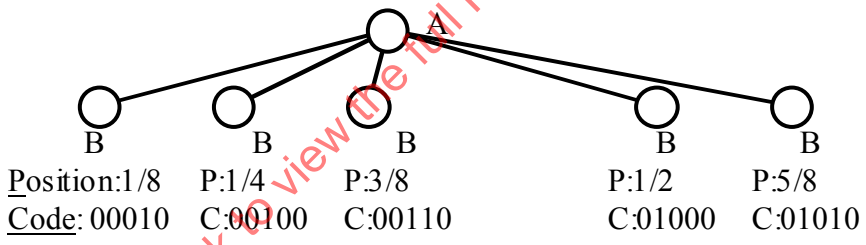

Figure 13 — Assignment of Position Codes to a set of child elements

When a new element B is inserted at any position then a new Position Code representing rational numbers is used so that the correct ordering in the binary document tree is unambiguously specified (see Figure 14).

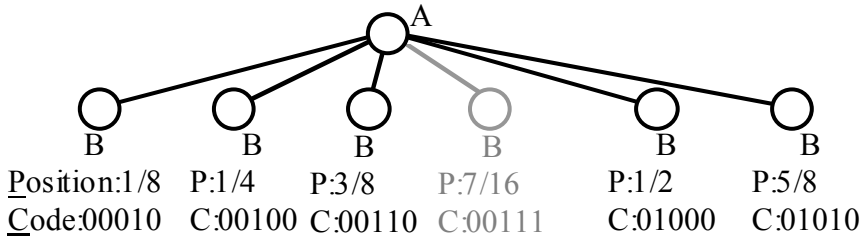

Figure 14 — Position Code of an inserted child element B (grey node)

6.6.5.5.2 Single Element Position Codes

A SPC is used when the following four conditions are met: (1) a Position Code is present according to 6.6.5.5.1 and (2) the MPCOnlyFlag is either not present or set to false and (3) the corresponding complexType does not contain model groups with maxOccurs > 1 and (4) if the content model of the corresponding complexType definition is not an ALL group. 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.

6.6.5.5.3 Multiple Element Position Codes

In case of complexTypes with complex content which contain model groups with maxOccurs > 1 or which contain an ALL group, 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.

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

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 i^{th} 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 mall= 'unbounded'

MPA_{all} = 'unbounded'

else

$MPA_{all} = m_{all} * number\_of\_children$

where

"$number\_of\_children_{i}$" is equal to the number of children of the all particle

"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

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

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.

If according to 6.6.5.5.1 the position is represented as rational number then the value is encoded as vlurmsbf5 and the value 0 shall be omitted.

If according to 6.6.5.5.1 the position is represented as integer number then the length in bits of the MPC is determined by the following method, which uses the 'max occurs' property of the effective content particles of the type definition.

6.6.5.5.4 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 document tree. In this case, the following implicit positions are assigned to each added node for which a position code is expected in the TBC addressing this node:

— If Position Codes represent integer numbers:

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

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

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

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

Step1: Determine Z.

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

Set i=0,P=0.

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

Step3: while (i<Z) {P(i)=P+=2/N; i++;}. End.In the implicit assignment of rational position codes, the step 2 performs an ascending-oriented assignment and the step 3 performs a balance-oriented assignment. The ascending-oriented assignment is efficient in case of appending subsequent fragments context paths, whereas the balance-oriented assignment is efficient in case of inserting/replacing subsequent fragments context paths. The condition of step 2 controls the ratio between such ascending and balance-oriented assignment. Figure 15 shows an example of the implicit assignment of positions represented by rational number.

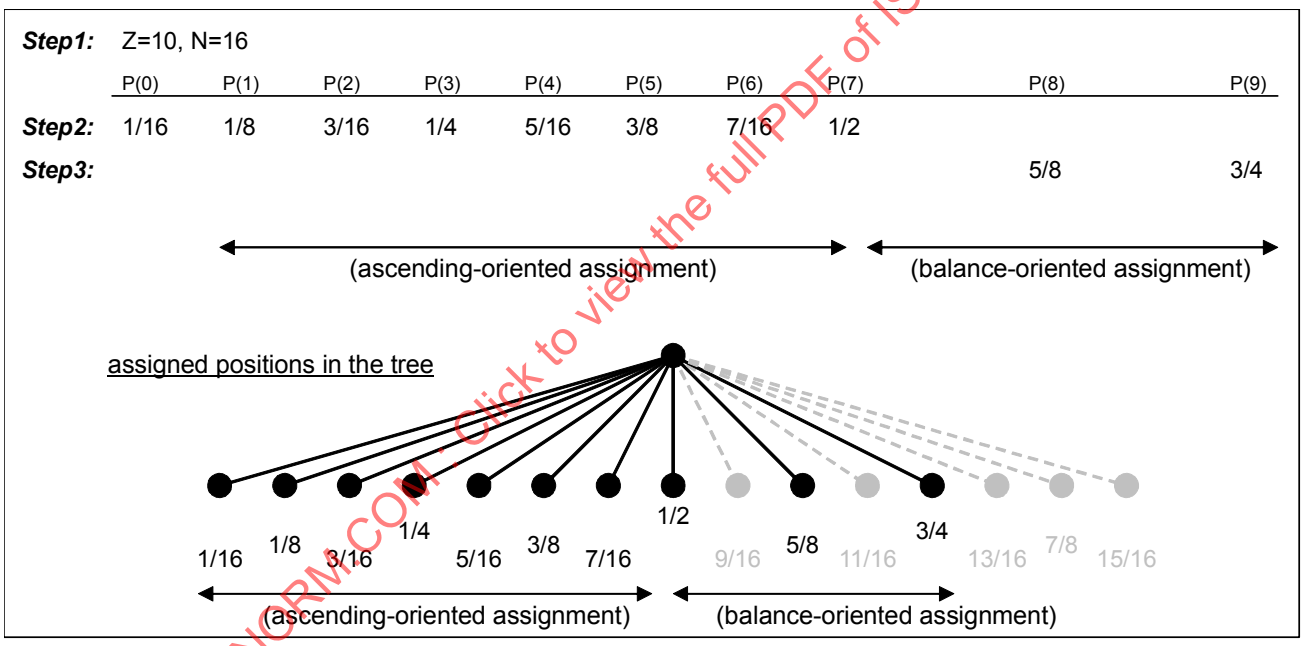

Figure 15 — an example on implicit assignment of positions represented by rational numbers

6.6.5.6 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. If position codes represent integer numbers, the position codes for the first fragment update payload are coded in the same way as in the case of a single payload, while the position codes for the other fragment update payloads within this fragment update unit are indicated in the context path by “Incremental Position Codes” as shown in Figure 16.

In the case of rational position codes, the structure of the context path is the same as the case of integer position codes. Only the Position Codes and the Incremental Position codes differ. For the Context Path rational position codes are used. Also for the incremental position codes rational position increments are specified. The order of rational position increments are predefined (see Figure 18).

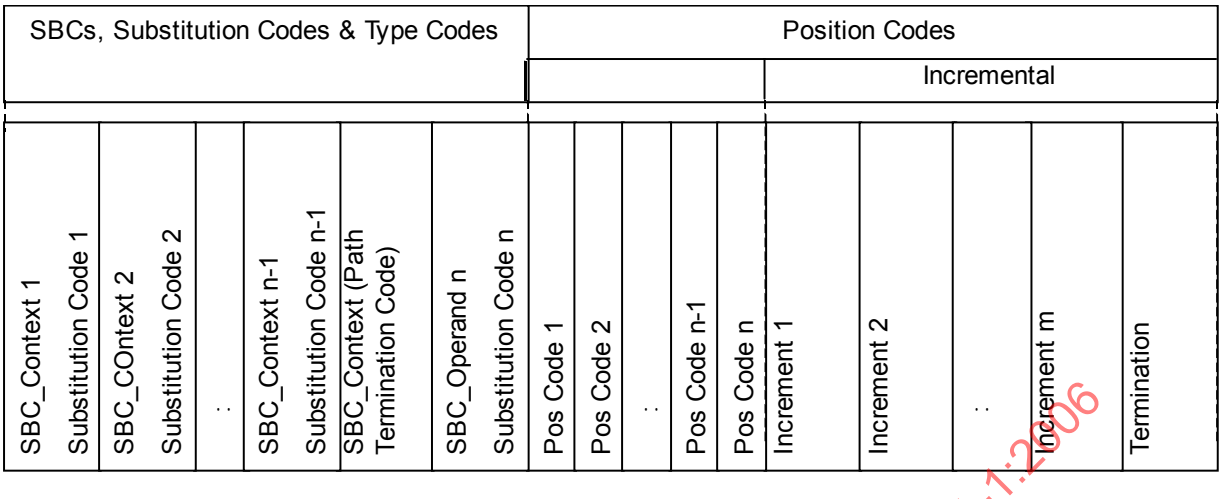

Figure 16 — Example of the structure of a Context Path (multiple fragment update payloads)

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 17.

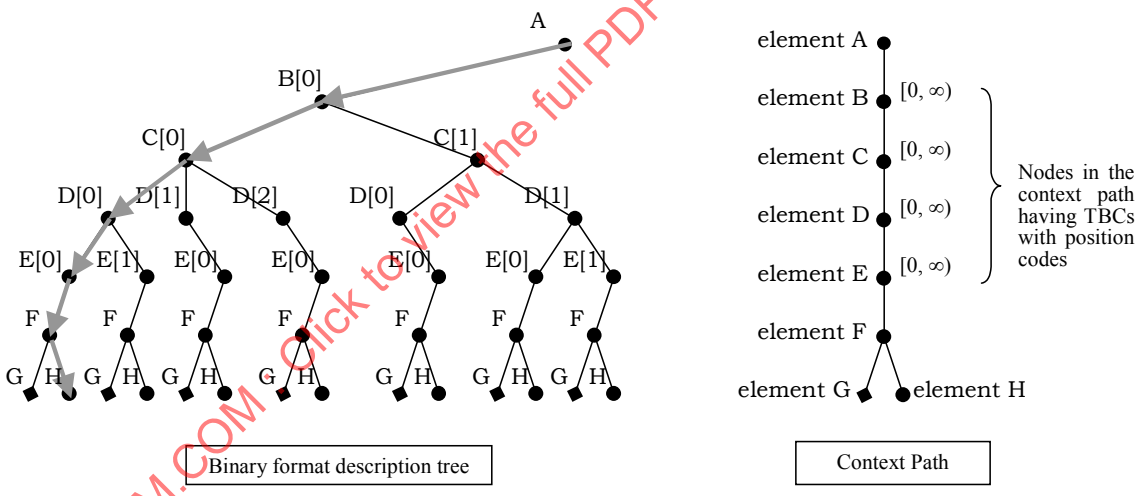

Figure 17 — Example for multiple-occurrence nodes in a context path

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. The position code for all multiple-occurrence nodes with a higher index is set to an initial value.

