

# ETSI TS 102 822-3-2 V1.4.1 (2007-11)

---

*Technical Specification*

**Broadcast and On-line Services: Search, select, and  
rightful use of content on personal storage systems  
("TV-Anytime");  
Part 3: Metadata;  
Sub-part 2: System aspects in a uni-directional environment**

---

European Broadcasting Union



Union Européenne de Radio-Télévision

EBU·UER



---

Reference

RTS/JTC-TVA-PH1-27-03-02

---

Keywords

broadcasting, content, data, TV, video

**ETSI**

650 Route des Lucioles  
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - NAF 742 C  
Association à but non lucratif enregistrée à la  
Sous-Préfecture de Grasse (06) N° 7803/88

---

**Important notice**

Individual copies of the present document can be downloaded from:

<http://www.etsi.org>

The present document may be made available in more than one electronic version or in print. In any case of existing or perceived difference in contents between such versions, the reference version is the Portable Document Format (PDF). In case of dispute, the reference shall be the printing on ETSI printers of the PDF version kept on a specific network drive within ETSI Secretariat.

Users of the present document should be aware that the document may be subject to revision or change of status. Information on the current status of this and other ETSI documents is available at

<http://portal.etsi.org/tb/status/status.asp>

If you find errors in the present document, please send your comment to one of the following services:

[http://portal.etsi.org/chaicor/ETSI\\_support.asp](http://portal.etsi.org/chaicor/ETSI_support.asp)

---

**Copyright Notification**

No part may be reproduced except as authorized by written permission.  
The copyright and the foregoing restriction extend to reproduction in all media.

© European Telecommunications Standards Institute 2007.

© European Broadcasting Union 2007.

All rights reserved.

**DECT™**, **PLUGTESTS™** and **UMTS™** are Trade Marks of ETSI registered for the benefit of its Members.  
**TIPHON™** and the **TIPHON logo** are Trade Marks currently being registered by ETSI for the benefit of its Members.  
**3GPP™** is a Trade Mark of ETSI registered for the benefit of its Members and of the 3GPP Organizational Partners.

# Contents

Intellectual Property Rights .....	6
Foreword.....	6
Introduction .....	7
1 Scope .....	8
2 References .....	9
3 Definitions, abbreviations and mnemonics .....	10
3.1 Definitions .....	10
3.2 Abbreviations .....	11
3.3 Mnemonics .....	11
4 System mechanisms in a unidirectional environment .....	12
4.1 Overview .....	12
4.1.1 Main features of a unidirectional environment .....	12
4.1.2 Access methods .....	12
4.1.3 Definition of a <i>TV-Anytime</i> metadata description .....	13
4.1.4 <i>TV-Anytime</i> metadata description size .....	13
4.2 Metadata general delivery framework .....	13
4.2.1 Introduction to fragmentation .....	14
4.2.2 Introduction to encoding .....	14
4.2.3 Introduction to encapsulation.....	14
4.2.4 Introduction to Indexing .....	14
4.2.5 Logical decoder architecture .....	14
4.3 Metadata description fragmentation .....	16
4.3.1 TVA metadata fragments .....	17
4.3.1.1 TVAMain fragment.....	17
4.3.1.2 MetadataOriginationInformation fragment .....	19
4.3.1.3 ProgramInformation fragment.....	19
4.3.1.4 GroupInformation fragment .....	20
4.3.1.5 OnDemandProgram and OnDemandService fragment .....	20
4.3.1.6 BroadcastEvent fragment .....	21
4.3.1.7 Schedule fragment.....	21
4.3.1.8 ServiceInformation fragment .....	23
4.3.1.9 CreditsInformation fragments .....	23
4.3.1.9.1 PersonName fragment .....	23
4.3.1.9.2 OrganizationName fragment .....	24
4.3.1.10 PurchaseInformation fragment.....	24
4.3.1.11 Review fragment .....	24
4.3.1.12 User Description information.....	25
4.3.1.13 ClassificationScheme fragments .....	25
4.3.1.13.1 CSAlias fragment .....	25
4.3.1.13.2 ClassificationScheme fragment .....	25
4.3.1.14 Segmentation.....	25
4.3.1.14.1 SegmentInformation fragment.....	26
4.3.1.14.2 SegmentGroupInformation fragment.....	26
4.3.1.15 Package Fragment .....	27
4.3.1.16 Interstitial Campaign Fragment.....	27
4.3.1.17 RMPI Fragment.....	27
4.3.1.18 Coupon Description Fragment .....	27
4.3.1.19 TargetingInformation Fragment.....	28
4.3.1.20 InterstitialBreak Fragment .....	29
4.3.1.21 Rule Fragment.....	29
4.3.1.22 Recording Cache Fragments .....	29
4.3.1.22.1 Request Fragment.....	29
4.3.1.22.2 Replace Fragment.....	30

4.3.1.22.3	Expire Fragment .....	30
4.3.2	Fragment Identification and Versioning .....	30
4.3.3	Element ordering .....	30
4.3.4	TVA access unit.....	30
4.3.5	Use of TVAIDType, TVAIDRefType and TVAIDRefsType .....	30
4.3.6	Use of ID, IDRef, XPath and xml:lang .....	31
4.4	Fragment encoding .....	31
4.4.1	TVA-init message .....	31
4.4.1.1	Overview .....	31
4.4.2	MPEG-7 system profile .....	33
4.4.2.1	DecoderInit.....	33
4.4.2.1.1	UnitSizeCode.....	33
4.4.2.1.2	InitialDescription.....	33
4.4.2.2	FragmentUpdateCommand .....	34
4.4.2.2.1	Guidelines for the use of the FragmentUpdateUnit .....	34
4.4.2.3	ContextMode.....	36
4.4.2.4	TV-Anytime codec .....	37
4.4.2.4.1	Classification scheme wrapper .....	37
4.4.2.4.2	dateTime Codec.....	37
4.4.2.4.3	date codec.....	38
4.4.2.4.4	Zlib optimized decoder.....	38
4.5	Carriage of TV-Anytime data .....	40
4.5.1	Containers .....	40
4.5.1.1	Carriage of containers .....	40
4.5.1.2	Classification of containers .....	40
4.5.1.3	Container identification.....	40
4.5.2	Container versioning.....	41
4.5.2.1	Container syntax .....	41
4.5.2.2	Container map .....	42
4.5.2.2.1	Container map requirements.....	42
4.6	Fragment encapsulation.....	43
4.6.1	Encapsulation format .....	43
4.6.1.1	Encapsulation structure .....	43
4.6.1.2	Moved fragments structure .....	44
4.6.1.3	Fragment_Reference formats .....	45
4.6.1.3.1	Referencing a BiM encoded fragment.....	45
4.6.1.4	Data repository .....	45
4.6.1.4.1	Binary data repository .....	45
4.6.1.5	Alternative Encoding formats .....	46
4.7	Fragment Management .....	46
4.7.1	Fragment Id.....	46
4.7.2	Fragment Add .....	46
4.7.3	Fragment Update .....	46
4.7.4	Fragment Move.....	47
4.7.5	Fragment Delete.....	47
4.8	Indexing.....	47
4.8.1	Introduction.....	47
4.8.2	Requirements .....	47
4.8.3	Carriage of Indexing Information .....	48
4.8.4	Data repository.....	49
4.8.4.1	String repository.....	49
4.8.5	Index structures.....	50
4.8.5.1	Identification of indices.....	50
4.8.5.1.1	Use of Ids.....	51
4.8.5.1.2	Use of XPath .....	51
4.8.5.2	Introduction to the multi-key index.....	52
4.8.5.3	Index List .....	53
4.8.5.4	Index .....	56
4.8.5.4.1	Field Value Ordering.....	56
4.8.5.5	Multi_field_sub_index .....	59
4.8.5.5.1	Single Layer Structures .....	59
4.8.5.5.2	Multi Layer Structures.....	59

4.8.5.6	Fragment locators structure .....	64
4.8.5.7	Fragment_locator formats .....	64
4.8.5.7.1	Referencing fragments in another container .....	64
4.8.5.7.2	Referencing a fragment within the same container .....	65
4.8.6	Binary representation of Simple Types.....	65
4.8.7	Indexes based on Classification Schemes .....	66
4.9	Notion of Validation.....	66
4.10	Extensibility of the <i>TV-Anytime</i> schema .....	66
4.10.1	Introduction.....	66
4.10.2	Extensibility rules .....	67
<b>Annex A (informative): Bibliography .....</b>		<b>69</b>
History .....		73

---

## Intellectual Property Rights

IPRs essential or potentially essential to the present document may have been declared to ETSI. The information pertaining to these essential IPRs, if any, is publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI Web server (<http://webapp.etsi.org/IPR/home.asp>).

Pursuant to the ETSI IPR Policy, no investigation, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

---

## Foreword

This Technical Specification (TS) has been produced by Joint Technical Committee (JTC) Broadcast of the European Broadcasting Union (EBU), Comité Européen de Normalisation ELECTrotechnique (CENELEC) and the European Telecommunications Standards Institute (ETSI).

**NOTE:** The EBU/ETSI JTC Broadcast was established in 1990 to co-ordinate the drafting of standards in the specific field of broadcasting and related fields. Since 1995 the JTC Broadcast became a tripartite body by including in the Memorandum of Understanding also CENELEC, which is responsible for the standardization of radio and television receivers. The EBU is a professional association of broadcasting organizations whose work includes the co-ordination of its members' activities in the technical, legal, programme-making and programme-exchange domains. The EBU has active members in about 60 countries in the European broadcasting area; its headquarters is in Geneva.

European Broadcasting Union  
 CH-1218 GRAND SACONNEX (Geneva)  
 Switzerland  
 Tel: +41 22 717 21 11  
 Fax: +41 22 717 24 81

The present document is part 3, sub-part 2 of a multi-part deliverable covering Broadcast and On-line Services: Search, select and rightful use of content on personal storage systems ("*TV-Anytime*"), as identified below:

- Part 1: "Phase 1 - Benchmark Features";
- Part 2: "Phase 1 - System description";
- Part 3: "Metadata";**
  - Sub-part 1: "Phase 1 - Metadata schemas";
  - Sub-part 2: "System aspects in a uni-directional environment";**
  - Sub-part 3: "Phase 2 - Extended Metadata Schema";
  - Sub-part 4: "Phase 2 - Interstitial metadata";
- Part 4: "Phase 1 - Content referencing";
- Part 5: "Phase 1 - Rights Management and Protection (RMP)";
- Part 6: "Delivery of metadata over a bi-directional network";
- Part 7: "Phase 1 - Bi-directional metadata delivery protection";
- Part 8: "Phase 2 - Interchange Data Format";
- Part 9: "Phase 2 - Remote Programming".

---

## Introduction

The present document is based on a submission by the *TV-Anytime* forum (<http://www.TV-Anytime.org>).

"*TV-Anytime*" (TVA) is a full and synchronized set of specifications established by the *TV-Anytime* Forum. TVA features enable the search, selection, acquisition and rightful use of content on local and/or remote personal storage systems from both broadcast and online services.

TS 102 822-1 [7] and TS 102 822-2 [8] set the context and system architecture in which the standards for Metadata, Content referencing, Bi-directional metadata and Metadata protection are to be implemented in the *TV-Anytime* environment. TS 102 822-1 [7] provides benchmark business models against which the *TV-Anytime* system architecture is evaluated to ensure that the specification enable key business applications. TS 102 822-2 [8] presents the *TV-Anytime* System Architecture. These two documents are placed ahead of the others for their obvious introductory value. Note that these first two documents are largely informative, while the remainder of the series is normative.

The features are supported and enabled by the specifications for Metadata (TS 102 822-3-1 [9], TS 102 822-3-2 (the present document), TS 102 822-3-3 [10] and TS 102 822-3-4 [11]), Content Referencing (TS 102 822-4 [12]), Rights Management (TS 102 822-5-1 [13] and TS 102 822-5-2 [14]), Bi-directional Metadata Delivery (TS 102 822-6-1 [15], TS 102 822-6-2 [16] and TS 102 822-6-3 [17]) and Protection (TS 102 822-7 [18]), Interchange Data Format (TS 102 822-8 [19]) and Remote Programming (TS 102 822-9 [20]). All Phase 1 and 2 Features listed in TS 102 822-1 [7] are enabled by the normative *TV-Anytime* tools specifications.

The metadata specifications TS 102 822-3-1 [9], TS 102 822-3-3[10], TS 102 822-3-4 [11], TS 102 822-8 [19] and TS 102 822-9 [20] address the description language, structure and semantics of *TV-Anytime* metadata descriptions. The present document introduces new network agnostic technologies used to process these descriptions for the purpose of transmission in a unidirectional environment.

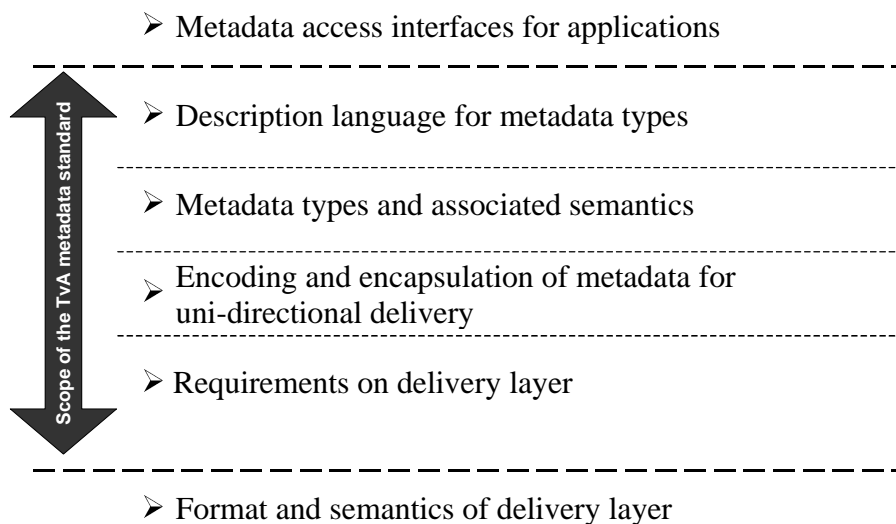
# 1 Scope

The present document is one in a series of Technical Specification documents produced by the *TV-Anytime* Forum. These documents establish the fundamental specifications for the services, systems and devices that will conform to the *TV-Anytime* standard, to a level of detail that is implementable for compliant products and services.

TS 102 822-1 [7] and TS 102 822-2 [8] set the context and system architecture in which the standards for Metadata, Content referencing, Bi-directional metadata and Metadata protection are to be implemented in the *TV-Anytime* environment. TS 102 822-1 [7] provides benchmark business models against which the *TV-Anytime* system architecture is evaluated to ensure that the specification enable key business applications. TS 102 822-2 [8] presents the *TV-Anytime* System Architecture. These first two documents are largely informative, while the remainder of the series is normative.

Although each in the series of documents is intended to stand alone, a complete and coherent sense of the *TV-Anytime* system standard can be gathered by reading all the specification documents in numerical order.

The following diagram depicts the combined scope of the *TV-Anytime* Specifications on Metadata TS 102 822-3-1 [9], TS 102 822-3-3 [10], TS 102 822-3-4 [11], TS 102 822-8 [19], TS 102 822-9 [20] and the present document "System Aspects in a Unidirectional Environment".



**Figure 1: Overview of the scope of TVA Specification on Metadata and System**

The metadata specifications TS 102 822-3-1 [9], TS 102 822-3-3[10], TS 102 822-3-4 [11], TS 102 822-8 [19] and TS 102 822-9 [20] address the description language, structure and semantics of *TV-Anytime* metadata descriptions. The present document introduces new network agnostic technologies used to process these descriptions for the purpose of transmission in a unidirectional environment.

The actual format and semantics of the delivery layer are specific to the particular uni-directional environment in which *TV-Anytime* is deployed. However, in order for the encoding and encapsulation mechanisms to operate as intended the delivery layer must meet certain requirements. These normative requirements are defined in annex B of TS 102 822-2 [8].



---

## 2 References

The following documents contain provisions which, through reference in this text, constitute provisions of the present document.

- References are either specific (identified by date of publication and/or edition number or version number) or non-specific.
- For a specific reference, subsequent revisions do not apply.
- For a non-specific reference, the latest version applies.

Referenced documents which are not found to be publicly available in the expected location might be found at <http://docbox.etsi.org/Reference>.

NOTE: While any hyperlinks included in this clause were valid at the time of publication ETSI cannot guarantee their long term validity.

[1] ISO/IEC 10646 (2003): "Information technology - Universal Multiple-Octet Coded Character Set (UCS)".

[2] XML Schema, W3C Recommendations (version 20010502).

NOTE: Available at: <http://www.w3.org/TR/2001/REC-xmlschema-0-20010502>,  
<http://www.w3.org/TR/2001/REC-xmlschema-1-20010502>,  
<http://www.w3.org/TR/2001/REC-xmlschema-2-20010502>.

[3] IEEE Standard for Binary Floating-Point Arithmetic, Std 754-1985 Reaffirmed 1990.

[4] ISO/IEC 15938-1 (2002): "Information technology - Multimedia content description interface - Part 1: Systems".

[5] ISO/IEC 15938-2 (2002): "Information technology - Multimedia content description interface - Part 2: Description definition language".

[6] Zlib: "The Zlib API".

NOTE: Available at: <http://www.gzip.org/zlib>.

[7] ETSI TS 102 822-1: "Broadcast and On-line Services: Search, select, and rightful use of content on personal storage systems ("TV-Anytime"); Part 1: Benchmark Features".

[8] ETSI TS 102 822-2: "Broadcast and On-line Services: Search, select, and rightful use of content on personal storage systems ("TV-Anytime Phase 1"); Part 2: Phase 1 - System description".

[9] ETSI TS 102 822-3-1: "Broadcast and On-line Services: Search, select, and rightful use of content on personal storage systems ("TV-Anytime"); Part 3: Metadata; Sub-part 1: Phase 1 - Metadata schemas".

[10] ETSI TS 102 822-3-3: "Broadcast and On-line Services: Search, select, and rightful use of content on personal storage systems ("TV-Anytime"); Part 3: Metadata; Sub-part 3: Phase 2 - Extended Metadata Schema".

[11] ETSI TS 102 822-3-4: "Broadcast and On-line Services: Search, select, and rightful use of content on personal storage systems ("TV-Anytime"); Part 3: Metadata; Sub-part 4: Phase 2 - Interstitial metadata".

[12] ETSI TS 102 822-4: "Broadcast and On-line Services: Search, select, and rightful use of content on personal storage systems ("TV-Anytime Phase1"); Part 4: Phase 1 - Content referencing".

[13] ETSI TS 102 822-5-1: "Broadcast and On-line Services: Search, select, and rightful use of content on personal storage systems ("TV-Anytime"); Part 5: Rights Management and Protection (RMP) Sub-part 1: Information for Broadcast Applications".

- [14] ETSI TS 102 822-5-2: "Broadcast and On-line Services: Search, select, and rightful use of content on personal storage systems ("TV-Anytime"); Part 5: Rights Management and Protection (RMP) Sub-part 2: RMPI binding".
- [15] ETSI TS 102 822-6-1: "Broadcast and On-line Services: Search, select, and rightful use of content on personal storage systems ("TV-Anytime"); Part 6: Delivery of metadata over a bi-directional network; Sub-part 1: Service and transport".
- [16] ETSI TS 102 822-6-2: "Broadcast and On-line Services: Search, select, and rightful use of content on personal storage systems ("TV-Anytime"); Part 6: Delivery of metadata over a bi-directional network; Sub-part 2: Phase 1 - Service discovery".
- [17] ETSI TS 102 822-6-3: "Broadcast and On-line Services: Search, select, and rightful use of content on personal storage systems ("TV-Anytime"); Part 6: Delivery of metadata over a bi-directional network; Sub-part 3: Phase 2 - Exchange of Personal Profile".
- [18] ETSI TS 102 822-7: "Broadcast and On-line Services: Search, select, and rightful use of content on personal storage systems ("TV-Anytime Phase 1"); Part 7: Bi-directional metadata delivery protection".
- [19] ETSI TS 102 822-8: "Broadcast and On-line Services: Search, select, and rightful use of content on personal storage systems ("TV-Anytime"); Part 8: Phase 2 - Interchange data format".
- [20] ETSI TS 102 822-9: "Broadcast and On-line Services: Search, select, and rightful use of content on personal storage systems ("TV-Anytime"); Part 9: Phase 2 - Remote Programming".
- [21] ISO/IEC 8601: "Data elements and interchange formats - Information interchange - Representation of dates and times".

---

## 3 Definitions, abbreviations and mnemonics

### 3.1 Definitions

For the purposes of the present document, the following terms and definitions apply:

**application:** specific set of functions running on the PDR

NOTE: Some applications use metadata, either automatically or under *consumer* control.

**head end:** source of emission of the transport stream where metadata is inserted

**metadata:** data about content

EXAMPLE: The title, genre and summary of a television programme.

NOTE: In the context of *TV-Anytime*, metadata also includes consumer profile and history data.

**MPEG:** ongoing effort by the Motion Pictures Expert Group (working group SC29 WG11 of ISO/IEC) to specify a standard set of content-related metadata applicable to a broad range of applications

NOTE: Defined by *TV-Anytime* in TS 102 822-3-1 [9].

**partial description:** reconstructed portion of the *TV-Anytime* metadata description in the PDR obtained after having decoded a subset of the metadata fragment stream

NOTE: The present document instantiates the *TV-Anytime* schema and must therefore be schema valid with respect to it.