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

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

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

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

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

This order is defined first based on the resolution of the rational numbers which is the order of dividing the area (0,1) into halves repeatedly (Figure 18 (1)), i.e. the value of denominator. After that the ascending order is applied to the numbers in the same resolution (Figure 18 (2)).

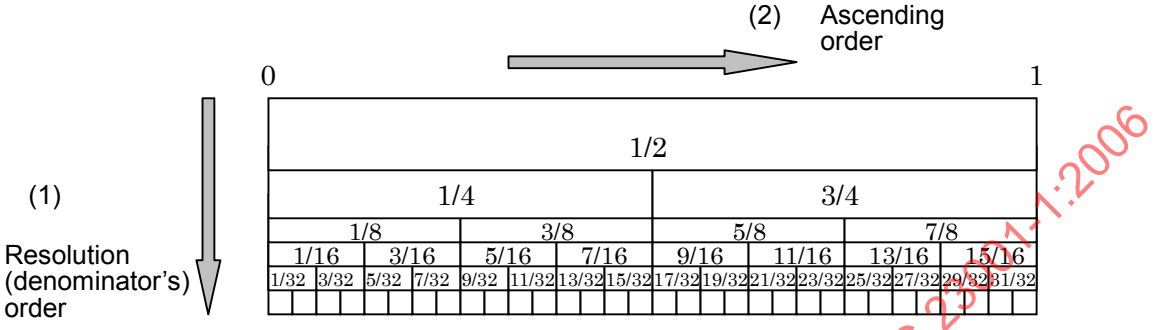

Figure 18 — Order of rational numbers

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

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

In 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 with integer position codes is given below:

Table 9 shows the case when the `NumberOfMultiOccurrenceLayer` is equal to 4 (i.e. shown by 3 bits). The multiple-occurrence nodes are indexed by 0 to 3.

Table 9 — 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 19, 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 19 (a). The code '011' denotes that the multiple-occurrence node with index 2 is updated as shown in Figure 19 (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 19 (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.

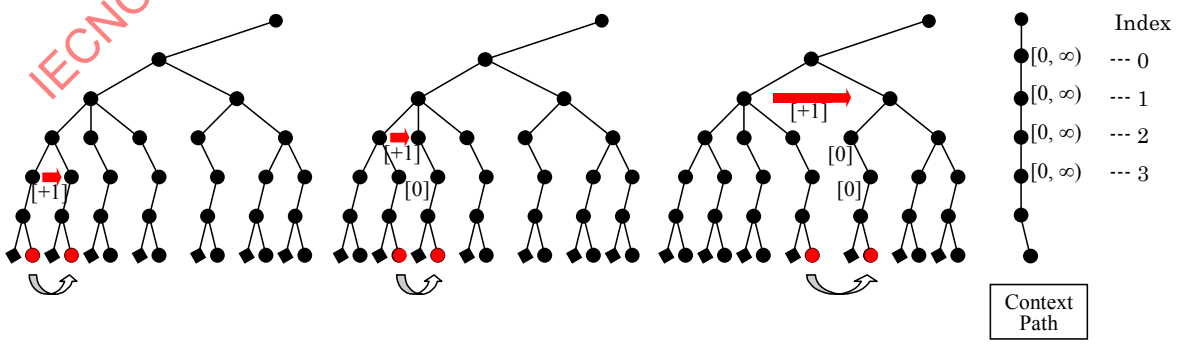

Figure 19 — Indicated Positions using Incremental Position Codes

Figure 20 shows an example of the encoding/decoding processes for the multiple payload mode with rational position codes. If Position Codes represent rational numbers, the position codes are sorted in the order of rational numbers before they are encoded (Figure 20 (1)). Once the positions are sorted, incremental position coding is applied to the rational position codes with the increment order of rational numbers and the initial value “1/2”. After the position codes are decoded, they are re-sorted in the ascending order (Figure 20 (2)).

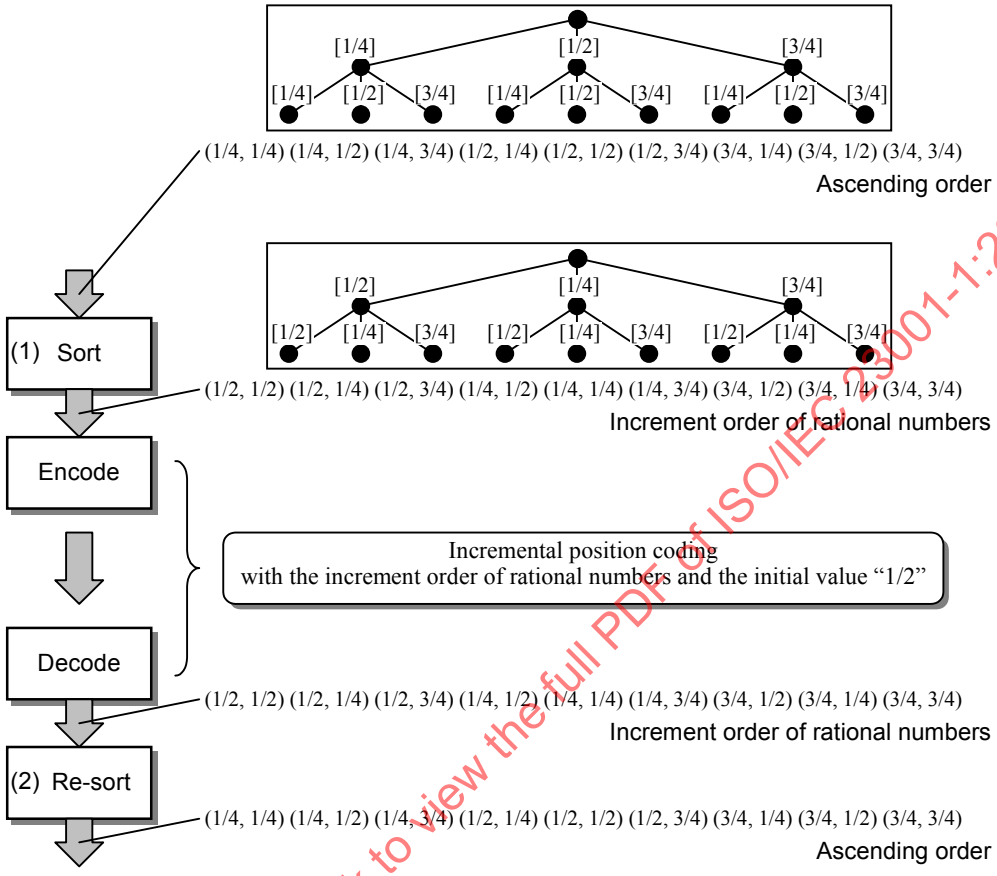

Figure 20 — Encoding/Decoding processes for the multiple payload mode with rational position codes

6.7 Binary Schema Update Unit

6.7.1 Overview

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

A schema update unit is composed of a namespace identifier, a set of code tables to represent global elements, global types and global attributes, followed by a binary encoded schema carrying the schema components definitions. This binary encoded schema is encoded using a specific profile of BiM specified in subclause 6.7.8. For that purpose a simple XML schema for schema encoding has been defined in this specification.

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

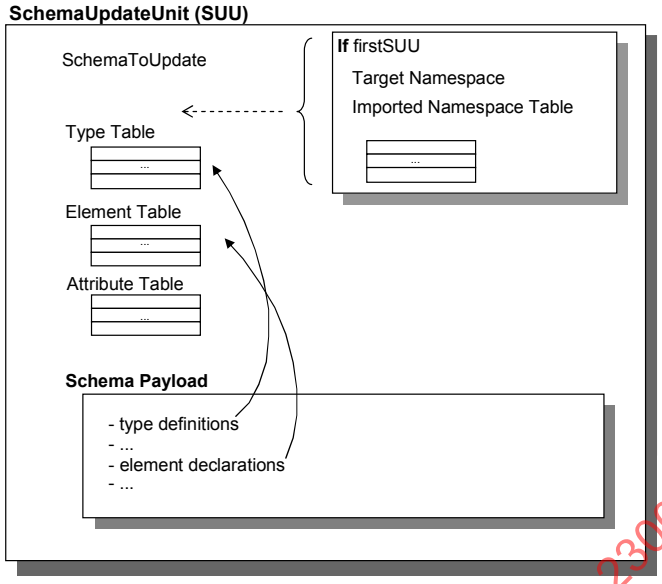

Figure 21 — A schema update unit

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

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

Once received by the decoder, a schema update unit immediately updates schema information managed by the decoder. All the optimised decoders associated to existing types are immediately applied to all types they derive from in accordance to the rules defined for the optimized decoders in Clause 8.

6.7.2 Syntax

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

6.7.3 Semantics

Name	*Definition*
SchemaToUpdate	Specifies the index in the table of schema which is updated by this fragment update unit.
FirstSUU	This flag is set to true if the SUU is a FirstSUU.
NamespaceURI_Length	Signals the length in bytes of the NamespaceURI_String.
NamespaceURI_String	UTF-8 representation of the namespace URI on which the SUU applies.
ImportedNamespaceTable	This table conveys the namespace referenced in the binary encoded schema as specified in subclause 6.7.4.
SchemaTypeTable	This table conveys the type code tables as specified in subclause 6.7.5.
SchemaElementTable	This table conveys the global elements and their possible substitutions as specified in subclause 6.7.6.
SchemaAttributeTable	This table conveys the global attributes as specified in subclause 6.7.7.
BinaryEncodedSchema	This conveys the binary encoded schema definitions as specified in subclause 6.7.8.

6.7.4 Imported NamespaceTable

6.7.4.1 Overview

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

6.7.4.2 Syntax

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

6.7.4.3 Semantics

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

6.7.5 Schema Type Table

6.7.5.1 Overview

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

6.7.5.2 Schema Type Table Syntax

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

6.7.5.3 Schema Type Table Semantics

The types are defined in the order of their type codes within the namespace as specified in subclause 6.6.5.4.

Name	*Definition*
NumberOfGlobalTypes	Defines the number of global types defined in the namespace.
PartialTransmission	Indicates that the transmission of the type table is partial.
NumberOfTransmittedTypes	Indicates the number of type definitions that are transmitted in the current SUU.
TransmittedType	Indicates the TypeCode of the type to be updated.
NumOfSubtypes[TransmittedType]	Indicates the number of subtype of the type 'TransmittedType' in the namespace.
NumOfSubtypes[i]	Indicates the number of subtype of the 'i^{th}' type of the namespace.

6.7.5.4 Externally Castable Type Table Syntax

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

6.7.5.5 Externally Castable Type Table Semantics

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

Name	*Definition*
IsThereExternallyCastableType	Signals that some types in the schema to update can be casted into types defined defined in other namespaces.
all_type_externally_castable	Signals that all types in the schema to update can be casted into types defined in other namespaces.
NumberOfExternallyCastableType	Indicates the number of types that can be casted into types defined in other namespaces.
ExternallyCastableType	Indicates the type code of a type which can be casted into types defined in other namespaces. In case this element is subject to a substitution (subclause 6.6.5.4), its `external_type_cast_possible` flag is set to '1'.

6.7.6 Schema Element Tables

6.7.6.1 Overview

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

6.7.6.2 Schema Element Table Syntax

SchemaElementTable (Schema){	**Number of bits**	**Mnemonic**
if (FirstSUU) {		
NumberOfGlobalElements	8+	vluimsbf8
ExternallyCastableElementTable(Schema)		
}		

PartialTransmission	1	blsbf
if (PartialTransmission){		
NumberOfTransmittedElements	5+	vluimbsf5
for (i=0; i < NumberOfTransmittedElements; i++) {		
TransmittedElement	5+	vluimsbf5
NumOfSubstituteElements[TransmittedElement]	5+	vluimsbf5
} else {		
for (i=0; i < NumberOfGlobalElements; i++) {		
NumOfSubstituteElements[i]	5+	vluimsbf5
}		
}		

6.7.6.3 Schema Element Table Semantics

The elements are defined in the order of their element codes within the namespace as specified in subclause 6.6.5.3.

Note – `HierarchyBasedSubstitutionCodingFlag` is always set to true when additional schemas are supported.

Name	*Definition*
NumberOfGlobalElements	Defines the number of global elements defined in the namespace.
PartialTransmission	Indicates that the transmission of the element table is partial.
NumberOfTransmittedElements	Indicates the number of element definitions that are transmitted in the current SUU.
TransmittedElement	Indicates the `SubstitutionSelect` code of the element to be updated.
NumOfSubstituteElements [TransmittedElement]	Indicates the number of substitute elements of the TransmittedElement in the updated namespace.
NumOfSubstituteElements [i]	Indicates the number of substitute elements of the ‘i^{th}’ element of the namespace.

6.7.6.4 Externally Substitutable Element Table Syntax

ExternallySubstitutableElementTable(Schema) {	**Number of bits**	**Mnemonic**
IsThereExternallySubstitutableElement	1	blsbf
If (IsThereExternallySubstitutableElement) {		
all_element_externally_substitutable	1	blsbf
If(!all_element_externally_substitutable) {		
NumberOfExternallySubstitutableElement	5+	vluimsbf5
for(i=0;i< NumberOfExternallySubstitutableElement;i++){		
ExternallySubstitutableElement	ceil(log2(Number OfGlobalElements in Schema))	blsbf
}		

6.7.6.5 Externally Substitutable Element Table Semantics

This table allows to specify which elements can be subject to an "external" element substitution i.e. a substitution in which the substitute element is defined in an other namespace.

Name	*Definition*
IsThereExternallySubstituableElement	Signals that some elements in the schema to update can be substituted into elements defined defined in other namespaces.
all_element_externally_substitutable	Signals that all elements in the schema to update can be substituted into elements defined in other namespaces.
NumberOfExternallySubstitutableElement	Indicates the number of elements that can be substituted into elements defined in other namespaces.
ExternallySubstitutableElement	Indicates the element code of an element which can be substituted into elements defined in other namespaces. In case this element is subject to a substitution (subclause 6.6.5.3), its `external_element_substitution_possible` flag is set to '1'.

6.7.7 Schema Attribute Table

6.7.7.1 Overview

This table conveys the global attributes of the updated schema.

6.7.7.2 Syntax

SchemaAttributeTable(Schema){	**Number of bits**	**Mnemonic**
if (FirstSUU) {		
NumberOfGlobalAttributes	8+	vluimsbf8
}		
PartialTransmission	1	blsbf
if (PartialTransmission){		
NumberOfTransmittedAttributes	5+	vluimbsf5
for (i=0; i<NumberOfTransmittedAttributes; i++) {		
TransmittedAttribute	ceil(log2(NumberOfGlobal Attributes in SchemaToUpdate))	uimsbf
}		
}		

6.7.7.3 Semantics

Name	*Definition*
NumberOfGlobalAttributes	Defines the number of global attributes defined in the namespace.
PartialTransmission	Indicates that the transmission of the element table is partial.
NumberOfTransmittedAttributes	Indicates the number of global attribute definitions that are transmitted in the current SUU.
TransmittedAttribute	Indicates the code of the received attribute.

6.7.8 Binary Encoded Schema

6.7.8.1 Overview

Each schema update unit carries a set of schema components definition in its `BinaryEncodedSchema`. This set is represented by an XML file conformant to a specific schema called the schema for encoding schema components. It is carried in a BiM encoded form using the schema for encoding schema components.

Note – The schema for encoding schema components is similar in its spirit to the XML Schema for schema. It has been however dedicated to the encoding of XML in BiM and not for validation as it is the case for the XML Schema for schema. It therefore concentrates on the features that are only used by a BiM decoder for decoding purposes only.

6.7.8.2 Decoding schema components using BiM

6.7.8.2.1 Binary Encoded Schema - DecoderInit

The following specific `DecoderInit` is used by the decoder for the decoding of binary encoded schema.

DecoderInit() {	**Value**	**Number of bits**
SystemsProfileLevelIndication	0x00	8
UnitSizeCode	000	3
NoAdvancedFeatures	0	1
ReservedBits	1111	4
AdvancedFeatureFlags_Length	0x01	8
InsertFlag	0	1
AdvancedOptimisedDecodersFlag	1	1
AdditionalSchemaFlag	0	1
AdditionalSchemaUpdatesOnlyFlag	0	1
FragmentReferenceFlag	0	1
MPCOnlyFlag	0	1
ReservedBitsZero	00	2
NumberOfSchemas	1	8+
SchemaURI_Length[0]	0x20 (i.e. 32)	8+
SchemaURI[0]	“urn:mpeg:mpeg7:schema Update:2002”	8* SchemaURI_Length[0]

LocationHint_Length[0]	0x00	8
NumOfAdvancedOptimisedDecoderTypes	0x01	8+
AdvancedOptimisedDecoderTypeURI_Length[0]	0x40 (i.e. 64)	8+
AdvancedOptimisedDecoderTypeURI[0]	"urn:mpeg:mpeg7:systems:SystemsAdvancedOptimisedDecodersCS:2003:1"	8* AdvancedOptimisedDecoderTypeURI_Length[0]
AdvancedOptimisedDecodersConfig () {		
NumOfAdvancedOptimisedDecoderInstances	0x00	8
NumOfMappings	0x00	8
}		
InitialDocument_Length	0x00	8
}		
}		

6.7.8.2.2 Binary Encoded Schema - Access Unit Constraints

A schema update unit is carried in one access unit constrained by the following rules:

— The access unit shall contain only one fragment update unit ;

— The fragment update unit shall update the top most node ;

— The fragment update unit shall use a "AddContent" command ;

— The fragment update unit shall have a context mode code set to '001' ;

— The `lengthCodingMode` code of the fragment update payload shall be set to '00' ;

— The `hasDeferredNodes` flag of the fragment update payload shall be set to '0' ;

— The `hasTypeCasting` flag of the fragment update payload shall be set to '1' ;

— The `hasNoFragmentReference` flag of the fragment update payload shall be set to '1'.

Moreover in the fragment update payload the following rules applies:

— The references (e.g. "base" or "type" attributes in XML Schema) to elements, types or attributes are encoded with "SchemaID" (in the local imported namespaces table) + "global code"

6.7.8.2.3 Binary Encoded Schema – Schema

The encoded schema shall respect the following constraints:

— Global types, elements and attributes are encoded in the same order than the one defined by their respective tables in the schema update units (`SchemaTypeTable`, `SchemaElementTable` and `SchemaAttributeTable`);

— Attributes in complex type definitions are sorted according to their expanded name;

— Content models shall be normalized as described in subclause 7.5.2.2.4.

6.7.8.3 Mapping schema components to the schema for encoding

6.7.8.3.1 Overview

The following subclauses specify the syntax elements and associated semantics of the schema for encoding schema updates.

The following schema wrapper shall be applied to the syntax defined in subclause 6.7.8.3.