**segment:** continuous portion of a piece of content, for example a single news topic in a news programme

**segmentation:** process of creating segments from a piece of content

**transport stream:** transport stream is made up of the A/V and/or the *TV-Anytime* data streams including the metadata

**TVA access unit:** container which holds one or more TVA fragments when carried over the transport stream

**TVA fragment:** self-consistent atomic portion of a metadata description sent to the decoder

**TVA metadata description:** actual document instantiating the TVA schema which is to be sent to the PDR

NOTE: The present document may be subject to partial updates in time using the fragmentation mechanism. In any case, it must be schema valid with respect to the TVA schema.

**TVA metadata fragment stream:** set of the many TVA fragments constituting a single TVA metadata description and inserted in the transport stream received by the decoder

**TVA MPEG-7 profile:** implementation profile of the ISO/IEC 15938-1 [4] standard, adopted by *TV-Anytime* for the encoding of the metadata description as specified in the present document

**TVA schema:** set of rules describing the syntax and semantics of the metadata

## 3.2 Abbreviations

For the purposes of the present document, the following abbreviations apply:

AU	Access Unit
BiM	Binary format of MPEG-7
CRID	Content Reference IDentifier

NOTE: An identifier for content that is independent of its location.

DDL	Description Definition Language
-----	---------------------------------

NOTE: The language used to define description schemes in MPEG-7 (see ISO/IEC 15938-2 [5]).

DL	Delivery Layer
ECG	Electronic Content Guide

NOTE: A means of presenting available content to the consumer, allowing selection of desired content.

GMT	Greenwich Mean Time
IPR	Intellectual Property Rights
MPEG	Motion Pictures Expert Group
PDR	Personal Digital Recorder
TVA	<i>TV-Anytime</i>
UI	User Interface
URI	Uniform Resource Identifier
UTF	Universal character set Transformation Format
XML	eXtensible Markup Language
XPath	XML Path Language

## 3.3 Mnemonics

For the purposes of the present document, the following mnemonics are defined to describe the different data types used.

**bslbf:** Bit string, left bit first, where "left" is the order in which bit strings are written. Bit strings are generally written as a string of 1 s and 0 s within single quote marks, e.g. "1000 0001". Blanks within a bit string are for ease of reading and have no significance.

**reserved:** A binary syntax element whose length is indicated in the syntax table. The value of each bit of this field shall be set to "1". These bits may be used in the future for TVA defined extensions.

**uimsbf:** Unsigned integer, most significant bit first (big-endian).

**vlumbsbf8:** Variable length coded unsigned integer, most significant bit first. The size of vlumbsbf8 is a multiple of one byte. The first bit of each byte specifies if set to "1" that another byte is present for this vlumbsbf8 code word. The unsigned integer is encoded by the concatenation of the seven least significant bits of each byte belonging to this vlumbsbf8 code word.

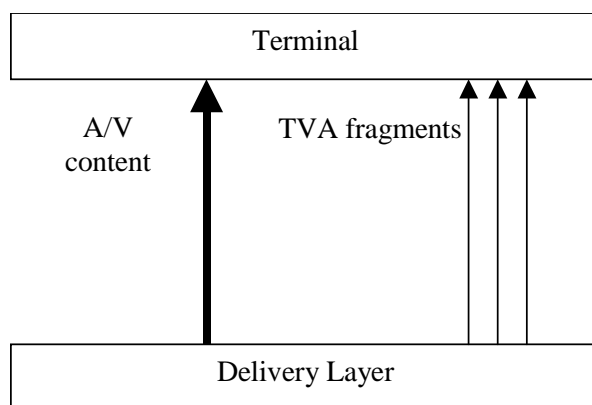
**vlumbsbf5:** Variable length code unsigned integer, most significant bit first. The first n bits which are 1 except for the n-th bit which is 0, indicates that the integer is encoded by n times 4 bits.

## 4 System mechanisms in a unidirectional environment

### 4.1 Overview

#### 4.1.1 Main features of a unidirectional environment

Unidirectional environments deliver content and metadata from the transmitting device (head-end) to the terminal device Personal Digital Recorder (PDR) over a one-way link. No communication is possible from the PDR to the head-end.



**Figure 2: Unidirectional environment**

The restrictions imposed by a unidirectional environment mean that a *TV-Anytime* metadata delivery system needs to have the following attributes.

All the *TV-Anytime* metadata descriptions required by the applications running on the PDR will need to be available in the streams provided by the DL to the terminal at the time when those applications need them.

Since the head-end is never advised that the PDR is connected and no acknowledgement is provided to indicate that the data has been correctly transmitted, the data will need to be broadcast cyclically during the time that they are potentially needed by the applications.

Unidirectional environments are in practice limited in the amount of bandwidth that can be allocated to metadata, the size occupied by the *TV-Anytime* metadata description in the transport stream needs to be kept as small as possible.

#### 4.1.2 Access methods

Resources available to PDRs will vary, as will the nature of different unidirectional environments. Because of this it is anticipated that there are a number of ways in which a receiver may wish to acquire and navigate *TV-Anytime* metadata descriptions. The present document has been designed to support the following methods of acquisition:

- **Method 1:** Acquire from the metadata stream and cache the data to disk with the receiver provides its own methods of navigation.
- **Method 2:** Use the TVA Indexing solution to enable online navigation of the metadata stream.

- **Method 3:** Cache both TVA indexing information and data to disk to provide an enhanced version of method 2.

### 4.1.3 Definition of a *TV-Anytime* metadata description

A *TV-Anytime* metadata *description* is the present document, which is to be sent to the PDR. It is produced in accordance with the schema specified in the *TV-Anytime* metadata specification (see TS 102 822-3-1 [9], clause 6, TS 102 822-3-3 [10], TS 102 822-3-4 [11], TS 102 822-8 [19], TS 102 822-9 [20]). *TV-Anytime* has selected the MPEG-7 DDL language [5], based on XML schema [2], to define its schema. MPEG-7 DDL is used to define the syntax, the types and default values that comprise the *TV-Anytime* specification. A *TV-Anytime* description can therefore be represented as an XML document instantiating the *TV-Anytime* schema and containing a single root element called *TVAMain*. Within the *TVAMain* element any number of *TV-Anytime* types can be instantiated. Such a description can then be declared schema valid with respect to the TVA schema.

### 4.1.4 *TV-Anytime* metadata description size

In many systems a *TV-Anytime* description can become very large. As an example, a valid *TV-Anytime* document could contain all the descriptive data supplied by a single provider to feed an ECG for the next 15 days. The data-set could consist of a set of descriptions for the whole list of programmes, series and groups of contents which are going to be played out for this period, the full schedule of all the related events, the description of the channels on which they are going to be broadcast, the classification scheme tables, the cast list tables, etc. This entire dataset is contained in the same document.

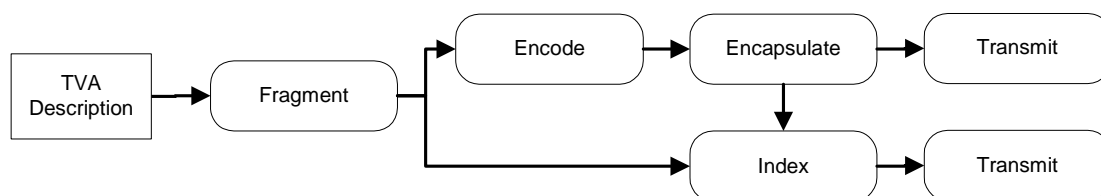
Sending this dataset as an XML document is inefficient for the following reasons:

- XML is a verbose textual format, making an XML document carrying this amount of data very large. In an environment with restricted bandwidth this will result in slow download times.
- Not all of the information carried will be relevant to the terminal at any one instance. Some part of the description may require to be accessed often whilst other parts may only be accessed occasionally.
- Some parts of an XML document may need to be updated without affecting other parts (e.g. modification of a single schedule event if one programme is replaced by another).
- Some parts of an XML document may need to be accessed and updated more often and more efficiently than the rest of the data set (e.g. dynamic segmentation metadata).

In order to overcome the problems associated with delivering *TV-Anytime* metadata as a single, homogenous document, *TV-Anytime* defines *fragmentation*, *encoding*, *encapsulation* and *indexing* mechanisms to apply to the *TV-Anytime* metadata descriptions before being transmitted to the terminal in a unidirectional environment.

## 4.2 Metadata general delivery framework

The delivery of *TV-Anytime* metadata can be viewed as five distinct processes.



**Figure 3: Processes associated with delivery of metadata**

### 4.2.1 Introduction to fragmentation

Fragmentation is the generic decomposition mechanism of a TVA metadata description into self-consistent units of data, called *TVA fragments*. The definition of TVA fragment types allowed by *TV-Anytime* can be found in clause 4.3.

The *self-consistency* capability of a TVA fragment means that:

- Fragments can be obtained in a random order.
- Each fragment can be transmitted and updated independently.
- After a fragment has been received and decoded, the resulting partial description is valid with respect to the TVA schema.

The set of all TVA fragments constituting a single TVA metadata description is transmitted to the terminal within a unidirectional environment as a stream of data. This stream is termed a *TVA metadata fragment stream*.

A TVA metadata fragment stream shall always be accessed via the TVA-init message which represents the entry point, as described in clause 4.4.1, followed by the TVA fragment containing the TVAMain root element, as described in clause 4.3.1.1. Optionally, the TVAMain fragment may be included in the TVA-init message as indicated in clause 4.4.1.

The transport of TVA metadata shall be fragmented as defined in the present document.

### 4.2.2 Introduction to encoding

To enable the efficient (in terms of bandwidth, navigability and updating) delivery of data within a unidirectional environment, it is necessary to represent the TVA metadata fragments in a binary format.

*TV-Anytime* has chosen the MPEG-7 BiM method as defined in ISO/IEC 15938-1 [4] (MPEG-7 Systems part) as the preferred method that would facilitate wide interoperability. However *TV-Anytime* appreciates that in some controlled environments, it may be desirable to enable the delivery of metadata using alternate encoding systems. To allow this, appropriate hooks are provided where necessary and the means to indicate the method of encoding used.

### 4.2.3 Introduction to encapsulation

Once the fragments have been encoded they need to be *encapsulated*. The process of encapsulation provides further information to enable a receiving device to manage a set of transmitted TVA fragments. A receiver needs to be able to uniquely identify a fragment within the TVA metadata fragment stream and also to be able to identify when the data within a fragment changes. This information is provided by the encapsulation layer.

For the transmission of fragments, the encapsulation mechanism shall be used.

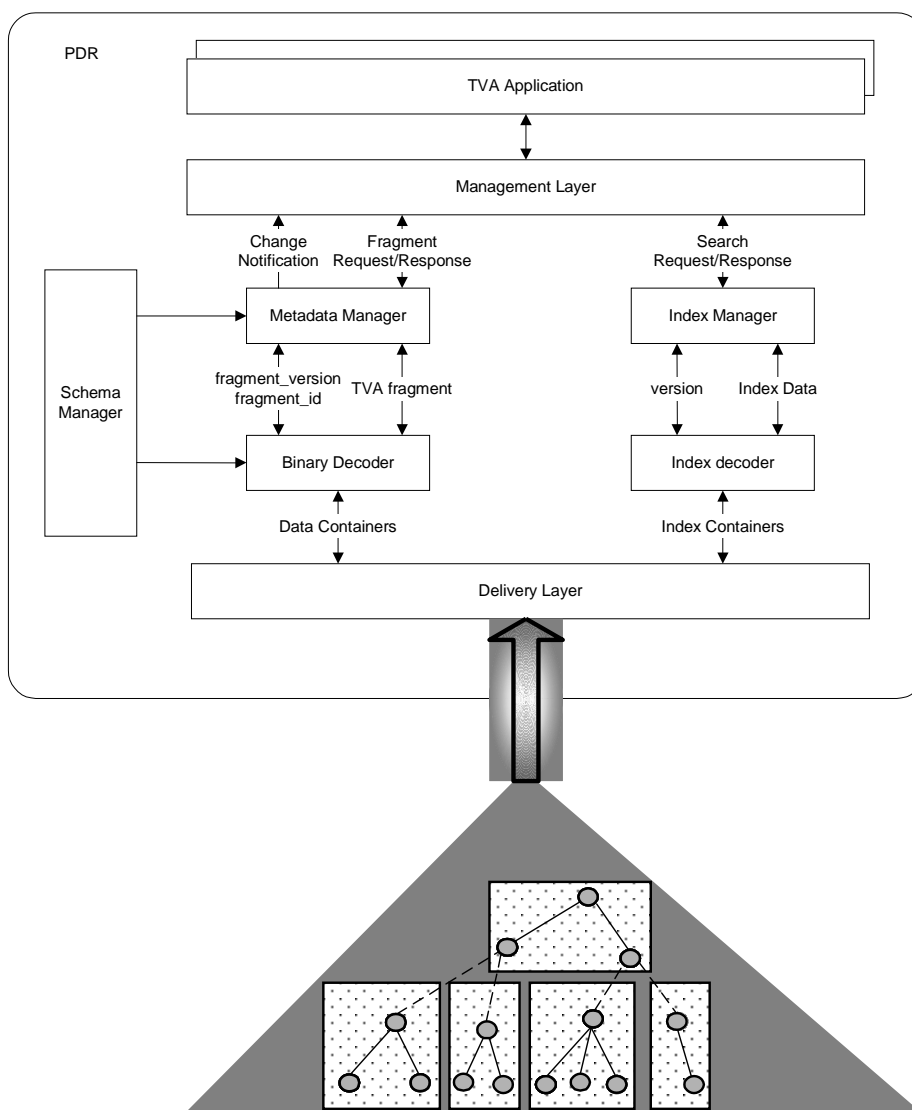
### 4.2.4 Introduction to Indexing

Within a *TV-Anytime* metadata fragment stream there are likely to be many hundreds of fragments. Due to the volume of information necessary to provide the enhanced functionality expected of a PDR it is important that there is an efficient mechanism for locating information from within the TVA metadata fragment stream. Indexing provides this functionality by allowing multiple views on the TVA metadata description. In addition to enable a device to quickly find a fragment of interest, indices can also for example be used to provide enhanced UI functionality such as an A-Z listing for Content Titles, Genre Listing etc.

Indexing is an **optional** part of the present document, however it is seen as a powerful mechanism when TVA metadata is to be delivered to receivers that have limited processing and storage capabilities.

### 4.2.5 Logical decoder architecture

The components of the logical decoder architecture are shown below. Each of these components has a specific role in the process of reconstructing a TVA description or in the navigation of the TVA fragments via the indexing system.



**Figure 4: Functional metadata processing architecture in a unidirectional environment**

- Delivery Layer:** The delivery layer provides the mechanism on which the TVA metadata fragment stream is transmitted. The system shall locate the TVA-init message within the delivery layer, to initialize the metadata management system. The TVA-init provides information about the encoding of the TVA fragments and whether Indexing data is available. Once the metadata management system has been configured, the system is ready to process TVA fragments and indexing data. TVA fragments are encapsulated within a data container. Depending on the caching model used one or more data containers e.g. acquire container "N" or acquire all data containers - are requested by the management system.
- Binary Decoder:** The binary decoder takes a data container and decodes the TVA fragment and delivers the decoded TVA fragment along with the fragments version and unique identifier to the Metadata Manager. The schema to which all fragments within the TVA metadata fragment stream shall conform is defined within the DecoderInit part of the TVA-init message.
- The Schema Manager:** This is a black box component in the terminal which provides the Binary Decoder with details about the schemas, declared in the TVA-init message and instantiated by the related TVA metadata description. It is a requirement that the terminal knows the schema used to produce a TVA metadata description prior to processing the TVA metadata fragment stream. **However, the method in which a PDR acquires this schema is out of scope for the current version of the TV-Anytime metadata standard.**
- Metadata Manager:** This component is responsible for managing requests from the Management Layer for TVA fragments and also for notifying the Management layer of changes to fragments previously acquired from the TVA metadata fragment stream.

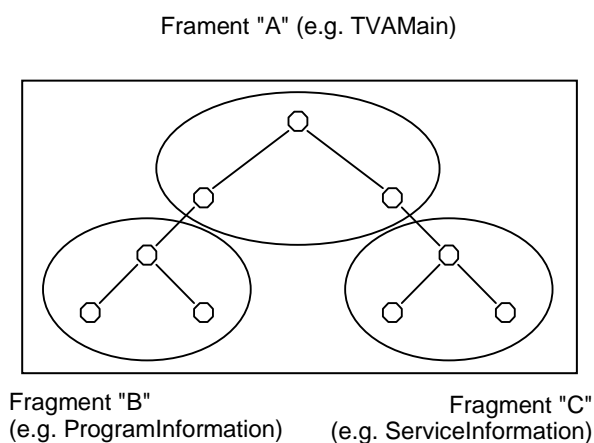
- **Management Layer:** The role of the management layer is very much dependant on the functionality of the TVA Application which is interacting with this layer. In some instances it will just notify the Application when a change as occurred to the TVA fragment stream e.g. new fragment, deleted fragment takes. In other cases it will ask for a specific fragment, which involves an index lookup. The Management Layer is then responsible for using the Index to locate a fragment and then using the Metadata manager to load the fragment from the TVA metadata fragment stream.
- **Index Decoder:** This component takes an Index container from the delivery layer and makes the raw data available to the Index Manager, along with the index container version.
- **Index Manager:** This component takes search requests from the Management Layer and returns a set of matching references to fragments, which the Management layer then sends to the Metadata manager for loading. To perform the search the appropriate Index containers are loaded and the search performed using the supplied parameters.

### 4.3 Metadata description fragmentation

To enable the efficient delivery, updating and navigation of a *TV-Anytime* metadata description, a number of normative TVA fragment types have been defined.

A fragment is the ultimate atomic part of a *TV-Anytime* metadata description that can be transmitted independently to a terminal. A fragment shall be self consistent in the sense that:

- It shall be capable of being updated independently from other fragments.
- The way it is decoded, processed and accessed shall be independent from the order in which it is transmitted relative to other fragments.
- The decoding of a fragment and its addition to the partial description shall give a *TV-Anytime* schema valid description. Note that a partial description must have at least the fragment delivering the root element (TVAMain).



**Figure 5: Fragmentation of a *TV-Anytime* metadata description**

When an update occurs to one or more elements or attributes within a previously transmitted fragment, the entire fragment must be transmitted again i.e. no mechanisms for sub-fragment updates are provided.



### 4.3.1 TVA metadata fragments

The *TV-Anytime* metadata description can be split into a number of standardized types of self contained fragments. These *TV-Anytime* normative fragments are defined as follows.

#### 4.3.1.1 TVAMain fragment

The TVAMain fragment is special since every TVA fragment stream must contain one instance of this fragment type. The TVAMain fragment contains the TVAMain / ExtendedTVAMain root element plus a limited range of child nodes.

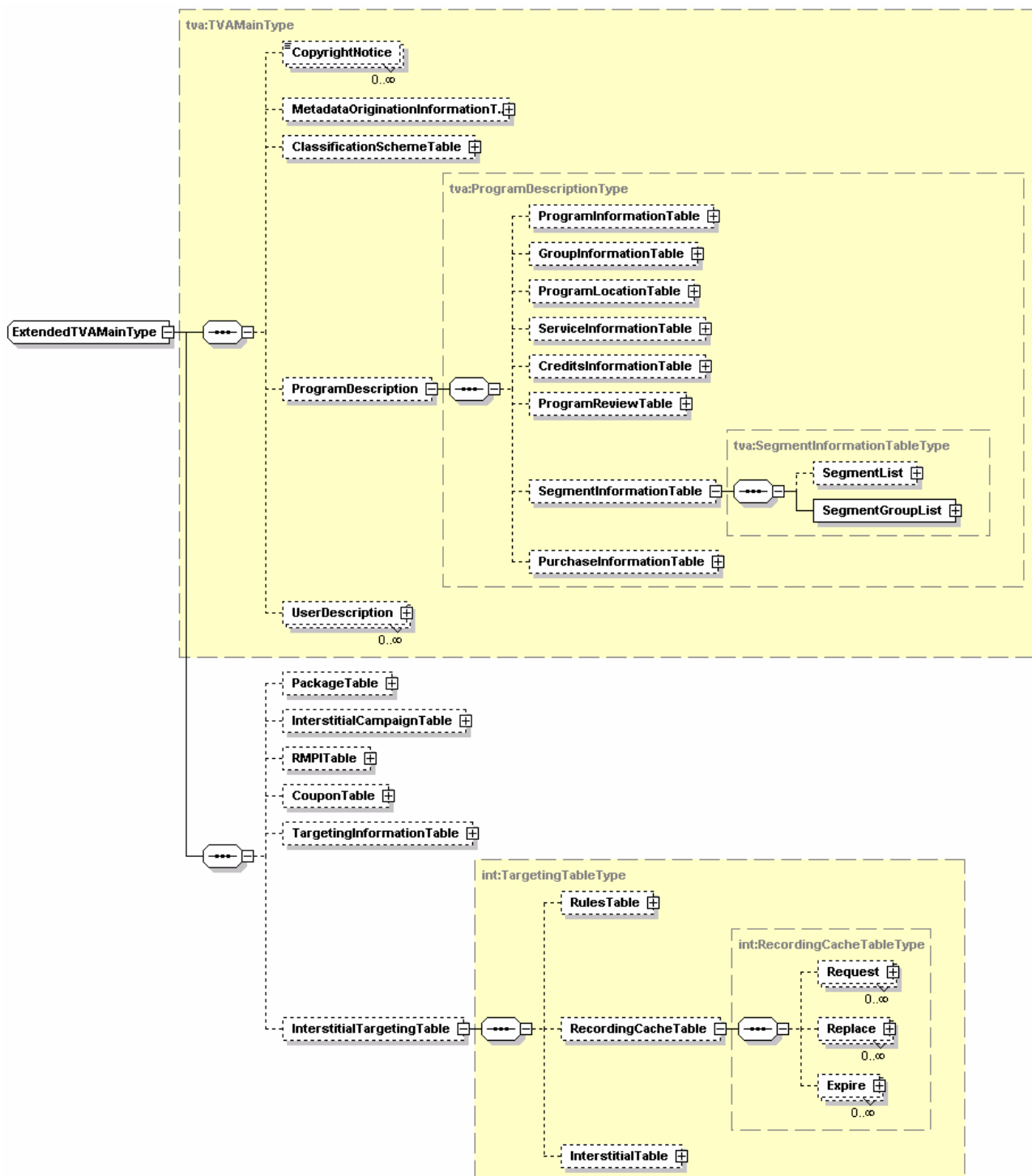


Figure 6: UML-like representation of a TVAMain fragment

As can be seen from the representation in figure 6 the TVAMain fragment shall contain child elements down to and including the following:

- MetadataOriginationInformationTable.
- ClassificationSchemeTable.
- ProgramInformationTable.
- GroupInformationTable.
- ProgramLocationTable.
- ServiceInformationTable.
- CreditsInformationTable.
- ProgramReviewTable.
- SegmentInformationTable:
  - SegmentList; SegmentGroupList.
- PurchaseInformationTable.
- PackageTable.
- RMPITable.
- CouponTable.
- TargetingInformationTable.
- InterstitialTargetingTable:
  - RulesTable.
  - RecordingCacheTable.
  - InterstitialTable.