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

   xmlns:m7s="urn:mpeg:mpeg7:systems:encodingschema:amd1:2004"
   targetNamespace="urn:mpeg:mpeg7:systems:encodingschema:amd1:2004"
     elementFormDefault="qualified"
     attributeFormDefault="unqualified">

   <!-- here clause 6.7.8.3 schema definitions -->

</schema>
```

6.7.8.3.2 Main schema element and type

6.7.8.3.2.1 Syntax

```
<xs:element name="schema" type="schemaType"/>

<xs:complexType name="schemaType">
 <xs:sequence>
   <xs:element name="typeDefinitions" type="typeDefinitionsType" minOccurs="0"/>
   <xs:element name="anonymousTypeDefinitions"
type="anonymousTypeDefinitionsType"
                                        minOccurs="0"/>
   <xs:element name="elementDeclarations" type="elementDeclarationsType"
                                        minOccurs="0"/>
   <xs:element name="attributeDeclarations" type="attributeDeclarationsType"
     minOccurs="0"/>
 </xs:sequence>
 <xs:attribute name="elementFormDefault"   type="qualificationType"
                          use="optional" default="unqualified"/>
 <xs:attribute name="attributeFormDefault" type="qualificationType"
                          use="optional" default="unqualified"/>
</xs:complexType>

<xs:simpleType name="qualificationType">
 <xs:restriction base="xs:string">
   <xs:enumeration value="qualified"/>
   <xs:enumeration value="unqualified"/>
 </xs:restriction>
</xs:simpleType>
```

6.7.8.3.2.2 Semantics

Name	*Definition*
schema	The root element of the schema update.
schemaType	The set of schema components updated by this schema update. Note - The namespace of this schema is encoded within the SchemaUpdateUnit and therefore not encoded here.
typeDefinitions	conveys the list of named types (or type globally defined).
anonymousTypeDefinitions	conveys the list of anonymous types (or type locally defined).
elementDeclarations	conveys the list of global elements.
attributeDeclarations	conveys the list of global attributes.
elementFormDefault	identical to the ‘elementFormDefault’ attribute defined in XML schema.
attributeFormDefault	identical to the ‘attributeFormDefault’ attribute defined in XML schema.
qualificationType	A type used to define the qualification (qualified/unqualified) of elements and attributes. This type is used by the 'form', 'elementFormDefault' and 'attributeFormDefault' attributes.

6.7.8.3.3 Element Declaration

6.7.8.3.3.1 Syntax

```
<xs:complexType name="elementDeclarationsType">
 <xs:sequence minOccurs="0" maxOccurs="unbounded">
  <xs:element name="globalElement">
   <xs:complexType>
    <xs:sequence>
     <xs:element name="elementTypeReference" type="typeReferenceType"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required"/>
   </xs:complexType>
  </xs:element>
 </xs:sequence>
</xs:complexType>

<xs:simpleType name="nameStringType">
 <xs:restriction base="xs:string"/>
</xs:simpleType>
```


6.7.8.3.3.2 Semantics

Name	*Definition*
elementDeclarationsType	The list of all global element declarations carried by this schema update unit. This list shall be ordered as specified in the specification (See subclause 6.7.6).
nameStringType	Defines the type of all the names used in the schema i.e. attribute, element and type names.

6.7.8.3.4 Type Declaration

6.7.8.3.4.1 Syntax

```
<xs:complexType name="typeDefinitionsType">
 <xs:choice maxOccurs="unbounded">
   <xs:element name="complexType" type="namedComplexTypeType"/>
   <xs:element name="simpleType" type="namedSimpleTypeType"/>
 </xs:choice>
</xs:complexType>

<xs:complexType name="anonymousTypeDefinitionsType">
 <xs:element name="complexType" type="anonymousComplexTypeType"
                                  maxOccurs="unbounded"/>
 <xs:element name="simpleType" type="anonymousSimpleTypeType"
    maxOccurs="unbounded"/>
</xs:complexType>
```

6.7.8.3.4.2 Semantics

Name	*Definition*
typeDefinitionsType	This type conveys the list of all the global types (complex and simple) carried by this schema update. The global types (or named type) are the types globally defined in an XML schema declaration.This list shall be ordered as defined in 6.7.5. The number of types contained in this list is encoded in the `SchemaTypeTable` (see 6.7.5.2).
complexType	conveys complex type definition.
simpleType	conveys the simple type definition.
anonymousTypeDefinitionsType	The list of all the anonymous types (or locally defined). No order is required on this list. In the case of a partial transmission, all the anonymous type required for resolving the type referencing mechanism shall be present in the schema update unit (see type referencing mechanism in 6.7.8.3.6).
complexType	conveys complex type definition.
simpleType	conveys the simple type definition.

6.7.8.3.5 Type Definition

6.7.8.3.5.1 Syntax

```
<xs:complexType name="typeType" abstract="true">
  <xs:sequence>
    <xs:element name="derivation" minOccurs="0">
      <xs:complexType>
        <xs:sequence>
            <xs:element name="baseTypeReference" type="typeReferenceType"
                                              minOccurs="1"/>
```

```
      </xs:sequence>
      <xs:attribute name="type" type="derivationType"
       use="required"/>
    </xs:complexType>
  </xs:element>
 </xs:sequence>
</xs:complexType>

<xs:simpleType name="derivationType">
 <xs:restriction base="xs:string">
  <xs:enumeration value="extension"/>
  <xs:enumeration value="restriction"/>
 </xs:restriction>
</xs:simpleType>

<xs:complexType name="complexTypeType" abstract="true">
 <xs:complexContent>
  <xs:extension base="typeType">
   <xs:sequence>
    <xs:element name="attributes" minOccurs="0">
     <xs:complexType>
      <xs:sequence>
       <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="attribute" type="localAttributeType"/>
        <xs:element name="attributeRef" type="attributeRefType"/>
       </xs:choice>
       <xs:element name="anyAttribute" type="anyAttributeType"
        minOccurs="0"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="content" type="contentModelType"/>
   </xs:sequence>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>

<xs:complexType name="namedComplexTypeType" >
 <xs:complexContent>
  <xs:extension base="complexTypeType">
   <xs:attribute name="name" type="nameStringType" use="required"/>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>

<xs:complexType name="anonymousComplexTypeType">
 <xs:complexContent>
  <xs:extension base="complexTypeType">
   <xs:attribute name="id" type="AnonymousTypeIDType" use="required"/>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>

<xs:complexType name="simpleTypeType" abstract="true">
 <xs:complexContent>
  <xs:extension base="typeType">
   <xs:sequence>
    <xs:choice>
     <xs:element name="list">
      <xs:complexType>
```

```
            <xs:sequence minOccurs="1">
              <xs:element name="itemTypeReference" type="typeReferenceType"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="union">
          <xs:complexType>
            <xs:sequence minOccurs="1" maxOccurs="unbounded">
              <xs:element name="memberTypeReference" type="typeReferenceType"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:choice>
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
        <xs:element name="facet" type="facetType"/>
      </xs:sequence>
    </xs:sequence>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>

<xs:complexType name="namedSimpleTypeType" >
 <xs:complexContent>
  <xs:extension base="simpleTypeType">
   <xs:attribute name="name" type="nameStringType" use="required"/>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>

<xs:complexType name="anonymousSimpleTypeType">
 <xs:complexContent>
  <xs:extension base="simpleTypeType">
   <xs:attribute name="id" type="AnonymousTypeIDType" use="required"/>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>
```


6.7.8.3.5.2 Semantics

Name	*Definition*
typeType	The abstract type of all types. This type defines derivation information for its subtypes `complexTypeType` or `simpleTypeType`.
derivation	defines the derivation methods by which a type is defined from its supertype (extension or restriction).
baseTypeReference	identifies the supertype of the types as defined in XML schema.
derivationType	identifies the derivation type (i.e. extension or restriction) as defined in XML schema.
complexTypeType	The abstract type of all complex type definitions. Its subtypes are `namedComplexTypeType` (used in case of type globally defined) and `anonymousComplexTypeType` (used in case of type locally defined)
attributes	The list of attributes declared within the complex type. The list shall be transmitted in the order defined in 7.5.3. In case of derivation by restriction,

	the entire list of attributes shall be listed. In case of derivation by extension, only new attributes shall be listed.
attribute	a locally defined attribute (cf. 6.7.8.3.7).
attributeRef	a reference to a global attribute defined in a schema (cf. 6.7.8.3.7).
anyAttribute	indicates the use of the anyAttribute (cf. 6.7.8.3.7).
content	defines the content model of a complexType. (cf. 6.7.8.3.8).
namedComplexTypeType	A globally defined complex type. Its name shall be present.
anonymousComplexTypeType	A locally defined complexType. An ID is associated to each anonymous type (cf. 6.7.8.3.6).
simpleTypeType	The abstract type of all simple type definitions. Its subtypes are `namedSimpleTypeType` (used in case of type globally defined) and `anonymousSimpleTypeType` (used in case of type locally defined).
list	If the Simple type is defined as a list, this element contains a reference to the item type which constitutes the element of the list.
union	If the simple type is defined as an union, this elements contains a reference to the different possible items of the union. The order of the elements has the same semantics than the defined in XML schema.
facet	conveys the facets of a simple type.
namedSimpleTypeType	a globally defined simple type. Its name shall be present.
anonymousSimpleTypeType	a locally defined simpleType (or anonymous type). An ID shall be associated to each anonymous type (cf. 6.7.8.3.6).

6.7.8.3.6 Type and Element Referencement

6.7.8.3.6.1 Syntax

```
<xs:complexType name="typeReferenceType">
 <xs:choice>
   <xs:element name="namedTypeReference">
     <xs:complexType>
       <xs:attribute name="NamespaceID" type="NamespaceIDType"
              use="optional"/>
       <xs:attribute name="TypeID" type="TypeIDRefType"
          use="required"/>
     </xs:complexType>
   </xs:element>
   <xs:element name="anonymousTypeReference">
     <xs:complexType>
       <xs:attribute name="idref" type="AnonymousTypeIDRefType"
          use="required"/>
     </xs:complexType>
   </xs:element>
 </xs:choice>
</xs:complexType>
```

```
<xs:complexType name="elementReferenceType">
 <xs:sequence>
  <xs:element name="namedElementReference">
   <xs:complexType>
    <xs:attribute name="NamespaceID" type="NamespaceIDType"
       use="optional"/>
    <xs:attribute name="ElementID" type="ElementIDRefType"
       use="required"/>
   </xs:complexType>
  </xs:element>
 </xs:sequence>
</xs:complexType>

<xs:simpleType name="NamespaceIDType">
 <xs:restriction base="xs:nonNegativeInteger"/>
</xs:simpleType>

<xs:simpleType name="TypeIDRefType">
 <xs:restriction base="xs:nonNegativeInteger"/>
</xs:simpleType>

<xs:simpleType name="ElementIDRefType">
 <xs:restriction base="xs:nonNegativeInteger"/>
</xs:simpleType>

<xs:simpleType name="AnonymousTypeIDType">
 <xs:restriction base="xs:nonNegativeInteger"/>
</xs:simpleType>

<xs:simpleType name="AnonymousTypeIDRefType">
 <xs:restriction base="AnonymousTypeIDType"/>
</xs:simpleType>
```


6.7.8.3.6.2 Semantics

Name	*Definition*
typeReferenceType	A reference to a type. This type is used when an element refers to a type or when a type refers to its super type. Two kind of types can be referenced: a named type (globally defined) or a anonymous type (locally defined).
namedTypeReference	The 'NamespaceID' attribute gives the index of the namespace, in the imported namespace table, in which the type is defined. The 'TypeID' attribute gives the index of the global type in the schema identified by the NamespaceID. If 'NamespaceID' attribute is not present, the namespace of the global type is the target namespace of the Schema Update.
anonymousTypeReference	The idref attribute gives the index in the table of anonymous type.

elementReferenceType	A reference to an element.The 'NamespaceID' attribute gives the index of the namespace, in the imported namespace table, in which the element is defined. The 'ElementID' attribute gives the index of the global element in the schema identified by the NamespaceID. If 'NamespaceID' attribute is not present, the namespace of the global element is the target namespace of the Schema Update.
NamespaceIDType	Defines the namespace id and refers to the table of imported namespace defined in the Schema Update Unit.
TypeIDRefType	Defines the type id and refers to the table of types carried in the Schema Update Unit.
ElementIDRefType	Defines the element id and refers to the table of types carried in the Schema Update Unit.
AnonymousTypeIDType	Defines the type id of an anonymous type. The scope of this id is limited to the current schema update unit. Therefore, in case of non complete transmission, Id values can be reused to identify different anonymous types.
AnonymousTypeIDRefType	Defines a reference to an anonymous type.

6.7.8.3.7 Attribute definition

6.7.8.3.7.1 Syntax

```
<xs:complexType name="attributeDeclarationsType">
 <xs:sequence>
   <xs:element name="attribute" type="globalAttributeType"
      maxOccurs="unbounded"/>
 </xs:sequence>
</xs:complexType>

<xs:complexType name="attributeType" abstract="true">
 <xs:sequence>
   <xs:element name="typeReference" type="typeReferenceType"/>
 </xs:sequence>
 <xs:attribute name="name" type="nameStringType"  use="required"/>
 <xs:attribute name="defaultValue" type="xs:string" use="optional"/>
</xs:complexType>

<xs:complexType name="attributeRefType">
 <xs:attribute name="idref" type="xs:IDREF" use="required"/>
</xs:complexType>

<xs:complexType name="localAttributeType">
 <xs:complexContent>
   <xs:extension base="attributeType">
     <xs:attribute name="use" type="useType" use="required"/>
     <xs:attribute name="form" type="qualificationType" use="required"/>
   </xs:extension>
 </xs:complexContent>
</xs:complexType>
```

```
<xs:complexType name="globalAttributeType">
 <xs:complexContent>
   <xs:extension base="attributeType"/>
 </xs:complexContent>
</xs:complexType>

<xs:complexType name="anyAttributeType">
</xs:complexType>

<xs:simpleType name="useType">
 <xs:restriction base="xs:string">
   <xs:enumeration value="optional"/>
   <xs:enumeration value="required"/>
 </xs:restriction>
</xs:simpleType>
```

6.7.8.3.7.2 Semantics

Name	*Definition*
attributeDeclarationsType	The list of all the global attribute declarations carried by this schema update. This list is ordered as specified in subclause 6.7.7.
attributeType	An abstract type conveying the definition of an attribute. The type of the defined attribute is identified by a type reference. The name of the attribute shall be present. The defaultValue, if it exists, shall be encoded.
attributeRefType	A reference to a global attribute. It is used when a type references a global attribute.
localAttributeType	Defines the type of an attribute defined within a complex type. The `use` attribute shall be present. Its semantics is identical to the one defined in XML schema. The `form` attribute shall be instantiated, its semantics is identical to the one defined in XML schema.
globalAttributeType	Defines the type of a global attribute.
anyAttributeType	Indicates that any attribute of any schema can be present in the complexType.
useType	The type of the `use` attribute. It has two possible values: 'optional' or 'required'.

6.7.8.3.8 Content Model

6.7.8.3.8.1 Syntax

```
<xs:complexType name="contentModelType" abstract="true"/>

<xs:complexType name="emptyContentModelType">
 <xs:complexContent>
   <xs:extension base="contentModelType"/>
 </xs:complexContent>
</xs:complexType>
```

```
<xs:complexType name="simpleContentModelType">
 <xs:complexContent>
  <xs:extension base="contentModelType">
   <xs:sequence>
    <xs:element name="simpleTypeReference"
       type="typeReferenceType"/>
   </xs:sequence>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>

<xs:complexType name="complexContentModelType">
 <xs:complexContent>
  <xs:extension base="contentModelType">
   <xs:sequence>
    <xs:element name="particle" type="particleType"/>
   </xs:sequence>
   <xs:attribute name="mixed" type="xs:boolean" use="required"/>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>
```

6.7.8.3.8.2 Semantics

Name	*Definition*
contentModelType	This abstract type defines the content model of a complex type. It has three subtypes addressing the three content models defined by XML Schema: `emptyContentModelType`, `simpleContentModelType` and `complexContentModelType`.
emptyContentModelType	An empty content model.
simpleContentModelType	A simple content model. It includes a reference to the simple type which defines the content model of a complex type.
complexContentModelType	A complex content model. This model contains a particle (see XML Schema) and its 'mixed' attribute shall be instantiated.

6.7.8.3.9 Facet Definition

6.7.8.3.9.1 Syntax

```
<xs:complexType name="facetType" abstract="true">
 <xs:attribute name="name" type="possibleFacet" use="required"/>
 <xs:attribute name="value" type="xs:string"/>
</xs:complexType>

<xs:simpleType name="possibleFacet">
 <xs:restriction base="xs:string">
  <xs:enumeration value="maxExclusive"/>
  <xs:enumeration value="minExclusive"/>
```

```
  <xs:enumeration value="minInclusive"/>
  <xs:enumeration value="maxInclusive"/>
  <xs:enumeration value="enumeration"/>
  <xs:enumeration value="length"/>
  <xs:enumeration value="minLength"/>
  <xs:enumeration value="maxLength"/>
 </xs:restriction>
</xs:simpleType>
```

6.7.8.3.9.2 Semantics

Name	*Definition*
facetType	Defines the possible facets associated to a simple type. A facet is composed of a name and a value. The facet mechanism is equivalent to the one defined in XML schema.
possibleFacet	The set of possible facets are limited to the ones used by BiM (see 7.5.4).

6.7.8.3.10 Particle Defintion

6.7.8.3.10.1 Syntax

```
<xs:complexType name="particleType" abstract="true">
 <xs:attribute name="minOccurs" type="xs:unsignedInt" default="1"/>
 <xs:attribute name="maxOccurs" type="occurrenceType" default="1"/>
</xs:complexType>

<xs:complexType name="anyParticleType">
 <xs:complexContent>
  <xs:extension base="particleType"/>
 </xs:complexContent>
</xs:complexType>

<xs:complexType name="element">
 <xs:complexContent>
  <xs:extension base="particleType">
   <xs:sequence>
    <xs:element name="type" type="typeReferenceType"/>
   </xs:sequence>
   <xs:attribute name="name" type="nameStringType"/>
   <xs:attribute name="form" type="qualificationType"/>
   <xs:attribute name="nillable" type="xs:boolean"  use="required"/>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>

<xs:complexType name="elementRef">
 <xs:complexContent>
  <xs:extension base="particleType">
   <xs:sequence>
    <xs:element name="ref" type="elementReferenceType"/>
   </xs:sequence>
```

```
  </xs:extension>
 </xs:complexContent>
</xs:complexType>

<xs:complexType name="modelGroupType" abstract="true">
 <xs:complexContent>
  <xs:extension base="particleType">
   <xs:sequence maxOccurs="unbounded">
    <xs:element name="particle" type="particleType"/>
   </xs:sequence>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>

<xs:complexType name="sequence">
 <xs:complexContent>
  <xs:extension base="modelGroupType"/>
 </xs:complexContent>
</xs:complexType>

<xs:complexType name="all">
 <xs:complexContent>
  <xs:extension base="modelGroupType"/>
 </xs:complexContent>
</xs:complexType>

<xs:complexType name="choice">
 <xs:complexContent>
  <xs:extension base="modelGroupType"/>
 </xs:complexContent>
</xs:complexType>

<xs:simpleType name="occurrenceType">
 <xs:union memberTypes="unboundedType xs:unsignedInt"/>
</xs:simpleType>