The presence of and number of each of these types shall conform to the *TV-Anytime* schema. Elements that are children of these element types form fragments of their own and so shall not be included within the TVAMain / ExtendedTVAMain fragment.

It should be noted that updates to TVAMain / ExtendedTVAMain would typically be infrequent and an update to this fragment would cause the decoder to be re-initialized.

### 4.3.1.2 MetadataOriginationInformation fragment

The MetadataOriginationInformation element, as defined in TS 102 822-3-1 [9], shall form a single TVA fragment.

The MetadataOriginationInformation element is a child of the MetadataOriginationInformationTable, which is a member of the TVAMain fragment.

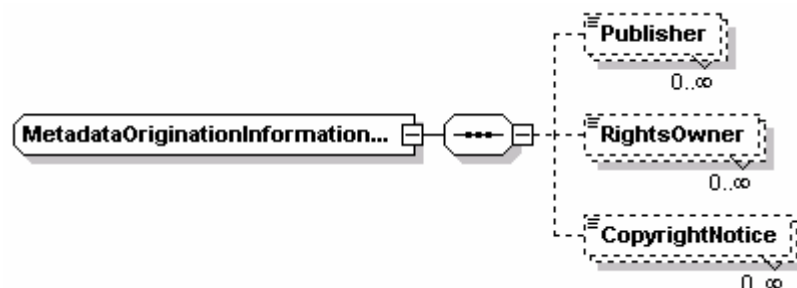


Figure 7: UML-like representation of a MetadataOriginationInformation fragment

### 4.3.1.3 ProgramInformation fragment

A ProgramInformation fragment is one of the key attractor types of a TVA system (TS 102 822-3-1 [9]), it carries an element of type ProgramInformationType and all child nodes thereof. This type contains descriptive information about a piece of content, the content being identified by a CRID.

The ProgramInformationElement is a child element of the ProgramInformationTable, which forms part of the TVAMain fragment.

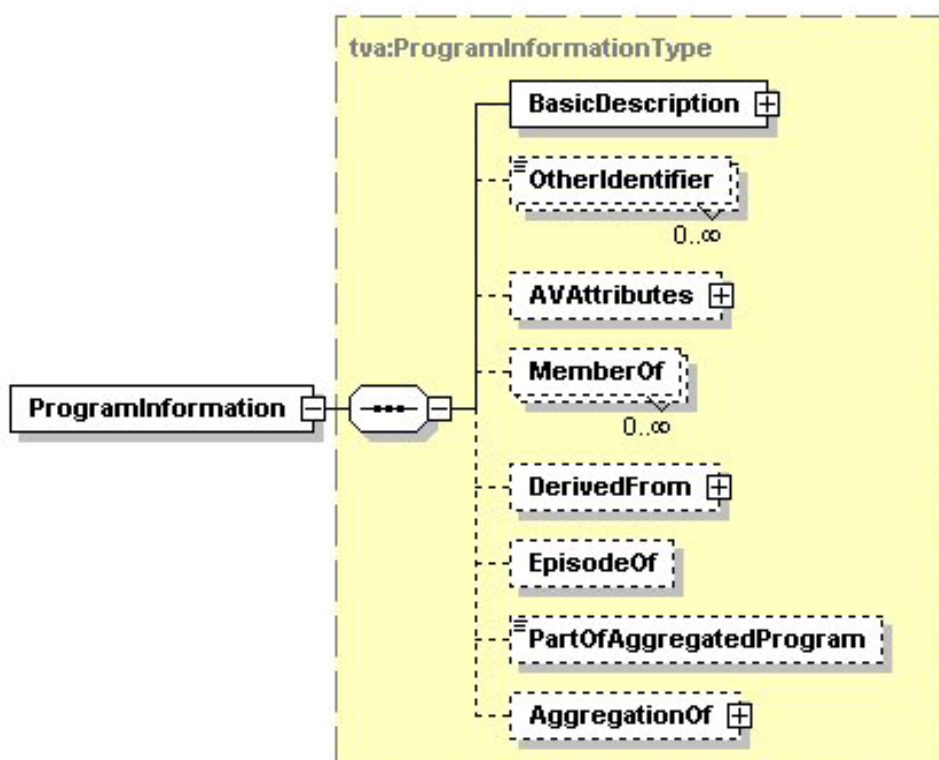


Figure 8: UML-like representation of a ProgramInformation fragment

The ProgramInformation fragment is completely self-contained with one exception. Within the CreditsItem, which is a child element of the CreditList element in the BasicDescription element, the optional PersonNameIDRef and OrganizationNameIDRef elements use TVAIDRefType attributes to reference TVAIDType attributes within corresponding entries within the CreditsInformationTable. This enables optimization in cases where a Credit Item description is common to more than one programme. In such a case, a CreditsItem only needs to be defined once e.g. for all the episodes of a Soap Opera. The guidelines given for the use of TVAIDType/TVAIDRefType in clause 4.3.5 should however be taken into account when assigning and using these ID values.

#### 4.3.1.4 GroupInformation fragment

A GroupInformation fragment contains an element of type GroupInformationType and all child nodes thereof. This contains descriptive information about a conceptual content group e.g. Series, Serial, Collection, etc. The GroupInformationElement is a child element of the GroupInformationTable, which forms part of the TVAMain fragment (TS 102 822-3-1 [9]).

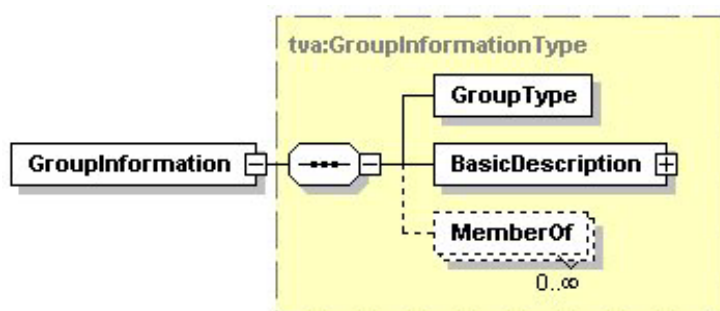


Figure 9: UML-like representation of a GroupInformation fragment

The GroupInformation fragment is completely self contained except for that described for the ProgramInformation fragment, namely within the CreditsItem, the optional PersonNameIDRef and OrganizationNameIDRef elements which the TVAIDRefType attribute which points to an entry within the CreditInformationTable. The guidelines given for the use of TVAIDType/TVAIDRefType in clause 4.3.5 should thus be taken into account when assigning and using these ID values.

#### 4.3.1.5 OnDemandProgram and OnDemandService fragment

Within a unidirectional environment the OnDemandProgramType and the OnDemandServiceType are seen to be of limited use.

However if a broadcaster wishes to transmit OnDemand program location information, a complete OnDemandProgram or an OnDemandService element instantiating respectively one of these types shall form a single fragment. Both the OnDemandProgram and the OnDemandService elements are child members of the ProgramLocationTable, which forms part of the TVAMain fragment.

An attribute of type TVAIDRefType (TS 102 822-3-1 [9]) is used by the serviceIDRef attribute within the OnDemandServiceType to reference the associated service information. It points to an element of type ServiceInformationType having a serviceId attribute, with a value equal to the serviceIDRef. This latter value shall be unique within the TVA metadata description. And the guideline in clause 4.3.5 on the assignment and use of TVAIDType/TVAIDRefType values should be taken into account when dealing with these values.

### 4.3.1.6 BroadcastEvent fragment

The BroadcastEventType has been designed to represent dynamic events in an environment where frequent updates are required (TS 102 822-3-1 [9]).

A BroadcastEvent fragment is thus an instance of the BroadcastEventType, a child element of the ProgramLocationTable, which forms part of the TVAMain fragment.

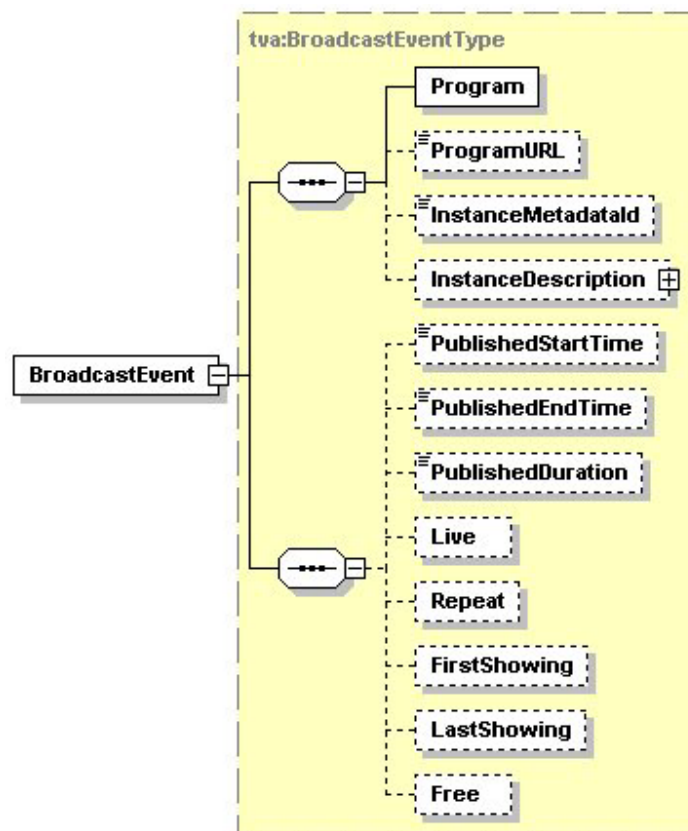


Figure 10: UML-like representation of a BroadcastEvent fragment

A TVAIDRefType value is used by the serviceIDRef attribute to identify the service on which the event described by this BroadcastEvent fragment will be broadcast. It points to the serviceId attribute, of a ServiceInformation element in the same metadata description, whose type is TVAIDType. This latter value shall be unique within the TVA metadata description. The guideline in clause 4.3.5 on the assignment and use of TVAIDType/TVAIDRefType values should be taken into account when dealing with these values.

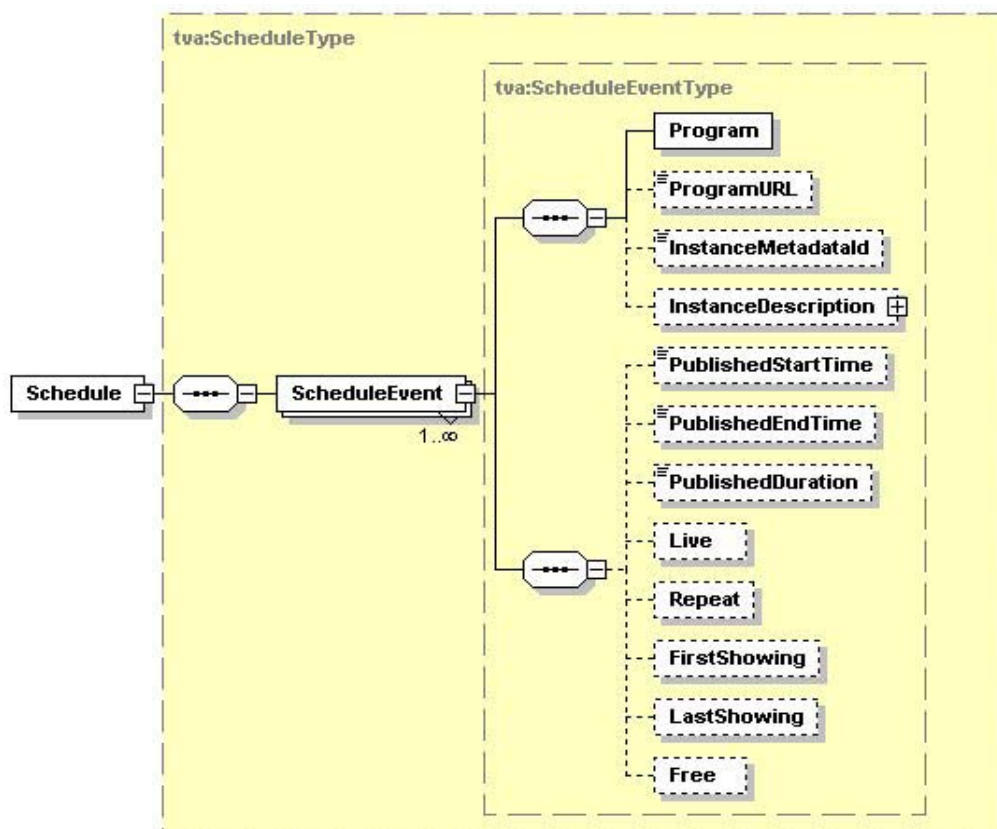
### 4.3.1.7 Schedule fragment

The Schedule type provides an alternative way to that of the BroadcastEvent type for describing events within a broadcast system. It provides a mechanism to group a number of consecutive events together, which span a given time period on a single service (TS 102 822-3-1 [9]).

The use of the ScheduleType for describing broadcast events has the following properties:

- To extract a single event from a schedule, the entire schedule must be decoded to locate the event of interest.
- The schedule type has been designed to aid large and collective updates. An update to a single event, within the schedule, will cause the entire Schedule fragment to be updated. However this is often not a problem, as subsequent events are often affected by a single event change.

As a result, the Schedule fragment instantiates the ScheduleType and is a child element of the ProgramLocationTable, which forms part of the TVAMain fragment.



**Figure 11: UML-like representation of a Schedule fragment**

A `TVAIDRefType` value is used by the `serviceIDRef` attribute to identify the service on which the events described by the schedule will be broadcast. It should be managed the same way as described in the previous clause for the `BroadcastEvent` fragment by taking into account the guidelines in clause 4.3.5 on the assignment and use of `TVAIDType`/`TVAIDRefType` values.

### 4.3.1.8 ServiceInformation fragment

A `ServiceInformation` fragment contains details about a single `Service` within a broadcast system (TS 102 822-3-1 [9]).

A `ServiceInformation` fragment is an instance of a `ServiceInformationType` and a child of the `ServiceInformationTable`, which is a member of the `TVAMain` fragment.

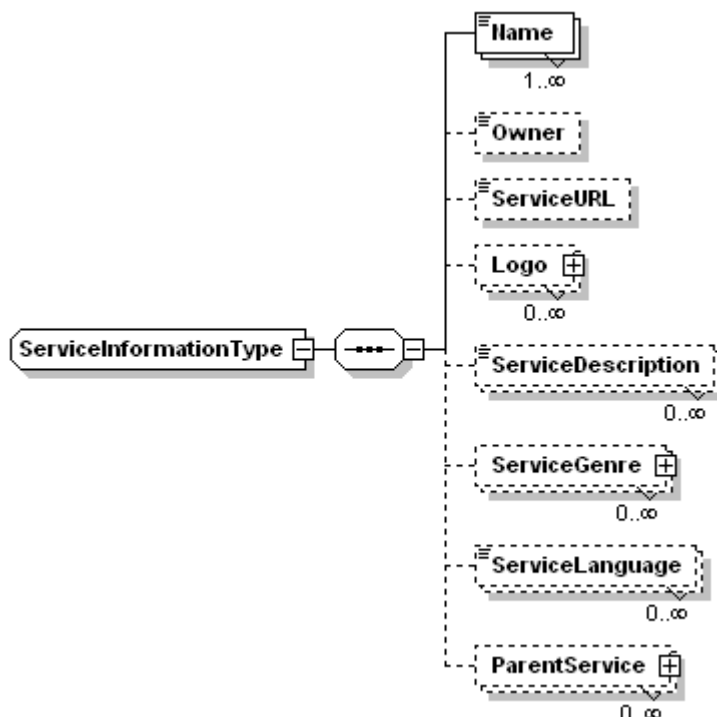


Figure 12: UML-like representation of a `ServiceInformation` fragment

### 4.3.1.9 CreditsInformation fragments

The `CreditsInformationTable` element instantiates the `CreditsInformationTableType` (TS 102 822-3-1 [9]).

It gathers together details about the people and organizations involved in the production of the different content items described in the metadata description. It is used to lighten the size of the description of each content item, by allowing the use of a pointer reference to a `PersonName` or `OrganizationName` element contained within the `CreditsInformationTable`. It may be useful, for example, where several content items share the same credit information, in which case the information only need to be instantiated once.

#### 4.3.1.9.1 PersonName fragment

A `PersonName` element instantiates a `PersonNameType` and shall form a single `TVA` fragment (TS 102 822-3-1 [9]). The `PersonName` element is a child element of the `CreditsInformationTable`, where the `CreditsInformationTable` forms a member of the `TVAMain` fragment.

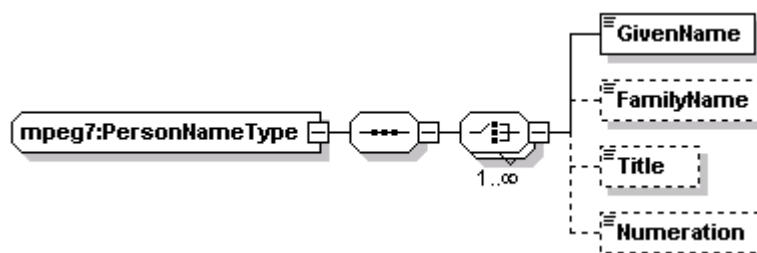


Figure 13: UML-like representation of `PersonNameType`

#### 4.3.1.9.2 OrganizationName fragment

The `OrganizationName` element shall form a single TVA fragment (TS 102 822-3-1 [9]).

The `OrganizationName` element is a child element of the `CreditsInformationTable`, where the `CreditsInformationTable` forms a member of the `TVAMain` fragment.

#### 4.3.1.10 PurchaseInformation fragment

The `PurchaseInformation` element shall form a single TVA fragment (TS 102 822-3-1 [9]).

The `PurchaseInformation` element is a child of the `PurchaseInformationTable`, which is a member of the `TVAMain` fragment.

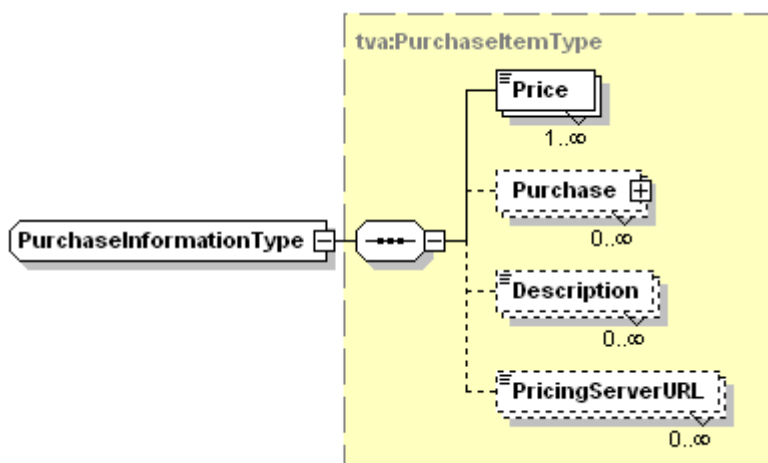


Figure 14: UML-like representation of a PurchaseInformation fragment

#### 4.3.1.11 Review fragment

A `Review` element instantiates the `MediaReviewType` and contains a single review for a content item identified by a CRID (TS 102 822-4 [12]).

The `Review` fragment is thus a child element of the `ProgramReviewTable`, which is a member of the `TVAMain` fragment (TS 102 822-3-1 [9]).

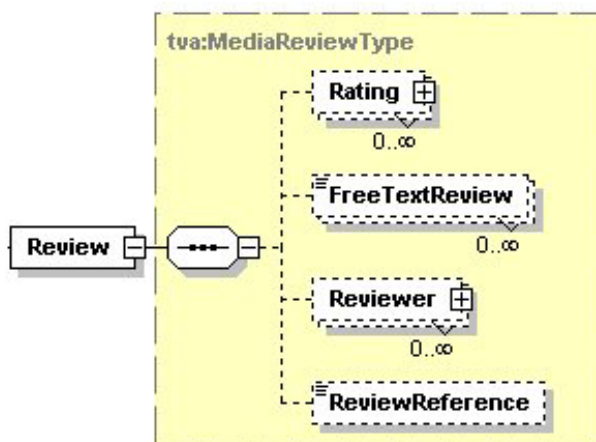


Figure 15: UML-like representation of a ProgramReviews fragment



#### 4.3.1.12 User Description information

The transmission of a `UserDescription` element is not seen to be of any practical use within a unidirectional environment and so no fragmentation structure has been defined.