<xs:simpleType name="unboundedType">
 <xs:restriction base="xs:string">
  <xs:enumeration value="unbounded"/>
 </xs:restriction>
</xs:simpleType>
```


6.7.8.3.10.2 Semantics

Name	*Definition*
particleType	An abstract type defining the type of all particles (cf. XML Schema). The range of occurrences is defined by the `minOccurs` and `maxOccurs` attributes.
anyParticleType	The any wildcard. This particle indicates that any global element of any namespace can be present in the document.
element	A local declaration of an element. This declaration is composed of a reference to a type, a name, a qualification form and a nillable property.
elementRef	A reference to an element globally defined.
modelGroupType	This particle is the super type of all groups: sequence, choice and all

sequence	A ‘sequence’ particle.
all	An ‘all’ particle.
choice	A ‘choice’ particle.
occurrenceType	The type for the `maxOccurs` attribute. Its value can be either an ‘xs:unsignedInt’ or the string value ‘unbounded’.

7 Binary Fragment Update Payload

7.1 Overview

The binary fragment update payload syntax (`FUPayload`) is specified in subclause 7.3. It is composed of flags which define some decoding modes and a payload content which is either an `element`, a simple value (`simpleType`) or a reference to a payload. The syntax of a binary `element` is specified in subclause 7.4. The element content (attributes, complex content or simple content) is decoded by the decoding processes specified in subclause 7.5. 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 document tree are decoded by dedicated decoders associated to simple types.

7.2 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 6.6.5.4.

SchemaComponent expanded name

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

A 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.

7.3 Fragment Update Payload syntax and semantics

7.3.1 FragmentUpdatePayload

7.3.1.1 Syntax

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

7.3.1.2 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 6.6).
SimpleType()	See 7.4.7.
DecodingModes()	See 7.3.2.
Element()	See 7.4.1. The “hot” value and its semantics are defined in 7.4.1.

7.3.2 Decoding Modes

7.3.2.1 Syntax

DecodingModes () {	**Number of bits**	**Mnemonic**
lengthCodingMode	2	bslbf
hasDeferredNodes	1	bslbf
hasTypeCasting	1	bslbf
hasNoFragmentReference	1	bslbf
ReservedBits	3	bslbf
}		

7.3.2.2 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 10.
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.
hasNoFragmentReference	A flag which specifies if this fragment update payload contains a fragment reference or the encoded fragment. This 1-bit flag can have the following values: — 0 : `hasNoFragmentReference` is equal to false, — 1 : `hasNoFragmentReference` is equal to true.
ReservedBits	Reserved for future extensions.

Table 10 — lengthCodingMode definition

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

7.4 Element syntax and semantics

7.4.1 Element

7.4.1.1 Syntax

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

7.4.1.2 Semantics

Name	*Definition*
FragmentReference	See 7.4.8.
NumberOfSchemas	The number of schema conveyed in the `DecoderInit` as specified in 6.2.
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 7.4.3) — “*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 11: — "*mono_not_frozen*" - The decoding of the element is performed using the mono-version decoding mode as specified in subclause 7.4.2. 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 7.4.3. The schemas used for the decoding of the element content are specified by the `SchemaID` field of each `ElementContentChunk` as specified in 7.4.4. — "*mono_frozen*" - The decoding of the element is performed using the mono-version decoding mode as specified in subclause 7.4.2. 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 7.4.3. Each one of its children elements is decoded using the mono-version decoding mode as specified in subclause 7.4.2. The schemas used for the decoding process are specified by the `SchemaID` of each `ElementContentChunk` as specified in 7.4.4.
ElementContentDecodingMode	An enumerated variable which determines in which mode the element decoding is 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 7.4.2. — *"multi"* - The decoding of the element is performed using the multiple-version decoding mode as specified in subclause 7.4.3. Its value is deduced from the `SchemaModeStatus` and the `SchemaModeUpdate` according to rules defined in Table 12.
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 12.
Mono-VersionElementContent()	See 7.4.2.
Multiple-VersionElementContent()	See 7.4.3.

Table 11 — Schema Mode Update

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

Table 12 — 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

7.4.2 Mono-version element content

7.4.2.1 Syntax

Mono-VersionElementContent (Enumeration ChildrenSchemaMode, SchemaComponentTheType) {	Number of bits	Mnemonic
If (!in_AnyElementDecoding) {		
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)		
}		
}		
}		

7.4.2.2 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 6.6.5.3.
PayloadTypeCode ()	Indicates the type information as specified in subclause 7.4.5.
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 `theType` is associated to an optimised type decoder as conveyed in the `DecoderInit` (refer to subclause 6.2).
optimisedDecoder ()	Triggers the optimised type decoder associated to the type `theType` as conveyed in the `DecoderInit` (refer to subclause 6.2).
Attributes()	Decodes element attributes as specified in subclause 7.5.3.
Content()	Decodes element content as specified in subclause 7.4.6.

7.4.3 Multiple-version element content

7.4.3.1 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 6.2). 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 document. 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.

7.4.3.2 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 22. 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 22). 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.

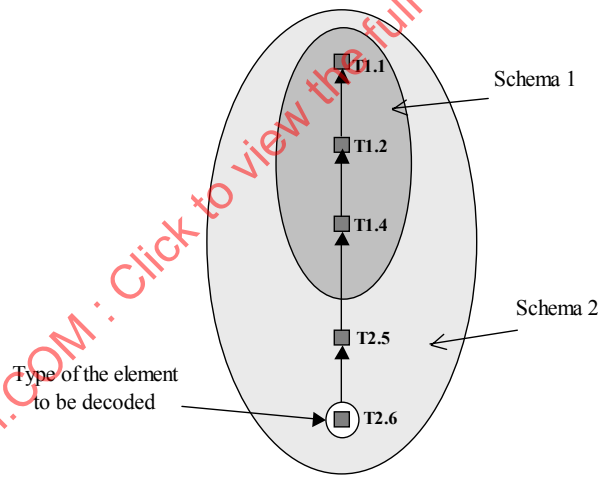

Figure 22 — Example of a type hierarchy defined across versions

Figure 23 — Example of a forward compatible encoding

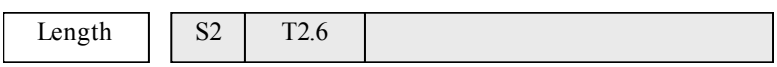

Figure 24 — Example of a non forward compatible encoding

7.4.3.3 Syntax

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

7.4.3.4 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 6.2 (optionally extended with the list of additional schemas).
SubstituteElementCode	The code of the substitute element. The code is computed following the rules defined in 6.6.5.3 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 7.4.4.
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.

7.4.4 ElementContentChunk

7.4.4.1 Syntax

SchemaComponent ElementContentChunk (Enumeration ChildrenSchemaMode, SchemaComponent currentType) {	**Number of bits**	**Mnemonic**
SchemaID	ceil(log2(`NumberOfSchemas + NumberOfAdditionalSchemas`))	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		
}		

7.4.4.2 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 6.2 (optionally extended with the list of additional schemas).
PayloadTypeCode ()	Decodes type information as specified in subclause 7.4.5. 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 document 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 7.5.3 using the element type `nextType`.
Content ()	Decodes element content as specified in subclause 7.4.6 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 7.5.3.

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 7.4.6.

7.4.5 PayloadTypeCode

7.4.5.1 Syntax

SchemaComponent PayloadTypeCode(SchemaComponent defaultType, boolean multi) {	**Number of bits**	**Mnemonic**
if (multi) {		
If(firstElementContentChunk()) {		
PayloadTypeCastFlag	1	bslbf
if (PayloadTypeCastFlag == 1) {		
PayloadTypeIdentificationCode	ceil(log2(number of possible subtypes of defaultType + sizeIncrease))	bslbf
effectiveType = getDerivedType(defaultType, PayloadTypeIdentificationCode)		
}		
} else		
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		
}		

7.4.5.2 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 6.6.5.4. 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.

7.4.6 Content

7.4.6.1 Syntax

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

7.4.6.2 Semantics

Name	*Definition*
SimpleType	See 7.4.7.
ComplexContent	Refers to the decoding process specified in subclause 7.5.2.

7.4.7 SimpleType

7.4.7.1 Syntax