#### 4.3.1.13 ClassificationScheme fragments

`ClassificationSchemes` elements are used to define a list of controlled terms used by a number of TVA data types e.g. Genre Classification, AV Coding types etc (TS 102 822-3-1 [9]). In addition `CSAlias` elements, used as aliases to the `ClassificationSchemes`, can be provided to reduce the size of the reference required to identify the Classification scheme used.

##### 4.3.1.13.1 CSAlias fragment

The `CSAlias` fragment instantiates the `ClassificationSchemeAliasType` defined by MPEG-7 and is a child of the `ClassificationSchemeTable`, which is a member of the `TVAMain` fragment (TS 102 822-3-1 [9]).

A single `CSAlias` element shall form a single `CSAlias` fragment.

##### 4.3.1.13.2 ClassificationScheme fragment

The `ClassificationScheme` element is an instance of the `ClassificationSchemeType`.

The `ClassificationScheme` fragment is thus a `ClassificationScheme` element, child of the `ClassificationSchemeTable`, which is a member of the `TVAMain` fragment (TS 102 822-3-1 [9]).

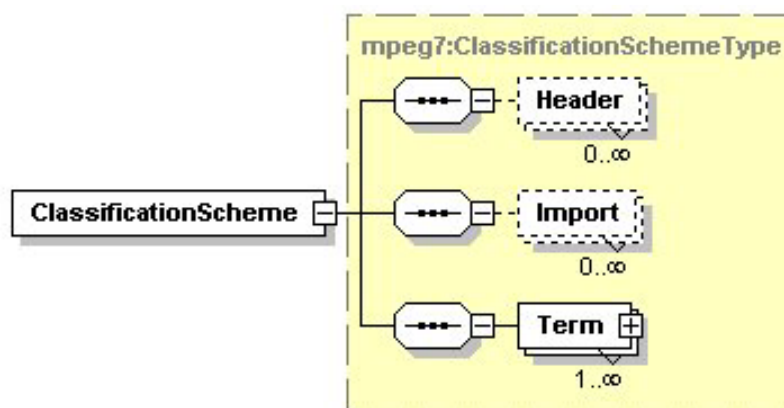


Figure 16: UML-like representation of a `ClassificationScheme` fragment

#### 4.3.1.14 Segmentation

Segmentation Information can be used to enhance the users viewing experience by providing the ability to view content in a non-linear way. Segmentation information is split up into two groups according to the two schema types they use, namely the `SegmentInformationType` and the `SegmentGroupInformationType`.

The `SegmentInformationType` provides details about the segment e.g. start offset, duration, description etc.

The `SegmentGroupInformationType` enables the Grouping of segments so for example to define the way in which content, should be navigated (TS 102 822-3-1 [9]).

The `SegmentGroupInformation` and `SegmentInformation` make extensive use of `TVAIDType`/`TVAIDRefType` and the guideline define in clause 4.3.5 should be followed.

#### 4.3.1.14.1 SegmentInformation fragment

A `SegmentInformation` element instantiates the `SegmentInformationType` and describes a single segment of a content item. It is seen as the smallest updateable unit and so shall form a single fragment.

A `SegmentInformation` fragment is thus a `SegmentInformation` element, which is a child of the `SegmentList` element, which forms part of the `TVAMain` fragment (TS 102 822-3-1 [9]).

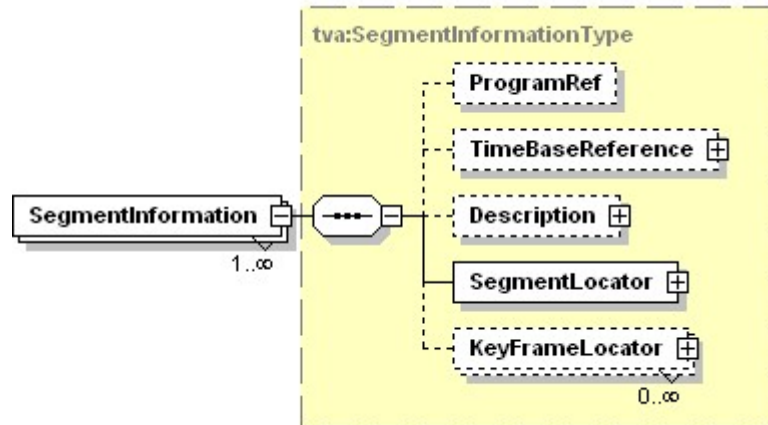


Figure 17: UML-like representation of a `SegmentInformation` fragment

#### 4.3.1.14.2 SegmentGroupInformation fragment

A `SegmentGroupInformation` element instantiates the `SegmentGroupInformationType`.

It allows the definition of a segment group, namely a set of references to `SegmentInformation` elements, which can be used to define a mode of navigation or a virtual piece of content e.g. Highlights of a football match. It is also possible for a segment group to reference another segment group providing hierarchical navigation of the content similar to a table of contents.

A `SegmentGroupInformation` fragment is thus a `SegmentGroupInformation` element, which is a child of the `SegmentGroupList` element, which forms part of the `TVAMain` fragment (TS 102 822-3-1 [9]).

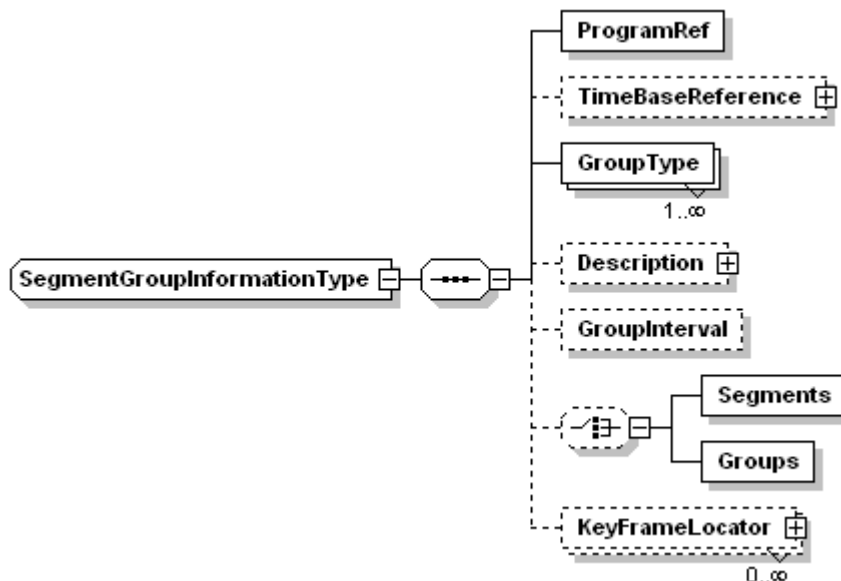


Figure 18: UML-like representation of a `SegmentGroup` fragment

#### 4.3.1.15 Package Fragment

The `Package` element (TS 102 822-3-3 [10]) shall form a single TVA fragment.

The `Package` element is a child of the `PackageTable`, which is a member of the `TVAMain` fragment.

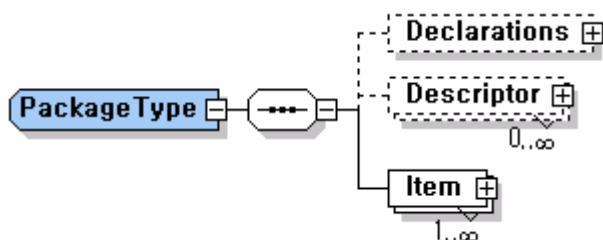


Figure 19: UML-like representation of a Package fragment

#### 4.3.1.16 Interstitial Campaign Fragment

The `InterstitialCampaign` element (TS 102 822-3-4 [11]) shall form a single TVA fragment.

The `InterstitialCampaign` element is a child of the `InterstitialCampaignTable`, which is a member of the `TVAMain` fragment.

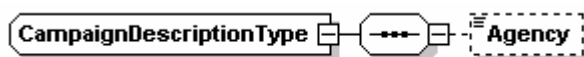


Figure 20: UML - like representation of an InterstitialCampaign fragment

#### 4.3.1.17 RMPIDescription Fragment

The `RMPIDescription` element (TS 102 822-5-1 [13]) shall form a single TVA fragment.

The `RMPIDescription` element is a child of the `RMPITable`, which is a member of the `TVAMain` fragment.

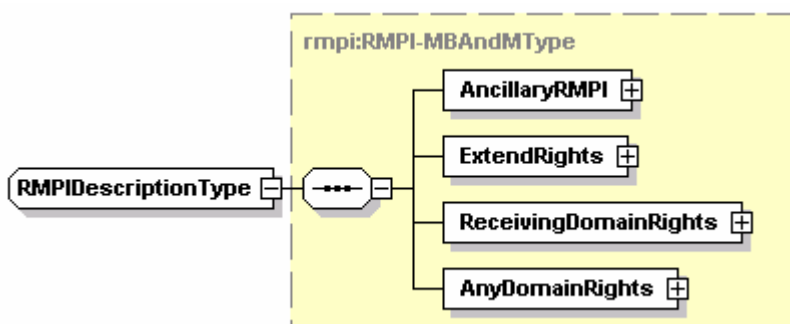


Figure 21: UML - like representation of an RMPIDescription fragment

#### 4.3.1.18 Coupon Description Fragment

The `CouponDescription` element (TS 102 822-3-3 [10]) shall form a single TVA fragment.

The `CouponDescription` element is a child of the `CouponTable`, which is a member of the `TVAMain` fragment.

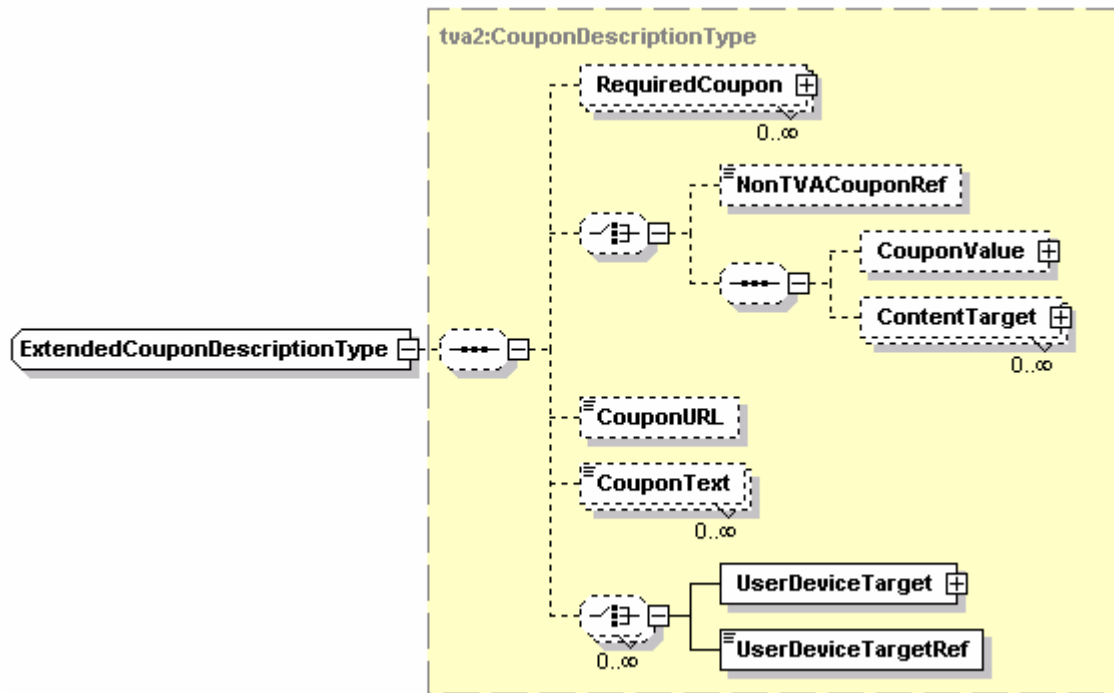


Figure 22: UML - like representation of a CouponDescription fragment

#### 4.3.1.19 TargetingInformation Fragment

The TargetingInformation element (TS 102 822-3-4 [11]) shall form a single TVA fragment.

The TargetingInformation element is a child of the TargetingInformationTable, which is a member of the TVAMain fragment.

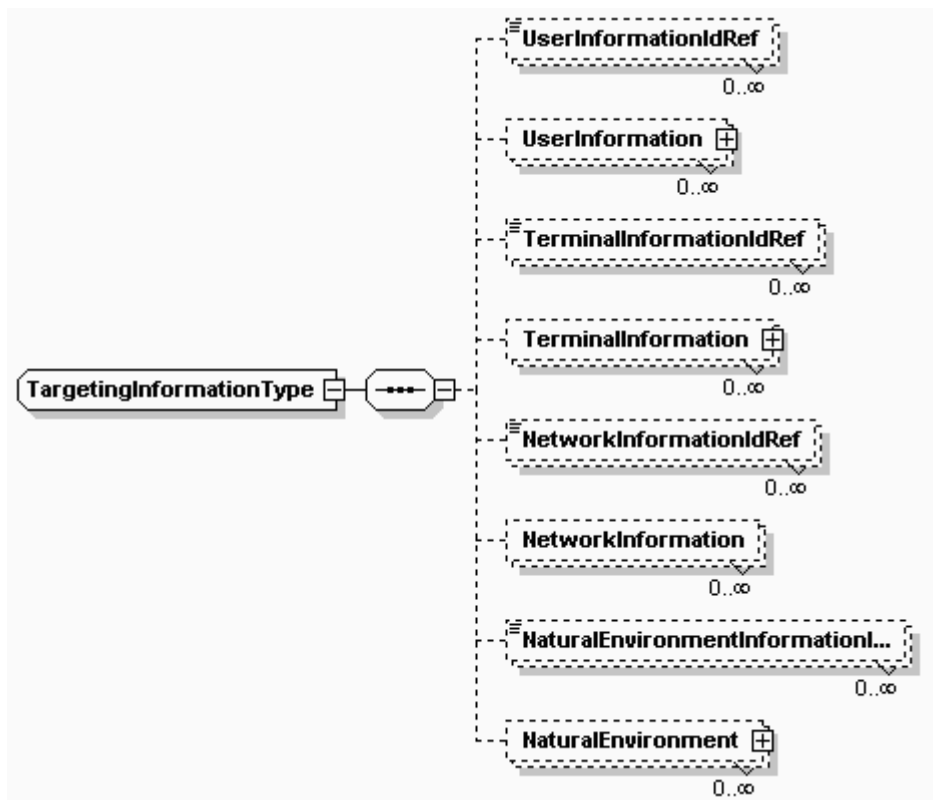


Figure 23: UML - like representation of a TargetingInformation fragment

#### 4.3.1.20 InterstitialBreak Fragment

The `InterstitialBreak` element (TS 102 822-3-4 [11]) shall form a single TVA fragment.

The `InterstitialBreak` element is a child of the `InterstitialTable`, which is a member of the `TVAMain` fragment.

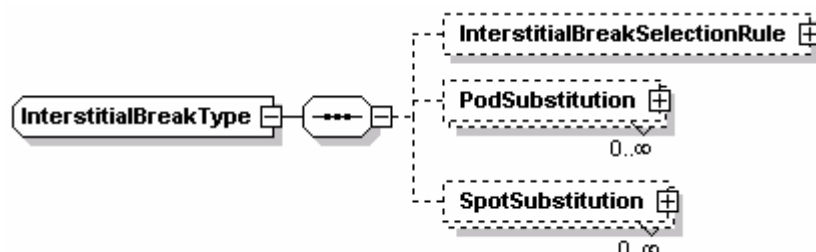


Figure 24: UML - like representation of an `InterstitialBreak` fragment

#### 4.3.1.21 Rule Fragment

The `Rule` element (TS 102 822-3-4 [11]) shall form a single TVA fragment.

The `Rule` element is a child of the `RulesTable`, which is a member of the `TVAMain` fragment.

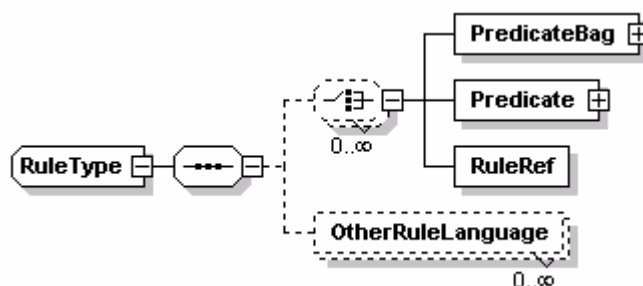


Figure 25: UML - like representation of a `Rule` fragment

#### 4.3.1.22 Recording Cache Fragments

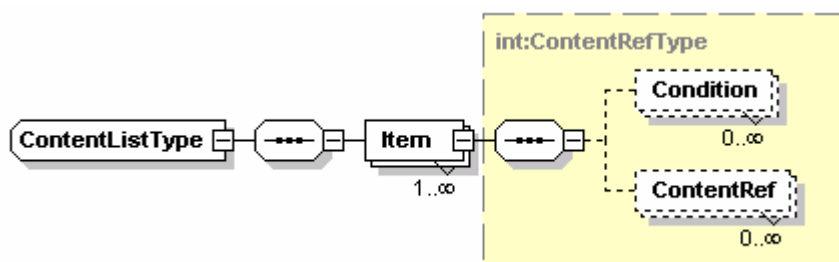


Figure 26: UML - like representation of the `ContentListType` used by the following fragments

##### 4.3.1.22.1 Request Fragment

The `Request` element (TS 102 822-3-4 [11]) shall form a single TVA fragment.

The `Request` element is a child of the `RecordingCacheTable`, which is a member of the `TVAMain` fragment.

#### 4.3.1.22.2 Replace Fragment

The `Replace` element (TS 102 822-3-4 [11]) shall form a single TVA fragment.

The `Replace` element is a child of the `RecordingCacheTable`, which is a member of the `TVAMain` fragment.

#### 4.3.1.22.3 Expire Fragment

The `Expire` element (TS 102 822-3-4 [11]) shall form a single TVA fragment.

The `Expire` element is a child of the `RecordingCacheTable`, which is a member of the `TVAMain` fragment.

### 4.3.2 Fragment Identification and Versioning

Many element types within the TVA schema have both a `fragmentVersion` and a `fragmentId` attributes, it is recommended not to use these attribute to identify a fragment or to indicate fragment versions in a unidirectional environment. This is because it is required to partially decode the fragment to extract the required information.

A means of assigning an identifier to a fragment and a version to a fragment at the appropriate level is described in clause 4.6.

### 4.3.3 Element ordering

When two or more elements of the same type occur within a metadata description, it is assumed that their order within the parent element is unimportant e.g. `ProgramInformation` elements within the `ProgramInformationTable`. Therefore, it is not a requirement to maintain the order of elements across the delivery system.

Where a parent element can contain more than one element type e.g. `PersonName` and `OrganizationName` within the `CreditsInformationTable` - it is the responsibility of the receiving terminal to ensure that the order of element types is maintained, with respect to the TVA schema.

### 4.3.4 TVA access unit

The TVA access unit is defined as being a container that holds one or more TVA fragments. A TVA fragment shall be wholly contained within a single container. TVA fragments within a TVA access unit can be of any type, provided that an individual's fragment type can be identified without fragment decoding.

Where BiM is used as the encoding solution a TVA access unit shall take the form of an MPEG-7 Access Unit (AU), with the TVA fragments type being identified, using the `ContextPath` within BiM.

It is the responsibility of the broadcaster to manage the number of TVA fragments within a TVA Access Unit, bearing the following in mind:

- The overhead of transmitting a container - The smaller the container the less efficient it will be in terms of bandwidth usage.
- The updating of a single TVA fragment within a container will cause the container to be updated.

### 4.3.5 Use of `TVAIDType`, `TVAIDRefType` and `TVAIDRefsType`

In a number of instances within the TVA metadata schema, elements make use of the `TVAIDType`, `TVAIDRefType` and `TVAIDRefsType` data types.

These replace the respective standard XML ID, IDREF and IDREFS data types. They are provided to enable elements to reference other elements within the same TVA metadata description. However as a TVA metadata description is split up into a number of TVA fragments and transmitted as a TVA metadata fragment stream, the following issues should be noted:

- The synchronization of TVAIDType, TVAIDRefType and TVAIDRefsType values between fragment versions.
- The possibility of changing references when dealing with partial metadata descriptions.

A TVA metadata fragment stream is treated as if it were a single XML instance document. The value assigned to an attribute instantiating a TVAIDType shall be unique within a single TVA metadata description. Also, all attributes instantiating the TVAIDRefType or the TVAIDRefsType must make references to elements containing an attribute of type TVAIDType, only within the same TVA metadata description. No provision is currently made for referencing elements across multiple TVA metadata descriptions.

The guideline rules for use of TVAIDType, TVAIDRefType and TVAIDRefsType are therefore:

- The value assigned to the TVAIDType element must be unique within a TVA metadata description.
- In case of an update to an element with an attribute of type TVAIDType, it may be appropriate to create a totally new element with a new TVAIDType value when this element changes significantly e.g. in the Credits list when an actor dies. This is to ensure that the original description can still be referenced from within the metadata description, rather than an updated one.
- Where there is a choice between an inline instantiation and the use of a reference, the broadcaster should weigh up the added complexity of using a TVAIDType/TVAIDRefType (additional level of indirection) against the potential bandwidth savings.
- It is important when thinking about the delivery dynamics to ensure that an element being referenced is available within the TVA metadata description, before or at the same time as the element, making the reference. The reverse is also true; when deleting elements, it is important to remove the element that makes the reference, before the element being referenced.