SimpleType(SchemaComponent theType) {	Number of bits	Mnemonic
If (! AdvancedOptimisedDecodersFlag) {		
if (useOptimisedDecoder(theType)) {		
optimisedDecoder(theType)		
} else {		
defaultDecoder(theType)		
}		
} else {		
if (numOfMappedOptimisedDecoder(theType) !=0) {		
optimisedDecoderID	ceil(log2(number of decoders associated to this type))	blsbf
advancedOptimisedDecoder(theType, optimisedDecoderID)		
} else {		
defaultDecoder(theType)		
}		
}		

7.4.7.2 Semantics

A simple type can be associated to a single default simple type decoder, a single simple optimised decoder and one or several optimised decoders. This syntax table describes the process to select the proper simple type decoder for each simple type to be decoded.

Name	*Definition*
useOptimisedDecoder()	Returns "true" if the type `effectiveType` is associated to an optimised type decoder as conveyed in the `DecoderInit` (refer to subclause 6.2).
numOfMappedOptimisedDecoder ()	Returns the number of advanced optimised decoders associated to the type `theType` as described in Clause 8.
optimisedDecoder ()	Triggers the simple optimised type decoder associated to the type `effectiveType` as conveyed in the `DecoderInit` (refer to subclause 6.2).
defaultDecoder ()	Triggers the default decoder associated to the type "theType" as specified in subclause 7.5.4.1.
advancedOptimisedDecoder(aType, anInteger)	Triggers the advanced optimised decoder identified by the `optimisedDecoderID` field which is associated to the type `theType` in the optimised decoder mapping.

7.4.8 FragmentReference

7.4.8.1 Syntax

FragmentReference () {	**Number of bits**	**Mnemonic**
isDeferred	1	bslbf
hasSpecificFragmentReferenceFormat	1	bslbf
ReservedBits	6	bslbf
If (hasSpecificFragmentReferenceFormat == '1') {		
FragmentReferenceFormat	8	bslbf
} else {		
FragmentReferenceFormat = SupportedFragmentReferenceFormat[0]		
}		
FragmentRefLength	8+	vluimsbf8
/** FragmentRef **/		
if(FragmentReferenceFormat == "0x01") {		
URIFragmentReference()		
} else {		
ReservedBits	FragmentRef Length	
}		
/** Fragment ref end **/		
}		

7.4.8.2 Semantics

Name	*Definition*
isDeferred	A flag which signals if the fragment is deferred, and therefore can be acquired later on by the terminal. If the `isDeferred` has a value of "0", the decoder shall resolve the reference and obtain the fragment payload according to the process described in subclause 5.7.1. If the `isDeferred` has a value of "1", then the decoded value of the fragment shall include a deferred reference containing the specified fragment reference as specified in subclause 5.7.2. A node that is represented by a deferred fragment reference shall be treated by the decoder as a deferred node.
hasSpecificFragmentReferenceFormat	A flag which signals if the fragment reference format is different from the default one defined in the `DecoderInit` (see subclause 6.2): — If `hasSpecificFragmentReferenceFormat` has a value of '1', the decoder shall use the `FragmentReferenceFormat` field as indication of the fragment reference format. — If `hasSpecificFragmentReferenceFormat` has a value of '0' then the decoder shall take the fragment reference format to be that defined within the `DecoderInit` i.e. the default setting.
FragmentReferenceFormat	An 8-bits value which uniquely identifies the format of the contained fragment reference as defined in Table 3. This field shall only be present when `hasSpecificFragmentReferenceFormat` has a value of '1'.
FragmentRefLength	The number of bits that follows this field, which denotes the size of the `FragmentRef` field.

7.4.9 URIFragmentReference

7.4.9.1 Syntax

URIFragmentReference () {	**Number of bits**	**Mnemonic**
href	FragmentRefLength	bslbf
}		

7.4.9.2 Semantics

Name	*Definition*
href	Defines the URI of the URI fragment reference in UTF-8 format. The field is of variable length and its actual length shall be inferred from the `FragmentRefLength` field defined within the `FragmentReference` (see subclause 7.4.8).

7.5 Element Content decoding process

7.5.1 Overview

The element content decoder relies on schema analysis. The schema analysis generates a “finite state automaton decoder” that models the decoding of a complex content. The use and construction of finite state automaton decoders is defined in subclause 7.5.2.

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.

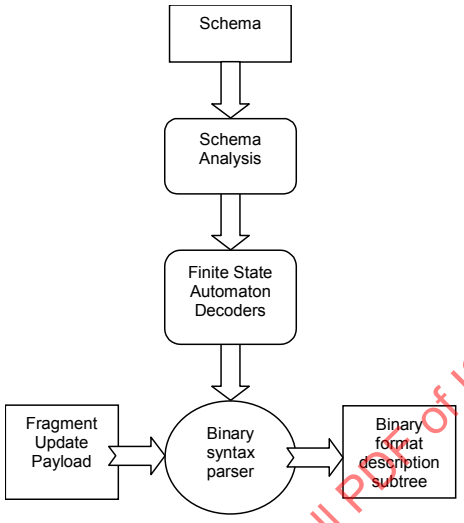

Figure 25 — The complex content decoding process

7.5.2 Complex content decoding process

7.5.2.1 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 7.4.6 or the `ContentDelta` syntax element defined in subclause 7.4.4, the token is placed in the “start state”. When the token reaches the “final state” the decoding of the complex content is finished.

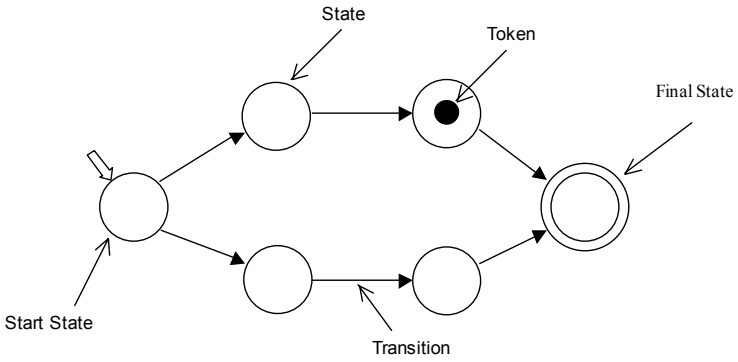

Figure 26 — Example of an automaton

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 document.

— *Wildcard transitions:* Wildcard transitions, when crossed, specify to the decoder that an undefined element is present in the document.

— *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 document 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.

  — *Mixed transitions*: Mixed transitions are a special kind of code transitions. Their binary code value is always equal to 1. Mixed transitions, when crossed, specify to the decoder that a string is present between the previous decoded element and the next one.

— *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 7.5.2.2. The decoding process using finite state automaton decoders is specified in subclause 7.5.2.3. The behaviors of the above mentioned states and transitions are specified in subclause 7.5.2.4.

7.5.2.2 Finite state automaton decoder construction

7.5.2.2.1 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.

7.5.2.2.2 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 equal 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 7.2)

7.5.2.2.3 Phase 2 - Syntax tree generation

7.5.2.2.3.1 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, wildcard nodes, group nodes and occurrence nodes. Element declaration nodes associate an element name to its type. Wildcard nodes represent element wildcards. Both element declaration and wildcard nodes are leaves of the syntax tree and are derived respectively from their element declaration and wildcard particles. Group nodes define a composition group (sequence, choice or all) and are derived from group particles. A group node contains only occurrence nodes. Occurrence nodes are derived from the 'min occurs' and 'max occurs' property of the particle and contain group nodes, element declaration or wildcard nodes.

The syntax tree is reduced to improve the compression efficiency of the binary format by the transformations defined in subclauses 7.5.2.2.3.2, 7.5.2.2.3.3 and 7.5.2.2.3.4. 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}".

7.5.2.2.3.2 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 27.

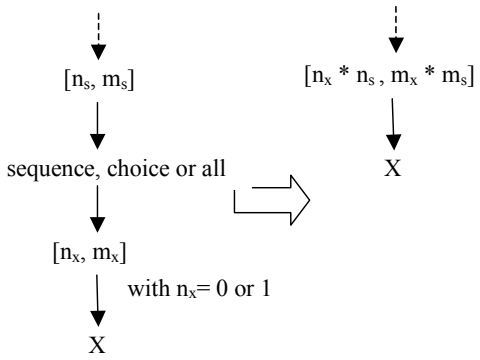

Figure 27 — Group simplification rule

7.5.2.2.3.3 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.

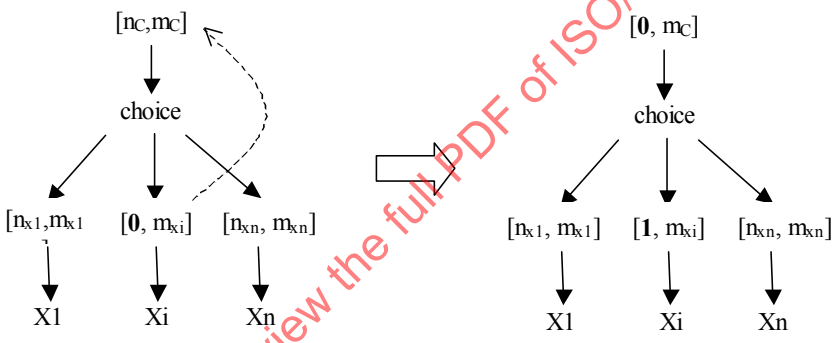

Figure 28 — Empty choice simplification rule

7.5.2.2.3.4 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.

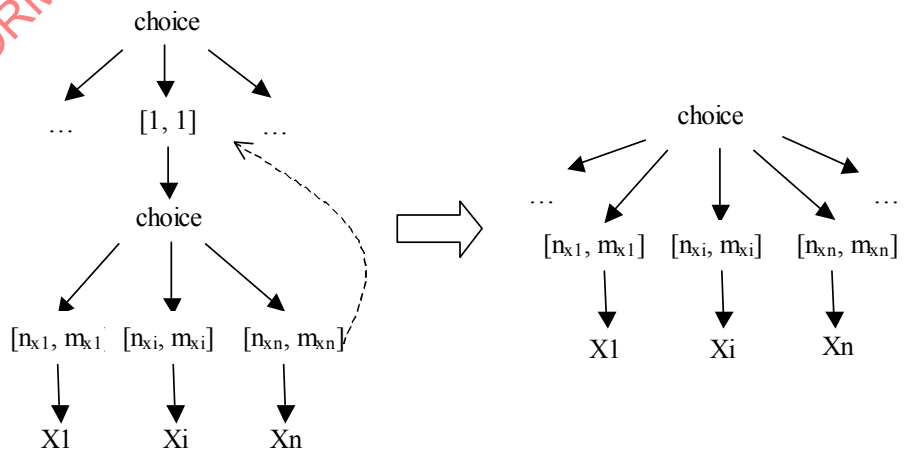

Figure 29 — Choice simplification rule

7.5.2.2.4 Phase 3 - Syntax tree normalization

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

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,

— A wildcard node signature is constructed from the wildcard schema component it is associated to (see XML Schema – Part 1, Chapter 3.10.1) by the concatenation, whitespace separated, of:

  — the ":wildcard" key word

  — the "process contents" property of the wildcard schema component preceded by the character ':' (i.e. ":skip", ":strict" or ":lax"),

  — the "namespace constraint" property represented by a set of whitespace separated keywords, where

    — the 'any', 'not' and 'absent' values are respectively identified by the ":any", ":not", ":absent" keywords,

    — the ":any" or ":not" keywords are always first,

    — the namespaces and, if present the ":absent" keyword, are alphabetically sorted.

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:

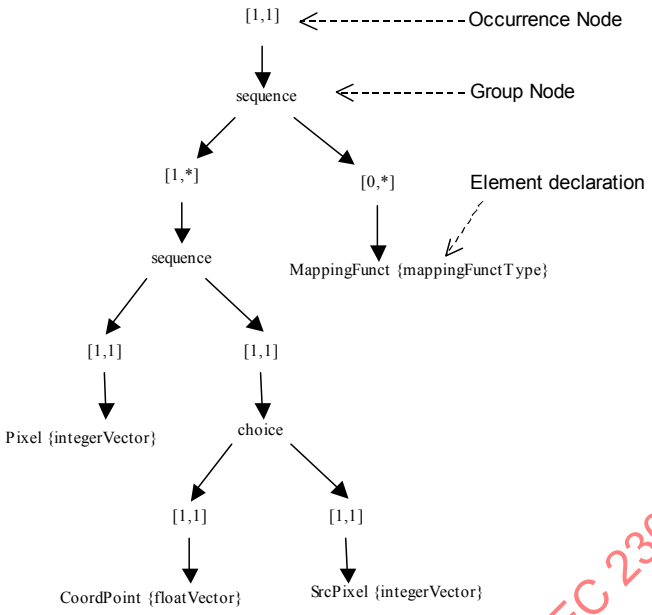

Figure 30 — Example - The syntax tree of coordinate mapping complexType

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

Wildcard signature example

Given the following wildcards defined in a schema which target namespace is "http://www.mpeg7.org/example":