### 4.3.6 Use of ID, IDRef, XPath and xml:lang

As explained in clause 4.3.5, the use of ID and IDRef should be avoided under the following circumstances.

Some datatypes within the MPEG7 stub schema make use of these built-in XML schemas datatypes. Unless the scope of these types is restricted to a single fragment, they should not be used. The reason is that it is not possible to maintain referential integrity across two or more fragments.

Some datatypes within the MPEG7 stub schema make use of XPath expressions. For similar reasons, the scope of XPath resolution shall be restricted to a single fragment. The use of XPath resolution across two or more fragments should also be avoided because canonical form of the XML source document is not preserved.

Some datatypes within the MPEG7 stub schema make use of xml:lang attributes. The scope of this attribute should be restricted to a single fragment.

## 4.4 Fragment encoding

### 4.4.1 TVA-init message

#### 4.4.1.1 Overview

The TVA-init message specified in the present clause is used to configure parameters required for the decoding of the TVA metadata fragment stream. There shall only be one TVA-init associated with a TVA metadata fragment stream.

A delivery layer suitable for conveying a TVA metadata fragment streams, shall provide a means of delivering the TVA-init message to the terminal before any fragment decoding occurs.

Syntax	No. of bits	Mnemonic
TVA-init {		
EncodingVersion	8	uimsbf
IndexingFlag	1	bslbf
reserved	7	
DecoderInitptr	8	bslbf
if(EncodingVersion == '0x01') {		
BufferSizeFlag	1	bslbf
PositionCodeFlag	1	bslbf
reserved	6	
CharacterEncoding	8	uimsbf
if (BufferSizeFlag=='1') {		
BufferSize	24	uimsbf
}		
}		
if(!IndexingFlag) {		
IndexingVersion	8	uimsbf
}		
reserved	0 or 8+	
DecoderInit( )		bslbf
}		

**EncodingVersion:** This field indicates the method of encoding used to represent the TVA metadata fragments. Table 1 provides the possible set of values for this field.

**Table 1: Table of values for the EncodingVersion parameter**

Value	Encoding Version
0x00	Reserved
0x01	TVA MPEG_7 profile (BiM) ISO/IEC 15938-1 [4]
0x02 - 0x0F	TVA reserved
0x10 - 0xFF	User defined

**IndexingFlag:** Indicates if one or more indexes are carried within the stream.

**DecoderInitptr:** This field conveys a pointer, which defines the offset in bytes from the start of the TVA-init message where the DecoderInit can be found.

**BufferSizeFlag:** Indicates if a BufferSize for the Zlib coded is defined. If not defined the decoder shall assume a maximum of 1 000 bytes.

**PositionCodeFlag:** This flag indicates if the BiM contextPath Position Code is used in the encoded fragment. When set to "0" the Position Code within the contextPath shall be ignored. In that case, it must be noted that the canonical format of the instance description is not preserved, i.e. the order of the elements within the rebuilt description is not preserved.

**CharacterEncoding:** This field conveys the character encoding scheme for all textual data used within the TVA metadata fragment stream. Table 2 defines the set of possible values for this field.

**Table 2: Character encoding and their termination values**

Value	Description	Termination Value
0x00	7 bit ASCII (ISO/IEC 10646 [1])	0x00
0x01	UTF-8	0x00
0x02	UTF-16	0x0000
0x03	GB2312	0x00
0x04	EUC-KS	0x0000
0x05	EUC-JP	0x0000
0x06	Shift_JIS	0x0000
0x07 - 0xE0	TVA reserved	Undefined
0xE1 - 0xFF	User defined	Undefined



**BufferSize:** This element conveys the maximum number of bytes a Zlib buffer will decompress to.

**IndexingVersion:** This element indicates the method used to represent TVA indices. It provides the possible set of values for this element.

**Table 3: Table of values for the IndexingVersion field**

Value	Indexing Version
0x00	reserved
0x01	TVA Index Version 1
0x02 - 0xEF	TVA reserved
0xF0 - 0xFF	User defined

**reserved:** Variable data space for inserting future initialization parameters. In this version, the length of this field shall be 0x00.

**DecoderInit:** This element conveys the `DecoderInit`. The format of the `DecoderInit` is dependant on the encoding method used. In the case of BiM (`EncodingVersion` = 0x01) it shall be as defined in clause 4.4.2.1.

## 4.4.2 MPEG-7 system profile

Due to the characteristics of a uni-directional environment, a number of restrictions have been imposed on how the MPEG-7 BiM profile shall be used. The present clause details these restrictions.

### 4.4.2.1 DecoderInit

The `DecoderInit` is used to configure parameters required for the decoding of the binary fragments. There is only one `DecoderInit` associated with one `TVAmetadata` fragment stream. The `DecoderInit` shall take the form as specified in ISO/IEC 15938-1 [4] with the following caveats.

#### 4.4.2.1.1 UnitSizeCode

**ISO Semantics:** This is a coded representation of `UnitSize`. `UnitSize` is used for the decoding of the binary fragment update payload.

**Restriction:** This `UnitSizeCode` variable shall be set to the default value: "000".

#### 4.4.2.1.2 InitialDescription

**ISO Semantics:** This conveys portions of a description using the same syntax and semantics as an MPEG7 access unit. The `InitialDescription` provides an initial state for the binary description tree.

**Restriction:** The TVA fragment containing the `TVAMain` element and is the entry point of the TVA metadata fragment stream. The `InitialDescription` may be carried in two ways:

- Along with the `DecoderInit`.
- Independently from the `DecoderInit`.

In the case where the `InitialDescription` is sent independently from the `DecoderInit`, it must be received and decoded by the receiving terminal before processing of any other TVA fragments. The delivery layer shall provide signalling to indicate where the `InitialDescription` is to be found.

#### 4.4.2.2 FragmentUpdateCommand

**ISO Semantics:** The `FragmentUpdateCommand` code word specifies the command that shall be executed on the binary format description tree. It should be ignored when the `PositionCodeFlag` in the `TVA-Init` is set to "0".

**TVA Semantics:**

In the TVA framework, the semantics of the `FragmentUpdateCommand` is refined such as:

- When a fragment is modified (i.e. updated or enriched), the replace command (`ReplaceContent`) is used.
- When a fragment is no longer valid (the programme has been dropped), the delete command (`DeleteContent`) is used.
- When a fragment is obsolete, the fragment is no longer transmitted.

##### 4.4.2.2.1 Guidelines for the use of the FragmentUpdateUnit

The `FragmentUpdateUnit` uses the `ContextPath` along with a `Position Code` to determine where the TVA fragment being updated should be placed relative to its parent element. The `ContextPath` provides the absolute path from the metadata description root (`TVAMain`) to the element, of which this fragment is a child. In a classical BiM implementation the `Position Code` is used to indicate the position of the TVA fragment among its sibling fragments, so as to maintain the original document order of sibling elements. In a TVA implementation of BiM it is not a requirement to maintain the original metadata description sibling ordering. It is not a requirement to support the use of the `Position Code` within a TVA implementation. However if it is used, the `Position Code` provides a handle to a previously transmitted TVA fragment.

###### 4.4.2.2.1.1 Position Code allocation

As described above, the `Position Code` provides a handle to a transmitted TVA fragment. This `Position Code` must be unique for all children of a given parent element. Due to the nature of a TVA metadata fragment stream i.e. a living description, constantly changing - over time the `Position Code` value will become very large. Therefore it would be advantageous to reuse previously allocated values that are no longer valid. However the following should be taken into account:

- A sufficient period of time should have elapsed since the use of a specific `Position Code` value to ensure that receiving terminals have automatically deleted from the cache the fragment previously assigned this `Position Code`.

###### 4.4.2.2.1.2 Fragment Add/Update

When a new or updated fragment is transmitted the `FragmentUpdateUnit` will have the following settings:

- `FragmentUpdateCommand` - Set to "0010" (`ReplaceContent`).
- `FragmentUpdateContext`.
- `ContextModeCode` - Set to "001" (Absolute path to fragments parent element, with no multiple payload support).
- `ContextPath Position Code` - A value that identifies the fragment within a TVA metadata description and shall be unique if used.

When the receiving terminal acquires the `FragmentUpdateUnit` it will use the `ContextPath` along with the `Position Code` to see if there is a TVA fragment already cached with the same `Position Code`. If a previous TVA fragment is found the previous TVA fragment shall be deleted and replaced with the new TVA fragment. If a previous TVA fragment is not found the TVA fragment is just cached along with the rest.

#### 4.4.2.2.1.3 Fragment Delete

When a fragment is no longer valid a Delete command can be sent to the receiving terminal, to inform the device that the fragment shall no longer be used. A Delete command will have the following settings:

- `FragmentUpdateCommand` - Set to "0011" (`DeleteContent`).
- `FragmentUpdateContext`.
- `ContextModeCode` - Set to "001" (Absolute path to fragments parent element, with no multiple payload support).
- `ContextPath Position Code` - The original unique value assigned to this TVA metadata description fragment.

When the receiving terminal acquires the `FragmentUpdateUnit` it will use the `ContextPath` along with the `Position Code` to see if the TVA fragment is within its cache. If the TVA fragment is found it shall no longer be used and removed from the terminal's cache.

It is assumed that the delete command will only be transmitted for a relatively short period of time, to enable currently listening terminals to keep their cache updated.

However due to the nature in which TVA fragments are acquired within a unidirectional environment, the Delete command should not be relied upon to remove elements from the terminals cache. The terminal should implement a timeout mechanism, where if a given fragment is not seen in the data stream for a specified period of time, the terminal shall assume that the fragment is no longer valid and remove it from its cache.

#### 4.4.2.2.1.4 Example decoder behaviour

A TVA metadata fragment stream generator has an internal representation of the decoder memory. This representation can be modelled by a buffer, which contains an infinite number of slots.

A *TV-Anytime* decoder contains several tables (`ProgramInformationTable`, etc.). These tables correspond to the fragmentation specification of a TVA metadata description.

If you consider the following scenario describing the management of the `programInformation` description, a fragment unit is composed of:

- a command (`ReplaceContent`, `DeleteContent`);
- a path, which gives, element and type information (provides the type of the payload);
- a `Position Code` (gives the position of the element (child number) encoded in the payload);
- a payload (the encoded TVA fragment).

At time  $t$ ,

Stream content (each fragment unit being carouseled)	Decoder memory																								
fragment unit: (ReplaceContent, a ProgramInformation / position 2, Payload122)	<table border="1"> <tr><td>1</td><td>Empty</td></tr> <tr><td>2</td><td>Payload122</td></tr> <tr><td>3</td><td>Empty</td></tr> <tr><td>4</td><td>Empty</td></tr> <tr><td>5</td><td>Empty</td></tr> <tr><td>6</td><td>Payload12</td></tr> <tr><td>7</td><td>Empty</td></tr> <tr><td>8</td><td>Empty</td></tr> <tr><td>9</td><td>Payload56</td></tr> <tr><td>10</td><td>Empty</td></tr> <tr><td>11</td><td>Empty</td></tr> <tr><td>...</td><td></td></tr> </table>	1	Empty	2	Payload122	3	Empty	4	Empty	5	Empty	6	Payload12	7	Empty	8	Empty	9	Payload56	10	Empty	11	Empty	...	
1	Empty																								
2	Payload122																								
3	Empty																								
4	Empty																								
5	Empty																								
6	Payload12																								
7	Empty																								
8	Empty																								
9	Payload56																								
10	Empty																								
11	Empty																								
...																									
fragment unit: (ReplaceContent, a ProgramInformation / position 6, Payload12)																									
fragment unit: (ReplaceContent, a ProgramInformation / position 9, Payload56)																									

At time  $t'$

Stream content (each fragment unit being carouseled)	Decoder memory																								
fragment unit: (ReplaceContent, a ProgramInformation / position 2, Payload122)	<table border="1"> <tr><td>1</td><td>Empty</td></tr> <tr><td>2</td><td>Payload122</td></tr> <tr><td>3</td><td>Empty</td></tr> <tr><td>4</td><td>Empty</td></tr> <tr><td>5</td><td>Empty</td></tr> <tr><td>6</td><td>Payload12 false</td></tr> <tr><td>7</td><td>Empty</td></tr> <tr><td>8</td><td>Payload6</td></tr> <tr><td>9</td><td>Empty</td></tr> <tr><td>10</td><td>Empty</td></tr> <tr><td>11</td><td>Empty</td></tr> <tr><td>...</td><td></td></tr> </table>	1	Empty	2	Payload122	3	Empty	4	Empty	5	Empty	6	Payload12 false	7	Empty	8	Payload6	9	Empty	10	Empty	11	Empty	...	
1	Empty																								
2	Payload122																								
3	Empty																								
4	Empty																								
5	Empty																								
6	Payload12 false																								
7	Empty																								
8	Payload6																								
9	Empty																								
10	Empty																								
11	Empty																								
...																									
fragment unit: (DeleteContent, a ProgramInformation / position 6)																									
fragment unit: (ReplaceContent, a ProgramInformation / position 8, Payload6)																									

Therefore, a TVA decoder can notify an application that payload 12 is not longer valid. If the application relies on payload 12, it will be notified about the deletion. But if the TVA decoder has not received the first fragment unit, which sets the value of the payload 12 at position 6, the TVA decoder ignores this "delete" fragment.

In addition payload 56 (position 9) is now obsolete and is no longer transmitted.

Therefore, a TVA metadata fragment stream generator manages the TVA metadata fragment stream in order to send these notifications to the decoder.

#### 4.4.2.3 ContextMode

**ISO Semantics:** The `ContextMode` specifies the addressing mode for the context path.

**Restriction:** The ContextMode code is limited to the value "001" (Navigate in "Absolute addressing mode" from the selector node to the node specified by the ContextPath).

#### 4.4.2.4 TV-Anytime codec

##### 4.4.2.4.1 Classification scheme wrapper

In the MPEG-7 framework, the use of a specific codec for a specific type is signalled using the codec configuration mechanism defined in ISO/IEC 15938-1 [4]. This mechanism associates a codec using its URI with a list of schema types. For that purpose, a URI is assigned to each codec in a classificationScheme, which defines the list of the specific codecs.

In the present document, this list is composed of 3 specific codecs: Zlib, dateTime and date. The following figure gives the standard classificationScheme as used by the TVA MPEG-7 profile.

```
<ClassificationScheme uri="urn:tva:metadata:cs:CodecTypeCS:2004">
  <Term termID="1">
    <Name xml:lang="en">ZlibCodec</Name>
    <Definition xml:lang="en">Encodes using Zlib</Definition>
  </Term>
  <Term termID="2">
    <Name xml:lang="en">tvadateTimeCodec</Name>
    <Definition xml:lang="en">Encodes date using Modified Julian
    Date & Time in Millisecond</Definition>
  </Term>
  <Term termID="3">
    <Name xml:lang="en">tvadateCodec</Name>
    <Definition xml:lang="en">Encodes date using Modified Julian
    Date</Definition>
  </Term>
</ClassificationScheme>
```

##### 4.4.2.4.2 dateTime Codec

The XML Schema primitive simple type dateTime is used widely within the TVA metadata Schema and so a specific codec has been designed for representing date time fields.

Times shall be based on GMT, with no provision provided for maintaining the local time offset information. Any requirement to localize time values shall be performed by the receiving terminal.

The following describes how the XML Schema primitive dateTime shall be encoded.

##### 4.4.2.4.2.1 Encoding

The dateTime primitive is represented as an 8-byte unsigned integer number (Big-Endian), Days are represented using the first 4 bytes using Modified Julian Date. Time is represented using the last 4 bytes expressed as the number of elapsed milliseconds since 00:00:00 hours.

The origin for the Modified Julian Date shall be Midnight on 17<sup>th</sup> November 1858.

Example dates:

Date	Modified Julian Date
1 <sup>st</sup> April 1980	44 330
30 <sup>th</sup> January 2000	51 573
1 <sup>st</sup> March 2001	51 969

Example dateTimes:

dateTime value	Encoded value
1980-04-01T02:00:00Z	0x0000AD2A006DDD00
2000-01-30T12:10:01Z	0x0000C975029C59A8
2001-03-01T00:00:00Z	0x0000CB0100000000

#### 4.4.2.4.3 date codec

The XML Schema primitive simple type date describes a date within the Gregorian calendar. The date takes the form of a string as defined by ISO/IEC 8601 [21].

##### 4.4.2.4.3.1 Encoding

The XML Schema date primitive shall be represented as a 4-byte unsigned integer (Big-Endian). It shall contain the number of days using the Modified Julian Date format, as described in clause 4.4.2.4.2.1.

#### 4.4.2.4.4 Zlib optimized decoder

In the TVA MPEG-7 profile, the following *Zlib codec* is used by default for the encoding of strings instead of the UTF-8 representation. This Zlib codec is reinitialized for each TVA fragment.

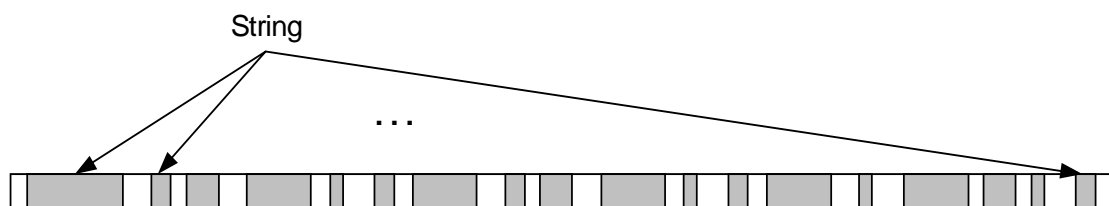
##### 4.4.2.4.4.1 Rationale

Classical lossless statistical compression algorithms (like Zip or GZip) are used in the present document to improve character strings compression. The present document uses the Zlib library [6].

In most cases, when strings are short (fewer than 100 characters), the performance of Zlib is poor. Indeed, this statistical compression algorithm requires a larger look ahead buffer to start eliminating redundancy. To achieve a good compression ratio, the proposed "Zlib optimized decoder" gathers different strings into one buffer before compressing it. The size of this buffer, noted *buffer\_size*, allows the encoder to balance the compression ratio and the memory needed at the decoder. The codec has to manage an input buffer of strings as described in the following clauses. The default value of the buffer size is set to 1 000 bytes and can be overridden in the TVA-init Message.

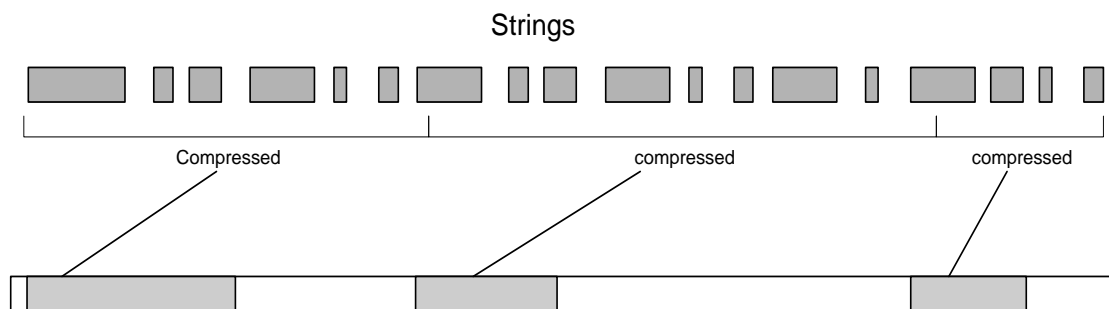
##### 4.4.2.4.4.2 Encoding

At the encoding phase, the buffer is fed with strings. When the buffer is full, it is compressed and the resulting compressed chunk of data is placed in the expected position of the first string compressed using this buffer. Figure 27 represents a BiM binary stream without the "Zlib optimized decoder". Strings (in grey) are dispersed along the entire bitstream.



**Figure 27: BiM bitstream without Zlib optimized decoder**

Figure 28 represents only the strings to be compressed and shows how they are gathered and compressed together to create a more compact bitstream and where the resulting compressed chunks are dispersed. The location of the compressed chunk ensures that during the decoding process when a string is required either the Zlib codec gets a compressed chunk, decompress it and delivers the string from the decompressed buffer or if its decompressed buffer is not empty, it delivers the string from its decompressed buffer.



**Figure 28: BiM stream with Zlib optimized decoder**

The decoding process is formally explained in clause 4.4.2.4.4.3.

#### 4.4.2.4.4.3 Decoding

At the decoding level, the compressed chunk of strings is decompressed into a buffer (still limited by the *buffer\_size* used at the encoder side). This buffer delivers the strings to the leaves (using the separator character). In the case where a value is encoded over two or more chunks, the value shall be the concatenation of all characters extracted from the buffers until the separator character.

#### 4.4.2.4.4.4 Behaviour

Syntax	No. of Bits	Mnemonic
ZlibDecoder() {		
ResultString=""		
TempChar = GiveNextCharInBuffer();		
While (TempChar != separatorChar) {		
ResultString = concat(ResultString, Tempchar)		
Tempchar = GiveNextCharInBuffer()		
}		
return ResultString		
}		

Syntax	No. of Bits	Mnemonic
GiveNextCharInBuffer() {		
If isEmpty (charsBuffer) {		
ZlibStringLength	8+	vluimsbf8
ZlibString	8* ZlibStringLength	bslbf
CharsBuffer = ZlibDecompress(ZlibString)		
}		
return nextChar(charsBuffer)		
}		

#### 4.4.2.4.4.5 Semantics

In order to obtain the next string, the decoder reads the `charsBuffer` until it gets a `separatorChar`. If the `charsBuffer` is totally consumed before reaching a `separatorChar`, the `charsBuffer` is refilled by decompressing the next chunk available in the bitstream.