```
<xs:any processContents="skip"/>

<xs:any namespace="##other" processContents="lax"/>

<xs:any namespace="urn:example:namespaceB urn:example:namespaceA"/>

<xs:any namespace="##targetNamespace"/>

<xs:any namespace="##local"/>
```

The signatures of these wildcards are respectively

:wildcard :skip :any

:wildcard :lax :not http://www.mpeg7.org/example

:wildcard :strict urn:example:namespaceA urn:example:namespaceB

:wildcard :strict http://www.mpeg7.org/example

:wildcard :strict :absent

7.5.2.2.5 Phase 4 - Finite state automaton generation

7.5.2.2.5.1 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 modified according to the following rule if the content type of the complex type is 'mixed':
  - a new final state is created. The old final state is linked with the new final state by two transitions, a shunt transition and a mixed transition.
- 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 7.5.2.2.5.2,
- At the end of the process, automata are realized in order to associate binary codes to the "code transitions" (refers to subclause 7.5.2.1).

7.5.2.2.5.2 Phase 4.a - Automata construction

7.5.2.2.5.2.1 Element declaration node automaton

An automaton for an element declaration node can have two different forms depending on the content type of the complexType in which the element is declared:

- If the content type of the complexType is not 'mixed' the automaton is composed of two states, a start state and a final state, and a transition between them. It is used to specify the "element name" / "type" association declared in the complex type definition. The transition is an "element transition" to which the element name of the element declaration node is associated. The target state of the transition is a "type state" to which the element type of the element declaration node is associated.
- If the content type of the complexType is 'mixed', the automaton is composed of three states, a start state, an intermediate and a final one. The start state is linked with the intermediate state by two transitions, a shunt transition and a mixed transition. The intermediate state is linked with the final state by an "element transition" to which the element name of the element declaration node is associated. The final state of the transition is a "type state" to which the element type of the element declaration node is associated.

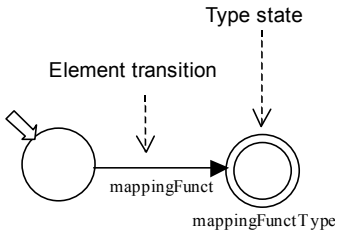

Figure 31 — Example of an element declaration node automaton

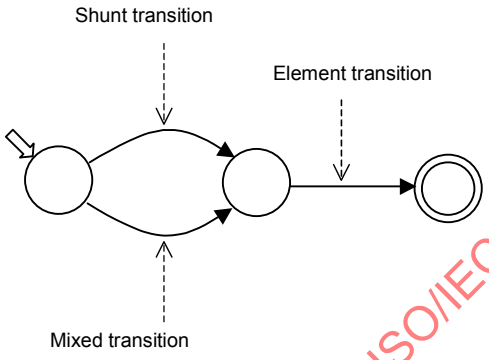

Figure 32 — Example of an element declaration node automaton in case of mixed content model

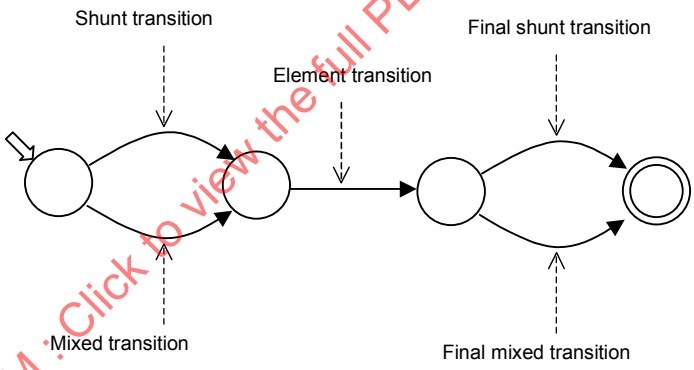

Figure 33 — Example of a complete complex content automaton in case of mixed content model

7.5.2.2.5.2.2 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.

— maxOccurs = 0

- two states are added to the child node automaton : a new start state and a new final state,
- a simple transition is added between the new start state and the new final state,

Note – In case of maxOccurs = 0, the transformation results in discarding the child automaton

— 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.

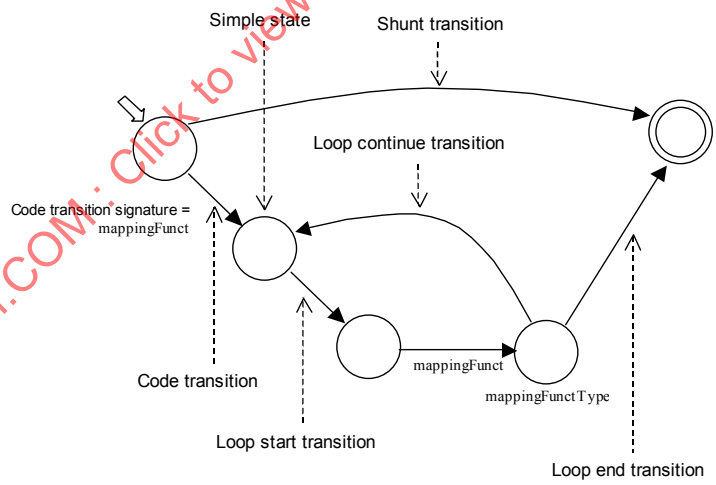

Figure 34 — Example of an occurrence node automaton

7.5.2.2.5.2.3 Choice node automaton

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

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

— Code transitions are added between the new start state and every start state of its child nodes automata. The signatures of these code transitions are equal to the signature of their corresponding child node,

— Simple transitions are added between every final state of its children and its new final state.

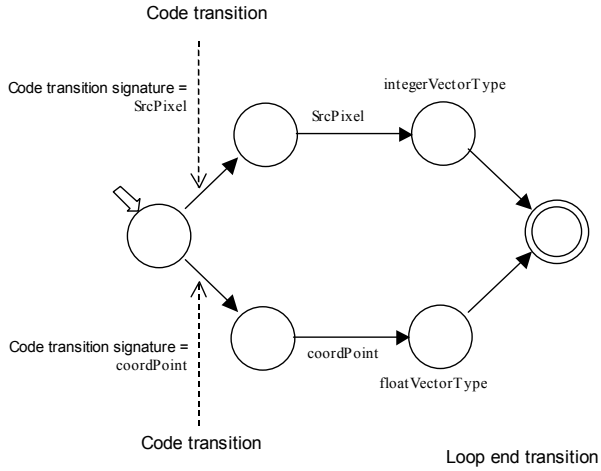

Figure 35 — Example of a choice node automaton

7.5.2.2.5.2.4 Sequence node automaton

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

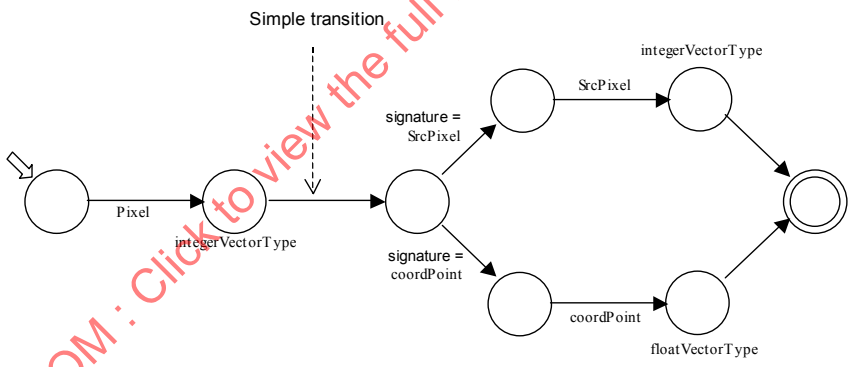

Figure 36 — Example of a sequence node automaton

7.5.2.2.5.2.5 All node automaton

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

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

```
AllConstructionProcedure(AllNodes, PreviousFinalS) {
  Let “FinalStates” be an empty list of automaton states
  Let “ListStates” be an empty list of automaton states
  For each syntax tree node “X” of the list “AllNodes” {
    Generate “XA”, the automaton of “X” using the rules defined in 7.5.2.2.5
```

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

An all node automaton is constructed by the following rules:

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

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

  — `ListOfFinalStates = allConstructionProcedure(AllChildNode, NSS)`

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

7.5.2.2.5.2.6 Wildcard node automaton

The wildcard node automaton can have two different forms depending on the content type of the complexType in which the element is declared:

— If the content type of the complexType is not 'mixed' the automaton is composed of two states, a start state and a final state, and a wildcard transition between them.

If the content type of the complexType is 'mixed', the automaton is composed of three states, a start state, an intermediate and a final one. The start state is linked with the intermediate state by two transitions, a shunt transition and a mixed transition. The intermediate state is linked with the final state by a wildcard transition.

7.5.2.2.5.3 Phase 4.b - Code realization

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

7.5.2.3 Decoding a complex content using a finite state automaton decoder

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