**ResultString:** A local variable representing the string expected.

**TempChar:** A local variable representing the character read in the `charsBuffer`.

**separatorChar:** A constant representing the `separatorChar` as defined by the `termination_value` of the related character encoding format specified in TVA-Init.

**ZlibStringLength:** Indicates the size in bytes of the compressed ZlibString. A value of zero is forbidden.

**ZlibString:** A representation of a compressed sequence of characters. The compression algorithm used is Zlib [6].

**ZlibDecompress(aString):** This decompresses a string and returns the decompressed string.

**NextChar(charsBuffer):** This function returns the first character of the `charsBuffer` and removes it from the `charsBuffer`.

**isEmpty (charsBuffer):** This function returns true if the `charsBuffer` is empty, otherwise false.

**Concat(aString, aCharacter):** This function returns the concatenation of a `String` and a `Character`.

**CharsBuffer:** A buffer of characters. It contains a list of string separated by a `separatorChar`.

## 4.5 Carriage of *TV-Anytime* data

*TV-Anytime* does not define the way in which the metadata should be carried within a specific delivery system. However *TV-Anytime* has defined a generic mechanism called a "Container" for grouping a number of related data structures together for transmission.

The following clauses describe the format of these containers and additional requirements on the delivery layer, to enable the identification of containers, the type of data a container carries and the current version of a container.

### 4.5.1 Containers

A container forms the top-level storage unit in which all TVA data within a unidirectional environment is transmitted. A container contains one or more related structures, which can be used to convey both data fragments and indexing information.

#### 4.5.1.1 Carriage of containers

*TV-Anytime* does not define the way in which these containers should be carried, as this is specific to the delivery system. However consideration has been given, to enable the container to be easily mapped on to standard delivery methods. For example in an MPEG-2 environment, the containers may be conveyed using sections, objects within a DSM-CC U-U Object Carousel or modules within a DSM-CC Data Carousel.

#### 4.5.1.2 Classification of containers

Containers are classified depending on what type of information they carry. The type of container shall be signalled with the delivery layer to enable a receiver to efficiently acquire containers carrying a specific type of data. This for example would enable a receiver to just acquire data containers, which it could use to populate a terminals local Database cache.

Containers are classified as follows:

- **Data Containers** - Containers carrying TVA metadata fragments and hold the following structures - `encapsulation`, `data_repository` (Binary data).
- **Index Containers** - Containers carrying Indexing information and holding the following structures - `index_list`, `index`, `multi_field_sub_index`, `data_repository`, `fragment_locators` (optional).

It should be noted that it is possible for a single container to carry both types of data.

The use of data containers as specified in the present document for the carriage of TVA fragments is mandatory.

#### 4.5.1.3 Container identification

To be able to identify a container, a container must be given a unique identifier (container ID). This shall take the form of a 16 bit value and shall be conveyed at the transport delivery layer. No provision is made to insert this identifier at the container level, since it should not be necessary to acquire the container to find out its container ID. The signalling of the container ID is out of scope for TVA and shall be stipulated by other appropriate standards bodies e.g. DVB, ATSC and ARIB.



## 4.5.2 Container versioning

Each container must have an associated version identifier. No provision is made within the container for this, as it should not be a requirement to load the container to see if the version has changed. Therefore version identification must be carried at the transport delivery layer. The definition of this version identifier is out of scope for TVA and shall be stipulated by other appropriate standards bodies e.g. DVB, ATSC and ARIB.

The container version is not used or referenced anywhere within a container or structure and so may take any form appropriate for the delivery mechanism.

### 4.5.2.1 Container syntax

Syntax	No. of Bits	Mnemonic
container () {		
container_header {		
num_structures	8	uimsbf
for (j=0; j<num_structures; j++) {		
structure_type	8	uimsbf
structure_id	8	uimsbf
structure_ptr	24	uimsbf
structure_length	24	uimsbf
}		
}		
for (j=0; j<byte_count; j++) {		
structure_body		
}		
}		

**num\_structures:** An 8 bit field specifying the number of structures contained within this container. A value of 0x00 is invalid.

**structure\_type:** An 8 bit field identifying the type of structure being referenced, according to table 4.

**Table 4: Structure\_type assignments**

Value	Description
0x00	Reserved
0x01	Encapsulation (see clause 4.6.1.1)
0x02	Data Repository (see clause 4.6.1.4)
0x03	Index List (see clause 4.8.5.3)
0x04	Index (see clause 4.8.5.4)
0x05	Multi Field Sub Index (see clause 4.8.5.5)
0x06	Fragment Locators (see clause 4.8.5.6)
0x07	moved_fragments (see clause 4.6.1.2)
0x08 - 0xDF	TVA Reserved
0xE0 - 0xFF	User defined

**structure\_id:** An 8 bit value which is used to distinguish between multiple occurrences of a specific `structure_type`. In some cases this is just an instance identifier (e.g. `index` and `multi_field_sub_index` structures) and in other cases it is used to distinguish the type of data carried within the structure (e.g. `data repository`).

In cases where the `structure_id` is not used the field shall be set to 0xff.

**Table 5: Structure\_type and their matching valid structure\_id**

Structure_type	structure_id	Description
0x00	0x00 - 0xFF	Reserved
0x01	0x00 - 0xFF	Reserved
0x02	0x00	Data Repository of type strings, see clause 4.8.4.1
0x02	0x01	Data repository of type binary data, see clause 4.6.1.4.1
0x02	0x02 - 0xFF	Reserved
0x03	0x00 - 0xFF	Reserved
0x04	0x00 - 0xFE	Used to identify a specify instance of an index structure, within a container
0x05	0x00 - 0xFE	Used to identify a specific instance of a multi_field_sub_index structure within a container
0x06	0x00 - 0xFE	Used to identify a specific instance of a fragment_locator structure
0x07 - 0xFF	0x00 - 0xFF	Reserved

**structure\_ptr:** A 24 bit field giving the offset in bytes from the start of this container to the first byte of the identified structure.

**structure\_length:** A 24 bit field which indicates the length in bytes of the structure pointed to by `structure_ptr`.

Entries within the `container_header` shall be ordered in ascending `structure_type` and `structure_id`. For example all structures of type `data_repository` shall be grouped together and items within the group ordered in ascending `structure_id`. This enables a device to efficiently locate a particular structure of interest.

**structure\_body:** Data forming one or more structures within this container.

#### 4.5.2.2 Container map

As has been mentioned above, containers do not include information such as, container Id, version or container category. This information must be carried at the delivery layer, informing a device when containers have changed and what the appropriate download parameters are. This higher-level messaging is termed a container map.

The concept of a container map can be found in a number of Broadcast delivery mechanisms, for instance the DSM-CC Data Carousel DII. In these instances the standard mechanisms should be used, provided they support the following set of requirements.

##### 4.5.2.2.1 Container map requirements

A specific container map implementation shall meet the following requirements:

- Signal the Id of each container - Containers are identified using their `container_id`. This shall be a unique number within the scope of a TVA metadata fragment stream. It is required that the id of a container is signalled at the Container map level, to enable the acquisition of a container with a given `container_id`.
- Identify the version of each container - The current version of each container shall be signalled and this shall increment whenever the contents of a container change. It shall be possible to monitor at a single point for version changes to a container. Ideally it should be possible to monitor just data containers, or just containers forming a single index.
- Identify the type of container i.e. Index Container, Data Container - This defines the type of data carried with a container and can be used by a receiving device to filter for example containers carrying TVA fragments.
- The ability to download all document containers (preferably in a parallel manner).
- Enable the ability to carry multiple TVA metadata fragment streams on the same delivery channel.

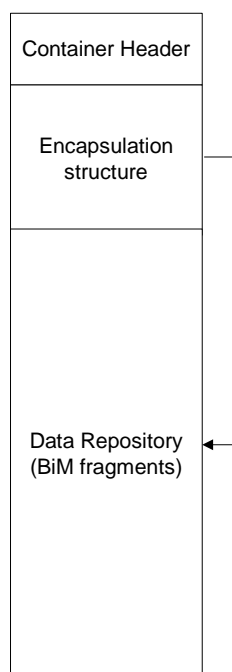
## 4.6 Fragment encapsulation

As described above a TVA-Anytime metadata description is split into a number of fragments, where a fragment forms a self-consistent unit of data. To enable a receiver to efficiently identify a change to a TVA fragment an encapsulation format has been defined. This provides TVA fragment Version information and an identifier specific to a TVA fragment. This enables a receiver to quickly identify fragments that have changed in relation to that cached on the receiving terminal.

### 4.6.1 Encapsulation format

The encapsulation data is provided by the encapsulation structure defined in figure 29. This structure is then transmitted along with the fragments using a "container" as described in clause 4.5.1.

**Data Container**



**Figure 29: Schematic representation of interrelationship between structures within a container**

#### 4.6.1.1 Encapsulation structure

The Encapsulation structure provides the encapsulation mechanism for a set of TVA fragments, by providing the ability to assign a unique identifier (`fragment_id`) for the lifetime of a TVA fragment and indicating the current version of a TVA fragment.

Each entry references a single TVA fragment carried within a `binary_data_repository` structure carried within the same container.

There shall only ever be one encapsulation structure within a single container.

Syntax	No. of Bits	Mnemonic
encapsulation_structure () {		
encapsulation_header {		
reserved_other_use	2	bslbf
reserved	6	bslbf
fragment_reference_format	8	uimsbf
}		
for(j=0; j<fragment_count; j++) {		
encapsulation_entry {		
fragment_reference()		
fragment_version	8	uimsbf
fragment_id	24	uimsbf
}		
}		
}		

**reserved\_other\_use:** This field shall be set to "11".

**fragment\_reference\_format:** This 8 bit value defines the format and interpretation of the `fragment_reference` field.

**Table 6: Valid fragment\_reference\_formats**

Value	Meaning
0x00	Reference to a BiM encoded fragment (see clause 4.6.1.3.1)
0x01 - 0xE0	TVA Reserved
0xE1 - 0xFF	User defined

**fragment\_reference:** A reference to a fragment, the interpretation of this field is dependent on the `fragment_reference_format`. Please refer to table 6 to determine how this field should be interpreted.

**fragment\_version:** An 8 bit value which identifies the version of the fragment referenced by this entry. When the data for the fragment identified by the `fragment_id` changes, the `fragment_version` shall increment modulo 255.

**fragment\_id:** A 24 bit value which uniquely identifies a metadata fragment within the TVA metadata fragment stream. The value assigned to a fragment shall be persistent for the life of that fragment so long as it is transmitted in the TVA metadata fragment stream. All entries within the `encapsulation_structure` shall be ordered by ascending `fragment_id`. This enables the efficient location of a fragment by using a binary search algorithm.

#### 4.6.1.2 Moved fragments structure

The `moved_fragments` structure is used to signal when a fragment has moved from one container to another. This is achieved by making an entry within the moved fragments structure carried within the fragments last container.

There shall be a maximum of one `moved_fragment` structure within a single container.

Syntax	No. of Bits	Mnemonic
moved_fragments () {		
for(i=0; i<num_of;i++){		
fragment_id	24	uimsbf
new_container_id	16	uimsbf
}		
}		

**fragment\_id:** The id of the fragment which has moved containers. All entries within the `moved_fragment_structure` shall be ordered by ascending `fragment_id`.

**new\_container\_id:** The id of the container, to where the fragment has moved.

### 4.6.1.3 Fragment\_Reference formats

There are a number of defined fragment locator formats to enable the referencing of fragments from within an encapsulation structure.

#### 4.6.1.3.1 Referencing a BiM encoded fragment

Syntax	No. of Bits	Mnemonic
BiM_fragment_reference () {		
BiM_fragment_ptr	16	uimsbf
}		

**BiM\_fragment\_ptr:** Offset in bytes from the start of the Binary repository within the container where the first byte of the FragmentUpdateUnit () for the BiM encoded fragment can be found.

### 4.6.1.4 Data repository

The Data Repository forms the base structure, used to hold string data and binary data. All references to the data repository are local i.e. from within the container. The type of data, which the data repository carries, is indicated by the structures associated structure\_id in the container\_header.

There may be more than one Data Repository within a container. However there shall only ever be a maximum of one data repository of a given type.

Syntax	No. of Bits	Mnemonic
data_repository () {		
if (structure_id == 0x00) {		
string_repository()		
}		
else if (structure_id == 0x01) {		
binary_repository()		
}		
else {		
user_defined_data_structure()		
}		
}		

**structure\_id:** An 8 bit value used to specify the type of data carried within this data repository. The structure\_id is not defined within this structure, but forms part of the structure instantiation in the container header (see clause 4.5.2.1).

#### 4.6.1.4.1 Binary data repository

The encoding of data in the binary repository is defined at the point of reference. Each item of data must either have a length explicitly encoded within it, or a length implicitly understood by the decoder (i.e. fixed length). No provision is made to define the data length within the binary data repository structure.

All entries shall be byte aligned.

There shall only ever be one binary data repository within a single container.

Syntax	No. of Bits	Mnemonic
binary_repository() {		
for (i=0; i<value_count; i++) {		
for (j=0; j<length; j++) {		
value_byte	8	bslbf
}		
}		
}		

**value\_byte:** A byte of binary value data.

#### 4.6.1.4.1.1 Carriage of BiM encoded fragments

Where the `binary_data_repository` contains BiM encoded data, a single binary data repository shall hold a single complete BiM Access Unit.

Syntax	No. of Bits	Mnemonic
<code>binary_repository() {</code>		
<code>BiMAccessUnit {</code>		
<code>NumberOfFUU</code>	8+	vluimsbf8
<code>for(i=0; i&lt; NumberOfFUU; i++) {</code>		
<code>FragmentUpdateUnit()</code>		
<code>}</code>		
<code>}</code>		
<code>}</code>		

#### 4.6.1.5 Alternative Encoding formats

Where BiM is not used for the encoding of fragments, the encoding solution:

- Shall provide a mechanism for indicating the type of TVA fragment.
- Optionally provide a mechanism for indicating the action to be performed e.g. Update fragment, Delete fragment, etc.

## 4.7 Fragment Management

In the previous clause a number of structures have been specified to enable the encapsulation of TVA fragments, within a TVA metadata fragment stream. These TVA fragments are dynamic and may change during their lifetime. In addition new TVA fragments will be added and old TVA fragments removed. The following describes how the defined encapsulation structures shall be used to enable the addition, deletion and updating of TVA fragments.

### 4.7.1 Fragment Id

The `fragment_id` is a 24 bit value which uniquely identifies a TVA fragment within a single TVA metadata fragment stream. This can be used by an application to track a TVA fragment during its lifetime within a TVA metadata fragment stream. It is valid to re-use a `fragment_id` value, provided that sufficient time has elapsed since the `fragment_id` value was last used.

### 4.7.2 Fragment Add

The addition of a new TVA fragment is straightforwardly achieved by creating an entry within the encapsulation structure having a unique `fragment_id` and an appropriate `fragment_version` and inserting the fragment into the binary data repository. The addition of a new entry into an existing container will cause the container's version number to increment.

### 4.7.3 Fragment Update

An update to a previously transmitted TVA fragment will cause the TVA fragment within the binary data repository to be updated. In addition the version number of the TVA fragment shall be incremented. These changes will cause the version number of the container in which the fragment is carried to increment.

## 4.7.4 Fragment Move

In some situations it may be required to move fragments between containers in which they are transmitted. It is recommended for efficiency and ease of management of fragments that this is kept to a minimum. However when required to move fragments between containers, the following should occur: The fragment is inserted into the new container, along with its original `fragment_id` and current `fragment_version`. An entry is inserted into the `moved_fragments` structure of the fragments previous container to indicate in which container the fragment can now be found. These operations will cause both the previous and the current container to have their version number incremented.

## 4.7.5 Fragment Delete

The deletion of a fragment can mean one of two things:

- The originally transmitted fragment is valid, but has been removed from the metadata fragment stream, as the content it describes is no longer available. For example the transmission time has passed.
- The originally transmitted fragment was invalid and so should not be used and discarded.

Both of these scenarios are supported as follows: when a fragment disappears from the metadata fragment stream it shall be treated as if the fragment has been deleted. Care should be taken when determining if a fragment is no longer in the TVA metadata fragment stream. If a fragment's `fragment_id` is not found in the expected containers encapsulation structure, the `moved_fragments` structure should be searched for an entry indicating that the fragment has moved containers. However if the metadata fragment stream is not constantly monitored for changes this mechanism should not be relied upon, as a fragment may have moved containers, but the corresponding fragment moved entry is no longer transmitted. The deletion of a fragment will cause the corresponding containers version number to be incremented.

In addition to the above methods, if BiM is used for fragment encoding and the `PositionCodeFlag` within the `TVAInit` is set to "1", the `BiM FragmentUpdateCommand`, which forms part of the `FragmentUpdateUnit` shall be used to signal the deletion of a fragment. In this case when a fragment is deleted the referenced BiM fragment will consist of a `FragmentUpdateUnit` with a `DeleteContent` command and no `FragmentUpdatePayload`. The change of fragment will cause the `fragment_version` within the encapsulation structure to be incremented.

This BiM Delete fragment command (see clause 4.4.2.2.1.3) if used shall be transmitted for a limited period of time, after which it should be removed. Receiver implementers should be aware that this mechanism should not be relied upon, as the receiver may be switched off or tuned away from the TVA metadata fragment stream, during the transmission of the Delete Command.

## 4.8 Indexing

### 4.8.1 Introduction

Data originally encapsulated in an XML document is not always best accessed as if it were an XML document when in the broadcast environment. In this environment, navigation of the document tree is relatively slow even when the location of the data in the tree is known. If the location is not known then the receiver must search through a set of data looking for a node with a particular value. In most cases this will be too slow to be practical. Indexing seeks to avoid these problems by avoiding the need to navigate the document tree. Indices provide direct access to a document TVA fragment by listing the values of a particular node (the index's key fields) and describing where the matching fragment(s) can be found in the carousel. Multiple indices can point to the same fragment, each using a different node as a key field.

### 4.8.2 Requirements

The indexing system shall be designed for metadata available in a unidirectional environment and whose TVA metadata fragments are carouselled.

The indexing system shall be compatible with any specific carousel format used to carry the TVA fragments.

The indexing system shall be designed in such a way that the data it uses may be broadcast cyclically but without being tied to a particular carousel mechanism.

The indexing system should be considered as a way to improve the navigability within the data set formed by the TVA fragments constituting a specific TVA metadata description, however this new system shall:

- be defined and used in addition and with regard to the existing solutions already standardized by *TV-Anytime*;
- be optional in the sense that for some metadata description or some application such an improved navigability may not be necessary or helpful.

The indexing system shall allow the indexing data to be used "on-line" when searching for a specific TVA fragment, namely without needing to be necessarily cached or completely acquired.

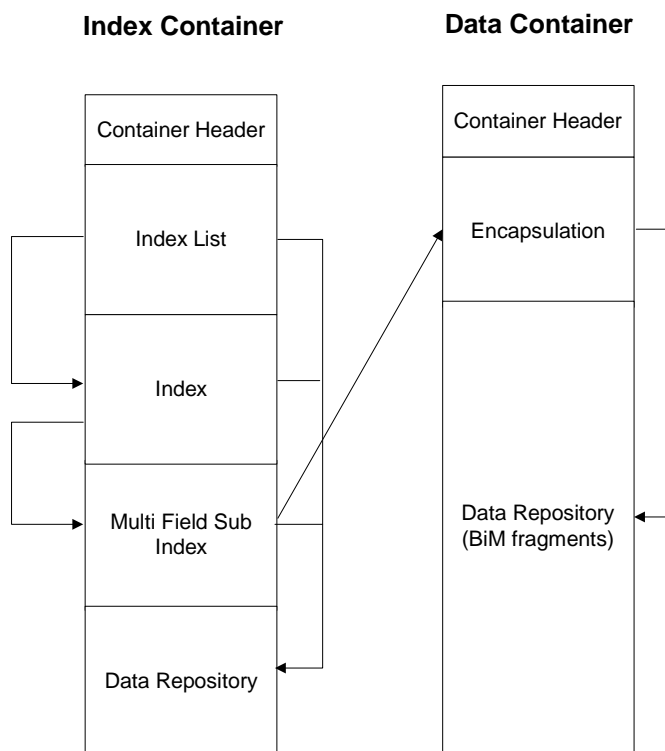
The indexing mechanism shall be defined as a way to retrieve using a certain index key a specific TVA fragment among all the TVA fragments constituting a metadata description and caroueled over a unidirectional stream:

- The nature of the key index may differ from one type of TVA fragment to another.
- For each of these possible standard index data type, *TV-Anytime* will specify what the encoded format and the sorting order are.
- The value of the key used to index a TVA fragment shall always be available directly or via an indirection in this TVA fragment.

The indexing mechanism should be extensible to support the possible definition of private new indexes through the use of hooks.

### 4.8.3 Carriage of Indexing Information

Indexing data is carried using the generic Container format specified in clause 4.4.2. The present clause makes use of some structures already define in clause 4.6 on fragment encapsulation.



**Figure 30: Schematic representation of interrelationship between Index containers and Data containers**



## 4.8.4 Data repository

The data repository within the context of an Index is used to carry data used in the representing of an index e.g. key field values. The syntax of a data repository is defined in clause 4.6.1.4 along with how binary data is carried within a data repository. In addition to binary data the data repository is used to carry Strings, as described below.

### 4.8.4.1 String repository

The string repository is used to hold all strings used by structures within the same container.

There shall only ever be one string repository per container. References to this repository are always local (that is, from the same container). Support is provided for identifying the string encoding system, to enable the use of non ASCII based character sets. The use of length fields or termination values are dependent on the string encoding used.

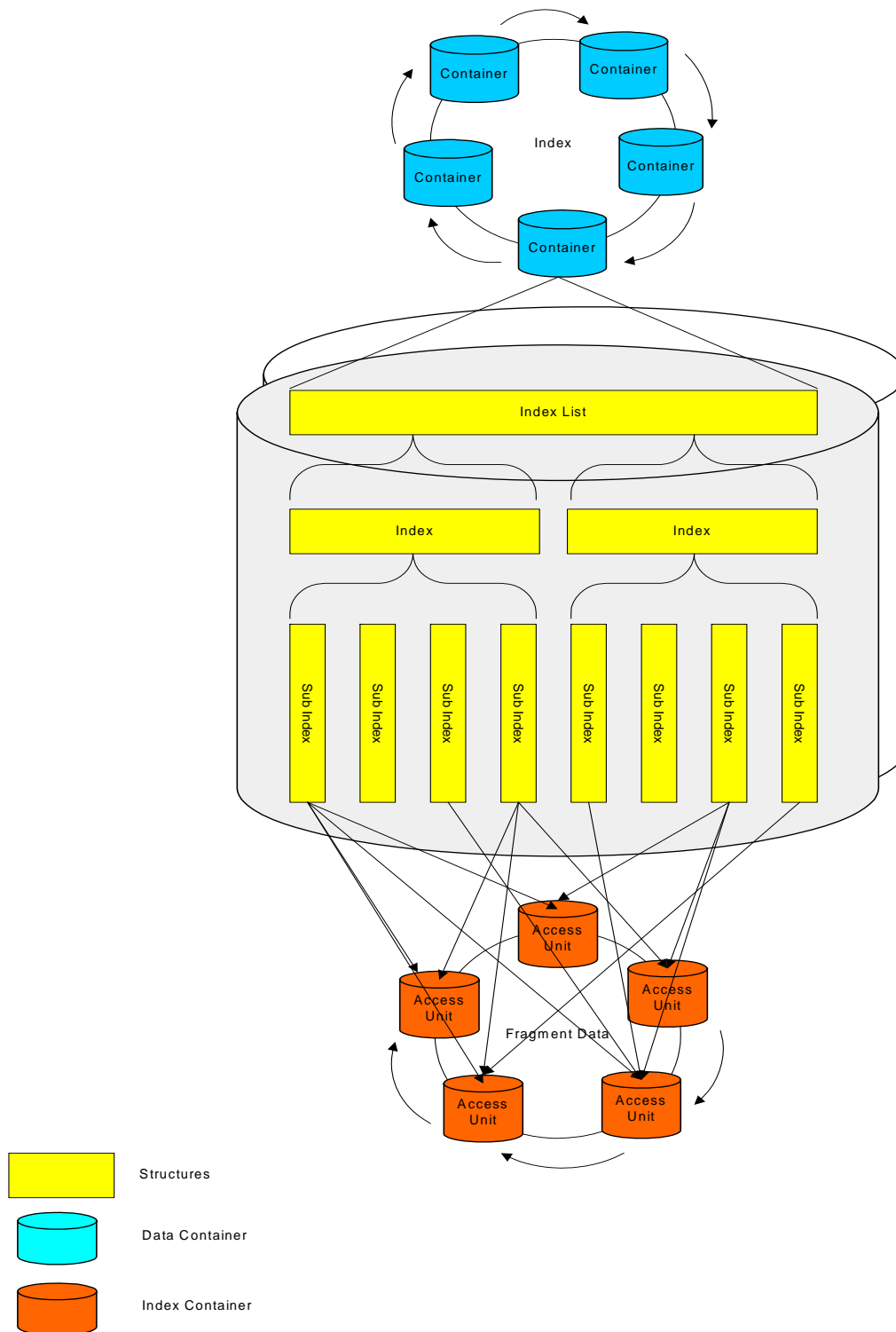
Syntax	No. of Bits	Mnemonic
string_repository() {		
encoding_type	8	uimsbf
for (i=0; i<strings_count; i++) {		
for (j=0; j<string(i).length; j++) {		
string_character	8+	
}		
string_terminator	8+	bslbf
}		
}		

**encoding\_type:** An 8 bit field used to define the character encoding system, according to table 2.

**string\_character:** A character of the encoded string. The number of bytes required to represent the character will be dependent on the string encoding system used.

**string\_terminator:** One or more bytes which indicate the end of a string. The actual value will be dependent on the string encoding system used.

## 4.8.5 Index structures



**Figure 31: Indexing structure**

### 4.8.5.1 Identification of indices

Indexes are keyed on schema simple types such as dateTime, CRID. Multiple indices can reference a single TVA fragment. It is important to be able to describe what an index is keyed on.

There are two important facts that the receiver needs to know about an index:

- The first fact is where in the schema hierarchy are the document fragments that it is referencing? It is important to know this because this describes the type and context of the fragment.
- The second fact is to know "by which key fields are these fragments indexed?" The key fields are typically child members (e.g. an attribute or an element) of the fragment type being indexed.

The specification provides two mechanisms:

- Id based identification.
- XPath expressions.

#### 4.8.5.1.1 Use of Ids

Ids can be used to identify both the type of TVA fragment and the element/attribute within the fragment, which the index is keyed on. The set of normative TVA fragments have been mapped to aid interoperability and a mechanism provided to enable metadata providers to define additional Ids to enable the support of enhanced metadata services.

#### 4.8.5.1.2 Use of XPath

W3C has defined a standard specifically for the referencing of elements and attributes within an XML document, which is called XPath. XPath is a syntax, which can describe a path to one or more nodes in a document.

The fragment XPath is an *absolute* path (i.e. it is relative to the root of the document), whilst the key XPath is relative to the fragment XPath. In other words, the context node of the key XPath is the node referred to by the fragment XPath. Combined together they describe the absolute path to the node that forms a key field. The XPath syntax can describe the location of any type of node including elements, attributes and text nodes, enabling any of these to be a key field.

The XPath syntax is rich and many parts are not necessary to describe an index. Therefore a restricted set of syntax, which a TVA compliant box should support has been specified:

- Absolute Location Paths only (*key\_xpath* is relative, but the combined path is absolute).
- Axes types "attribute" and "child" only are supported.
- Abbreviated Syntax only. The preceding restrictions mean that only the following two abbreviations are permitted:
  - "child::" is always omitted;
  - "@" is always used to represent "attribute::".
- "\*" is not allowed.
- Only the following two Node tests are allowed:
  - NameTest (in which "\*" is not allowed);
  - text().
- Predicates and Functions are not allowed.
- The union operator "|", is not allowed.

All XPath expressions will be evaluated within the following context:

- The context node is the root of an XML document containing a TVAMain element.
- The context position and context size are both 1.
- There are no variable bindings.
- There is no function library.

In addition all elements/attributes shall be namespace qualified.

#### 4.8.5.1.2.1 Example: Indexing by CRID and title

Considering a broadcast carousel that delivers fragments of type ProgramInformation the fragment XPath (in short form) would be:

```
/tva:TVAMain/tva:ProgramDescription/tva:ProgramInformationTable/tva:ProgramInformation
```

The most likely key to use for searching this set of data is the CRID. The field\_xpath is relative to the fragment XPath i.e.:

```
@tva:programId
```

A broadcaster may wish to index by title, as well as by CRID, to enable the receiver to search by title. The fragment XPath is:

```
/tva:TVAMain/tva:ProgramDescription/tva:ProgramInformationTable/tva:ProgramInformation
```

and the field\_xpath is:

```
tva:BasicDescription/tva:Title.text()
```

#### 4.8.5.2 Introduction to the multi-key index

Some applications in the receiver may request matching fragments for more than one query condition. In this case, using multiple key fields is quite efficient to answer such requests. The multi-key consists of more than one key fields.

Multi-key values are ranked in order as follows. For a multi-key of  $n$  key fields  $(k_1, k_2, \dots, k_n)$ , priority of each key field is ordered according to its position from left to right, i.e.,  $k_1$  has the highest priority and  $k_n$  has the lowest priority, etc.

For two multi-key values,  $(a_1, a_2, \dots, a_n)$  and  $(b_1, b_2, \dots, b_n)$ :

- $(a_1, a_2, \dots, a_n)$  is larger than  $(b_1, b_2, \dots, b_n)$  if and only if there exists an integer  $i$  ( $0 \leq i \leq n-1$ ) such that for every  $j$  ( $0 \leq j \leq i-1$ ),  $a_j = b_j$  and  $a_i > b_i$ .
- $(a_1, a_2, \dots, a_n)$  is smaller than  $(b_1, b_2, \dots, b_n)$  if and only if there exists an integer  $i$  ( $0 \leq i \leq n-1$ ) such that for every  $j$  ( $0 \leq j \leq i-1$ ),  $a_j = b_j$  and  $a_i < b_i$ .
- $(a_1, a_2, \dots, a_n)$  is equal to  $(b_1, b_2, \dots, b_n)$  if and only if for every  $i$  ( $1 \leq i \leq n$ ),  $a_i = b_i$ .

A typical query example that can be efficiently handled by the multi-key index is as follows:

- Search target fragment:

```
/TVAMain/ProgramDescription/ProgramLocationTable/BroadcastEvent
```

- Search condition:

```
100 <= ServiceId <= 110
```

```
9:00 PM <= PublishedStartTime.text() <= 10:00 PM
```

In this case, the "BroadcastEvent" fragments can be indexed by a multi-key index with key fields ServiceId and PublishedStartTime.

### 4.8.5.3 Index List

The index list structure provides a list of all indices that exist for the entire TVA metadata fragment stream. A receiver uses this structure to locate an index of interest, where an index is described using the structures `fragment_type` and `field_identifier` fields.

It should be noted that entries within the structure are not of fixed size. The Index list must be searched sequentially. The `index_descriptor_length` field is provided to enable a receiver to efficiently skip over index entries, which it is unable to parse due to unknown `fragment_types`, `field_identifiers`, or `field_encoding`.

There shall be a maximum of one index list structure per TVA metadata fragment stream.

Syntax	No. of Bits	Mnemonic
<code>index_list() {</code>		
<code>for (j=0; j&lt;num_indexes, j++) { ...</code>		
<code>index_descriptor_length</code>	8	uimsbf
<code>fragment_type</code>	16	uimsbf
<code>if(fragment_type == 0xffff) {</code>		
<code>fragment_xpath_ptr</code>	16	uimsbf
<code>}</code>		
<code>num_fields</code>	8	uimsbf
<code>for(k=0; k&lt;num_fields; k++) { ...</code>		
<code>field_identifier</code>	16	uimsbf
<code>if(field_identifier == 0xffff) {</code>		
<code>field_xpath_ptr</code>	16	uimsbf
<code>}</code>		
<code>field_encoding</code>	16	uimsbf
<code>}</code>		
<code>index_container</code>	16	uimsb
<code>index_identifier</code>	8	uimsbf
<code>}</code>		
<code>}</code>		

**index\_descriptor\_length:** An 8 bit field which defines the number of bytes proceeding this field which are used to describe the index.

**fragment\_type:** An id used to identify the type of TVA fragments which the index makes references to. In addition to the identifiers defined in the table below, their may be a set of unique identifier allocated on a per application basis.

Table 7 shows the set of allocated `fragment_type` values. The XPath expressions are namespace qualified and use the following namespace prefixes:

tva	urn:tva:metadata:2005
tva2	urn:tva:metadata:extended:2005
mpeg7	urn:tva:mpeg7:2005
int	urn:tva:metadata:interstitial:2005

Table 7: Fragment\_type assignments

Value	Description
0x0000	Reserved
0x0001	ProgramInformation fragment (/tva:TVAMain/ tva:ProgramDescription/ tva:ProgramInformationTable/ tva:ProgramInformation)
0x0002	GroupInformation fragment (/tva:TVAMain/ tva:ProgramDescription/ tva:GroupInformationTable/ tva:GroupInformation)
0x0003	OnDemandProgram fragment (/tva:TVAMain/tva:ProgramDescription/tva:ProgramLocationTable/tva:OnDemandProgram)
0x0004	BroadcastEvent fragment (/tva:TVAMain/tva:ProgramDescription/ tva:ProgramLocationTable/ tva:BroadcastEvent)
0x0005	Schedule fragment (/tva:TVAMain/ tva:ProgramDescription/ tva:ProgramLocationTable/ tva:Schedule)
0x0006	ServiceInformation fragment (/tva:TVAMain/ tva:ProgramDescription/ tva:ServiceInformationTable/ tva:ServiceInformation)
0x0007	PersonName fragment (/tva:TVAMain/ tva:ProgramDescription/ tva:CreditInformationTable/ tva:PersonName)
0x0008	OrganizationName fragment (/tva:TVAMain/tva:ProgramDescription/tva:CreditInformationTable/tva:OrganizationName)
0x0009	ProgramReviews fragment (/tva:TVAMain/ tva:ProgramDescription/ tva:ProgramReviewTable/ tva:Review)
0x000A	CSAlias fragment (/tva:TVAMain/ tva:ClassificationSchemeTable/tva:CSAlias)
0x000B	ClassificationScheme fragment (/tva:TVAMain/tva:ClassificationSchemeTable/tva:ClassificationScheme)
0x000C	Segment Information fragment (/tva:TVAMain/ tva:ProgramDescription/ tva:SegmentationTable/ tva:SegmentList/ tva:SegmentInformation)
0x000D	Segment Group Information fragment (/tva:TVAMain/ tva:ProgramDescription/ tva:SegmentationTable/ tva:SegmentGroupList/ tva:SegmentGroupInformation)
0x000E	TVAMain fragment (/tva:TVAMain)
0x000F	OnDemandService fragment (/tva:TVAMain/tva:ProgramDescription/ tva:ProgramLocationTable/ tva:OnDemandServiceType)
0x0010	PurchaseInformation fragment (/tva:TVAMain/tva:ProgramDescription/tva:PurchaseInformationTable/tva:PurchaseInformation)
0x0011	MetadataOriginationInformation fragment (/tva:TVAMain/tva:MetadataOriginationInformationTable/tva:MetadataOriginationInformation)
0x0012	Package fragment (/tva:TVAMain/tva2:PackageTable/tva2:Package)
0x0013	InterstitialCampaign Fragment (/tva:TVAMain/tva2:InterstitialCampaignTable/tva2:InterstitialCampaign)
0x0014	RMPIDescription Fragment (/tva:TVAMain/tva2:RMPITable/tva2:RMPIDescription)
0x0015	CouponDescription Fragment (/tva:TVAMain/tva2:CouponTable/tva2:CouponDescription)
0x0016	TargetingInformation Fragment (/tva:TVAMain/tva2:TargetingInformationTable/tva2:TargetingInformation)
0x0017	Rule Fragment (/tva:TVAMain/tva2:InterstitialTargetingTable/int:RulesTable/int:Rule)
0x0018	Request Fragment (/tva:TVAMain/tva2:InterstitialTargetingTable/int:RecordingCacheTable/int:Request/int:Item)
0x0019	Replace Fragment (/tva:TVAMain/tva2:InterstitialTargetingTable/int:RecordingCacheTable/int:Replace/int:Item)
0x0020	Expire Fragment (/tva:TVAMain/tva2:InterstitialTargetingTable/int:RecordingCacheTable/int:Expire/int:Item)
0x0021	InterstitialBreak Fragment (/tva:TVAMain/tva2:InterstitialTargetingTable/int:InterstitialTable/int:InterstitialBreak)
0x0022 - 0xFFFF	User defined
0xFFFF	W3C XPath Expression

**fragment\_xpath\_ptr:** If the fragment\_type is set to "0xffff" this provides a reference to the start of an XPath string. This reference is in the form of an offset, in bytes, from the start of the string repository in the current container. The value of this string is the XPath (in abbreviated XPath notation) to the root node of a TVA fragment.

**num\_fields:** The number of fields which this index is based upon. The fields shall be defined in their order of importance, where the first entry is the Primary field.

**field\_identifier:** An id used to identify the field on which the index is ordered. The specification allows the allocation of unique identifiers on a per application basis.

**Table 8: field\_identifier Assignments**

Value	Description
0x0000	Reserved
0x0001 - 0xFFFFE	User defined
0xFFFF	Indicates use of a W3C XPath style expression

**field\_xpath\_ptr:** Reference to the start of the fields XPath string (in abbreviated XPath notation) within the string repository belonging to the current container. This reference is in the form of an offset, in bytes, from the start of the string repository in the current container. The value of this string is an XPath expression that is relative to the `fragment_types` XPath, which identified the node that is used as one of the indexes key fields.

**field\_encoding:** Defines the encoding used to represent a key field value. This encoding determines how the "field\_value" within the `multi_field_sub_index`, the `low_field_value` and `high_field_value` within the index, shall be interpreted.

The `field_encoding` value has two purposes:

- It determines whether the content of the `field_value` within the `multi_field_sub_index` structure, the `low_field_value` and the `high_field_value` within the index structure, are inline or are found within a data repository structure (see table 4).
- It defines the encoding of the data held in the field-value within the `multi_field_sub_index` structure, the `low_field_value` and the `high_field_value` within the index structure (see table 9).

**Table 9: Encoding and interpretation of the field\_value, low\_field\_value and high\_field\_value field**

Encoding value	value field interpretation
0x0000 - 0x00FF	Field is a 16-bit offset in bytes from the start of the string data repository structure.
0x0100 - 0x01FF	Field contains an inline 2-byte value.
0x0200 - 0x0201 0x0300 0x0401	Field contains an inline 4-byte value.
0x0204 - 0x0206	Field contains an inline 1-byte value.
0x0202 - 0x0203	Field is a 16 bit offset in bytes from the start of the <code>binary_data_repository</code> structure
0x0302 0x0400	Field contains an inline 8-byte value.
0x0204 - 0x02FF 0x0402-0x04FF	Undefined
0x0500-0xFFFF	Reserved for future use.

Table 10: Encoding types and their respective sizes

field_encoding	Description	Encoding	Number of bits
0x0000	string type	Null-terminated string	variable (8+)
0x0001 - 0x00FF	Reserved for custom string types		
0x0100	signed short	two's complement - Big-Endian	16
0x0101	unsigned short	unsigned binary - Big-Endian	16
0x0102-0x01FF	Reserved for custom 2-byte types		16
0x0200	signed long	two's complement - Big-Endian	32
0x0201	unsigned long	unsigned binary - Big-Endian	32
0x0202	variable length signed integer	one bit to indicate sign (0: positive, 1: negative), followed by abs(value) using vluimsbf5.	variable (6+)
0x0203	variable length unsigned integer	vluimsbf8	variable (8+)
0x0204	boolean	0:False 1:True	8
0x0205	signed byte	two's complement	8
0x0206	unsigned byte	unsigned binary	8
0x0207-0x02FF	Reserved for custom integer types		
0x0300	signed float	IEEE standard 754-1985 [3] Big-Endian	32
0x0301	Reserved		
0x0302	signed double	IEEE standard 754-1985 [3] Big-Endian	64
0x0303-0x03FF	Reserved for custom rational types		
0x0400	dateTime	Modified Julian Date and Milliseconds (TVA BiM codec, clause 4.4.2.4.2)	64
0x0401	date	Modified Julian Date (TVA BiM codec, clause 4.4.2.4.3)	32
0x0402-0x04FF	Reserved for custom binary fragments		
0x0500-0xFFFF	Reserved for future use		

**index\_container:** The id of the container carrying the described index.

**index\_identifier:** This field identifies the relevant index structure within the identified container. To locate the correct index within the container (with `id = index_container`) the `container_header` is searched for a structure of type "index" and with a `structure_id = index_identifier`.

#### 4.8.5.4 Index

The index structure is the top level of an index. It provides a list of all sub-indices and the ranges of field values that those sub-indices carry. When considering a classic indexing system it is normal for there not to be any overlaps in the range of field values to be found within a given set of sub indexes. This is to minimize the amount of searching required to find a particular value.

Having overlapping sub-indices can lead to sequential searching of sub index structures, introducing an associated decrease in performance. However in some circumstances it may be desirable to allow this. For example where the indexed data is carried within the same container as the index and you wish to carousel the data out at different rates and the set of data to be carouselled would not typically form a single sub index without overlaps.

In the case of overlapping sub indexes they shall be declared within the index structure in descending order of search priority. Where the first declared sub index, which may contain the set of required field values has the highest priority.

##### 4.8.5.4.1 Field Value Ordering

The ordering of index entries within an index is dependent on a field's primitive XML schema simple type. In the case of strings the order may be dependent on the selected language and not necessarily in alphanumeric order.



Table 11: Defined index order for primitive simple types

Simple Type	Ordering
string	All string shall be ordered in increasing Lexicographical order. Lexicographical ordering is language dependent and may not be alphanumeric.
anyURI	Increasing alphanumeric order.
boolean	"False" precedes "True".
NMTOKEN	Increasing binary representation order.
gYear	Increasing numeric value.
integer	Increasing numeric value with negative values first.
date	Increasing date value.
nonNegativeInteger	Increasing numeric (binary) value.
positiveInteger	Increasing numeric (binary) value.
dateTime	Increasing dateTime (binary) value.
duration	Increasing duration (binary).
float	Increasing numeric value (negative values first).
double	Increasing numeric value (negative values first).

Syntax	No. of Bits	Mnemonic
index() {		
overlapping_subindexes	1	bslbf
single_layer_sub_index	1	bslbf
reserved	6	bslbf
fragment_locator_format	8	uimsbf
for (j=0; j<num_sub_indexes, j++) { ...		
for(k=0; k<num_fields; k++) { ...		
if (overlapping_subindexes == '1') {		
low_field_value	field encoding dependent	uimsbf
}		
high_field_value	field encoding dependent	uimsbf
}		
sub_index_container	16	uimsbf
sub_index_identifier	8	uimsbf
}		
}		

Given high\_field\_values,  $(a_1, a_2, \dots, a_n)$  and  $(b_1, b_2, \dots, b_n)$ , of two arbitrary sub-indexes among the sub-indices list, The sorting of sub-indices is determined as follows:

$(a_1, a_2, \dots, a_n)$  is larger than  $(b_1, b_2, \dots, b_n)$  if and only if there exists an integer  $i$  ( $0 \leq i \leq n - 1$ ) such that for every  $j$  ( $0 \leq j \leq i - 1$ ),  $a_j = b_j$  and  $a_i > b_i$ .

$(a_1, a_2, \dots, a_n)$  is smaller than  $(b_1, b_2, \dots, b_n)$  if and only if there exists an integer  $i$  ( $0 \leq i \leq n - 1$ ) such that for every  $j$  ( $0 \leq j \leq i - 1$ ),  $a_j = b_j$  and  $a_i < b_i$ .

$(a_1, a_2, \dots, a_n)$  is equal to  $(b_1, b_2, \dots, b_n)$  if and only if for every  $i$  ( $1 \leq i \leq n$ ),  $a_i = b_i$ .

Specifically, within the index() structure, if there is no overlapping between subindices, for all  $j$  between 0 and num\_sub\_indexes-1 (high\_field\_value[j,0], ..., high\_field\_value[j,k]) is smaller than (high\_field\_value[j+1,0], ..., high\_field\_value[j+1,k])

"j" is the sub\_index

"k" is the field

This function high\_field\_value[j,k] takes its value according to the loop defined in the index() table.

**overlapping\_subindexes:** When set to "1", indicates that one or more of the sub indices which form this index, overlap with respect to the range of values found within the sub index. Where sub indices overlap, the sub indices are declared in descending order of search priority. When set to "0", indicates that the sub indices do not overlap and the declared sub indices are ordered in ascending order.

**single\_layer\_sub\_index:** This field is used to indicate the syntax used within the corresponding `multi_field_sub_index` structures to represent indexes with multiple key fields. When set to "1" it indicates that all fields for a given index entry are declared together in a single `multi_field_sub_index` structure. When set to "0" it indicates that each key field is contained within a separate `multi_field_sub_index` structure.

**fragment\_locator\_format:** Identifies the format and interpretation of the fragment locator field used within the `multi_field_sub_index` (leaf field) to reference a TVA fragment.

**Table 12: Fragment reference types**

Value	Meaning
0x00	local_fragment_locator (see clause 4.8.5.7.2)
0x01	remote_fragment_locator (see clause 4.8.5.7.1)
0x02 - 0xE0	TVA Reserved
0xE1 - 0xFF	User defined

**low\_field\_value:** The lowest field value that can be found within the sub-index. The meaning of this field depends on the value of the `field_encoding` member of the index list structure. The lowest value of the field expressed may not be the lowest field value actually present in the given fragment, it merely indicates that the referenced sub index structure may contain entries with fields values in the given range. The type of encoding used and the interpretation of the `low_field_value` are defined by the `field_encoding` within the `index_list` structure.

**high\_field\_value:** The highest field value that can be found within the sub-index. The meaning of this field depends on the value of the `field_encoding` member of the relevant index list structure (see clause 4.8.5.3). The highest value of the field expressed may not be the highest field value actually present in the given fragment, it merely indicates that the referenced sub index structure may contain entries for key field values in the given range. The type of encoding used and the interpretation of the `high_field_value` are defined by `field_encoding` within the `index_list` structure.

It should be noted that the `high_field_value` for all but the first field may be lower than the previous `high_field_value` sub index entry. This is caused when there is a change in the value of the parent field.

For example if we have an index keyed on channel and event time fields, we could have a set of sub indexes with the following ranges:

- Sub index 1 - channel `high_field_value` = "3", event time `high_field_value` = "12:00".
- Sub index 2 - channel `high_field_value` = "4" event time `high_field_value` = "09:00".

Where the index uses multiple fields, the declaration order of the `high_field_values` shall match that defined for the index within the `index_list` structure.

When defining the range of values that a particular sub index shall cover, sufficient space should be left to enable the addition of further index entries without unduly impacting other sub indices. For example if a sub-index can hold a maximum of say 64K entries, it is recommended that the range of current entries should equal around half to two thirds the space. This leaves sufficient room for additional entries without having to changing the way in which the index is split into sub index structures.

**sub\_index\_container:** The id of the container carrying the described `multi_field_sub_index`.

**sub\_index\_identifier:** This field identifies the `multi_field_sub_index` structure instance containing the described sub index. To locate the sub index within the container (with `id = sub_index_container`) the `container_header` is searched, for a structure of type "`multi_field_sub_index`" with a `structure_id` equal to the `sub_index_identifier`.

### 4.8.5.5 Multi\_field\_sub\_index

A multi field sub index provides references to TVA fragments, which contain field values within the range specified for this sub index. The structure supports indexes with both single and multiple key fields. In the case of indices with multiple key fields, the syntax provides two methods:

- Single Layer - All key fields defined together within a single `multi_field_sub_index`.
- Multi Layer - Each `multi_field_sub_index` defines a single field of a key.

#### 4.8.5.5.1 Single Layer Structures

Single Layer Structures provide a simple mechanism for describing multiple key field indices. As each entry in the structure can be decoded one by one in a straightforward manner, this structure would be preferred in a situation where the received index data need to be reorganized in the PDR before its use. Note that the index data can be restructured inside the PDR according to its own storage method and query processing policy. For example, a PDR may want to reorganize one of the received indices in its own B-tree index.

In addition, the Single Layer Structure provides an efficient mechanism for representing multiple key field indexes, where there is typically a one to one mapping e.g. <surname, givenname>.

#### 4.8.5.5.2 Multi Layer Structures

Multi Layer Structures provide an efficient mechanism for describing multiple field indices with common key field values. This is achieved with the use of multiple `multi_field_sub_index` structures (see figure 32), where each structure is used to describe one layer of a multi field sub index, (layer is equal to a key field of a multi field index).

Each index entry, within the `multi_field_sub_index`, points to further `multi_field_sub_index` structures (except for the leaf field), which contain index entries having the declared field value.

The `multi_field_sub_index` structure is formed of two parts:

- `multi_field_header`.
- `multi_field_index_entries`.

The `multi_field_header` defines how the `multi_field_index_entries` sub structure should be interpreted and indirectly defines the size of each index entry.

All entries within the `multi_field_index_entries` sub structure are ordered in ascending order.

All entries are of a fixed size, which enables the sub structure to be efficiently searched using a binary search algorithm.

The number of entries within the structure is not explicitly defined, but can be inferred as follows:

```
num_entries = (structure_length -
sizeof(multi_field_header))/sizeof(multi_field_index entry)
```

It should be noted that the syntax used within `multi_field_sub_index` structures is not always common across all sub indices. Therefore the header of each `multi_field_sub_index` should be parsed to infer the syntax used within a given instance.

Syntax	No. of Bits	Identifier
multi_field_sub_index() {		
multi_field_header {		
leaf_field	1	bslbf
multiple_locators	1	bslbf
reserved	6	bslbf
}		
if(single_layer_sub_index == '0') {		
multi_layer_sub_index_structure()		
} else {		
single_layer_sub_index_structure()		
}		
}		

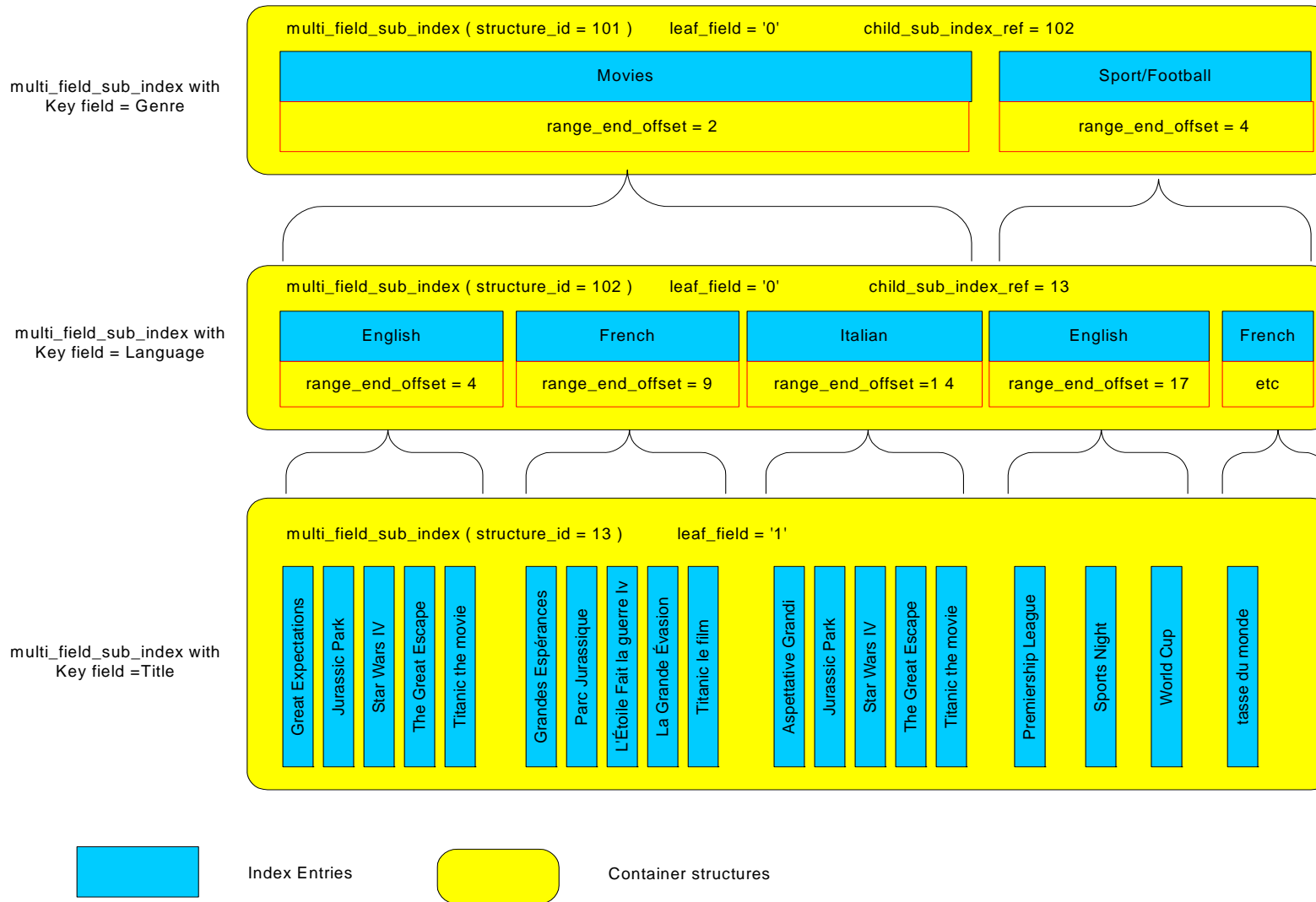


Figure 32: Example multi\_field\_sub\_index structure (using multi-layer syntax) for an index with three key fields (Genre, Language and Title)

Given `field_values`,  $(a_1, a_2, \dots, a_n)$  and  $(b_1, b_2, \dots, b_n)$ , of two `multi_field_entries`, the order between the two entries is determined as follows:

$(a_1, a_2, \dots, a_n)$  is larger than  $(b_1, b_2, \dots, b_n)$  if and only if there exists an integer  $i$  ( $0 \leq i \leq n - 1$ ) such that for every  $j$  ( $0 \leq j \leq i - 1$ ),  $a_j = b_j$  and  $a_i > b_i$ .

$(a_1, a_2, \dots, a_n)$  is smaller than  $(b_1, b_2, \dots, b_n)$  if and only if there exists an integer  $i$  ( $0 \leq i \leq n - 1$ ) such that for every  $j$  ( $0 \leq j \leq i - 1$ ),  $a_j = b_j$  and  $a_i < b_i$ .

$(a_1, a_2, \dots, a_n)$  is equal to  $(b_1, b_2, \dots, b_n)$  if and only if for every  $i$  ( $1 \leq i \leq n$ ),  $a_i = b_i$ .

Specifically, within the `multi_field_sub_index()` structure, for all  $j$  between 0 and `num_entries-1` (`field_value[j,0], ..., field_value[j,k]`) is smaller than (`field_value[j+1,0], ..., field_value[j+1,k]`)

**leaf\_field:** This shall be set to "1" when the `multi_field_sub_index` carries the leaf field of an index (last indexed field). Which indicates that the structure contains references to fragments and not to further `multi_field_sub_index` structures. This field is only used within multi layer sub indexes. When a single layer sub index is being described this flag shall be ignored.

**multiple\_locators:** A flag which when set to "1" indicates that there are potentially multiple referenced fragments which have the same set of key field values. This provides a more bandwidth efficient mechanism, when multiple fragments have the same set of key values. The actual fragment locators are carried in a separate structure within the container and an offset is used to reference the set of relevant locators within the structure. When the flag is set to "0" it indicates that `fragment_locators` are defined inline.

Syntax	No. of Bits	Mnemonic
<code>single_layer_sub_index_structure () {</code>		
<code>multi_field_index_entries {</code>		
<code>for (j=0; j&lt;num_entries; j++) { ...</code>		
<code>for(f=0; f&lt;num_fields; f++) {</code>		
<code>field_value</code>	field encoding dependent	uimsbf
<code>}</code>		
<code>if(multiple_locators == '1') {</code>		
<code>locator_end_offset</code>	16	uimsbf
<code>}</code>		
<code>else {</code>		
<code>fragment_locator()</code>		
<code>}</code>		
<code>}</code>		
<code>}</code>		

**field\_value:** The value of the key field of the referenced fragment. The meaning of this field depends on the value of the `field_encoding` member of the relevant index list structure (see clause 4.8.5.4). Only values of the key field within the range given for this `sub_index` structure, by the relevant index structure are allowed (see clause 4.8.5.4).

**fragment\_locator:** When the `multiple_locators` flag is set to "0" in the `multi_field_header` this field is used to reference a fragment, having the set of specified key field values. The format of this locator is dependent on the `fragment_locator_format` defined for this index within the `index_list` structure. For an explanation of the various defined locator formats please refer to clause 4.8.5.7.

**locator\_end\_offset:** When the `multiple_locators` flag is set to "1" in the `multi_field_header` this field is used to indicate the inclusive end offset within the `fragment_locators` structure where the set of valid locators can be found. The format of these locators is defined by the `fragment_locator_format` declared within the `index_list` structure. The `fragment_locator` structure instance in which the locators can be found is the `fragment_locator` structure with the same `structure_id` value as that of this sub index structure.

The `locator_start_offset` is implicit from the previous entry within the `multi_field_sub_index`, as follows:

- If it is the first entry within the `multi_field_index_entries` then `locator_start_offset` shall equal 0.
- If it is not the first entry, the previous entries `locator_end_offset + 1` shall be used as the current entries inclusive `locator_start_offset`.

```
if (current_index != 0) {
    locator_start_offset = multi_field_index_entries[current_index-1].locator_end_offset
+ 1;
} else {
    locator_start_offset = 0;
}
```

It should be noted that these references are based on fragment locator entries and not byte offsets. The actual byte offset within the `fragment_locators` structure is calculated as follows:

```
byte_offset = locator_end_offset * sizeof(fragment_locator);
```

Syntax	No. of Bits	Mnemonic
<code>multi_layer_sub_index_structure () {</code>		
<code>if (leaf_field='0') {</code>		
<code>child_sub_index_ref</code>	8	uimsbf
<code>}</code>		
<code>multi_field_index_entries {</code>		
<code>for (j=0; j&lt;num_entries; j++) { ...</code>		
<code>field_value</code>	field encoding dependent	uimsbf
<code>if(leaf_field == '1') { ...</code>		
<code>if(multiple_locators == '1') {</code>		
<code>locator_end_offset</code>	16	uimsbf
<code>}</code>		
<code>else {</code>		
<code>fragment_locator()</code>		
<code>}</code>		
<code>}</code>		
<code>else {</code>		
<code>range_end_offset</code>	16	uimsbf
<code>}</code>		
<code>}</code>		
<code>}</code>		

For all fields not described, please refer to the semantics for the `single_layer_sub_index_structure` fields found above.

**child\_sub\_index\_ref:** This value identifies a further `multi_field_sub_index` structure within the current container which holds index entries having a field value equal to that defined within this sub index. The combination of this value and the `range_end_offset` enables you to locate a set of index entries, which have a specific key field value.

**range\_end\_offset:** This field defines the set of entries within the referenced `multi_field_sub_index` (a `multi_field_sub_index` with its `structure_id` equal to `child_sub_index_ref`) having a key field value equal to that defined by the `field_value`.

The value is an inclusive offset from the start of the `multi_field_index_entries` of the target `multi_field_sub_index`, where the `end_offset` for the set of entries that have the declared value can be found. The `range_start_offset` is implicit from the previous, entry within the `multi_field_sub_index`, as follows:

- If it is the first entry within the `multi_field_index_entries` then `start_offset` shall equal 0.
- If it is not the first entry, the previous entries `range_end_offset + 1` shall be used as the current entries inclusive `range_start_offset`.

```

if (current index != 0) {
    range_start_offset = multi_field_index_entries[current index-1].range_end_offset +
1;
} else {
    range_start_offset = 0;
}

```

It should be noted that these references are based on index entries and not byte offsets. So the actual byte offset within the structure is calculated as follows:

```

byte_offset = (range_end_offset * sizeof(multi_field_index_entry)) +
sizeof(multi_field_header)

```

#### 4.8.5.6 Fragment locators structure

The `fragment_locators` structure is used to carry `fragment_locators`, where there are multiple fragments with the same set of key field values. This structure is referenced by the `multi_field_sub_index` structure.

There can be multiple `fragment_locators` structures within a single container and the `structure_id` is used to identify an instance. The `structure_id` shall be set to the same `structure_id` value to that of the associated sub index structure. So for example if the sub index has a `structure_id` of "0x03" then the corresponding `fragment_locators` structure will have a `structure_id` of "0x03".

Syntax	No. of Bits	Mnemonic
<code>fragment_locators() {</code>		
<code>for(int i=0; i&lt;num_locators; i++) { ...</code>		
<code>fragment_locator()</code>		
<code>}</code>		
<code>}</code>		

**num\_locators:** This value is inferred from the size of the structure which is declared within the containers `container_header`.

I.e. `num_locators = structure_length/sizeof(fragment_locator);`

**fragment\_locator:** This field is used to convey a reference to a TVA fragment. The format of this locator is dependent on the `fragment_locator_format` defined for this index within the `index_list` structure. For an explanation of the various defined locator formats please refer to clause 4.8.5.7.

#### 4.8.5.7 Fragment\_locator formats

There are a number of defined fragment locator formats to enable the referencing of fragments from an index entry.

##### 4.8.5.7.1 Referencing fragments in another container

When a data structure becomes quite large, or it is a requirement to be able to carousel the index at a different rate to that of the data, it is advantageous to split the index and data across independent containers. This format provides a mechanism for an index entry to reference a TVA fragment within another container.



Syntax	No. of Bits	Mnemonic
remote_fragment_locator() {		
target_container	16	uimsbf
target_fragment	24	uimsbf
}		

**target\_container:** The container ID of the container holding the encapsulation structure for the target fragment.

**target\_fragment:** A 24 bit identifier which uniquely identifies a fragment within the target container. To locate the actual fragment, the appropriate container should be loaded and the encapsulation structure searched to find a matching `fragment_id`. Once the `fragment_id` has been located the appropriate TVA fragment can be found.

#### 4.8.5.7.2 Referencing a fragment within the same container

It is quite possible to use the above method for referencing fragments within the same container, however it is not the most efficient way. Therefore the following method is supported.

Syntax	No. of Bits	Mnemonic
local_fragment_locator() {		
fragment_offset	16	uimsbf
}		

**fragment\_offset:** The offset within the encapsulation structure, where the fragment can be found. It should be noted that the offset is an index into the encapsulation structure and not a byte offset. The byte offset can be calculated as follows:

```
byte_offset = (sizeof(encapsulation_entry) * fragment_offset) +
encapsulation_header;
```

## 4.8.6 Binary representation of Simple Types

Within the Index list structure it is a requirement to define the encoding used to represent each of an index's key fields. The following is a list of XML Schema defined primitive simple types used within the *TV-Anytime* schema and how they should be encoded, when used within the context of an index.

SimpleType	field_encoding value	Encoding
string	0x0000	String
anyURI	0x0000	String
boolean	0x0204	unsigned byte with false = "0x00" and true = Non zero value e.g. 0xff
NMTOKEN	0x0000	String
gYear	0x0101	unsigned binary - Big-Endian
date	0x0401	Modified Julian Date (TVA BiM codec clause 4.4.2.4.3)
dateTime	0x0400	Modified Julian Date and Milliseconds (TVA BiM codec, clause 4.4.2.4.2)
duration	0x0400	Modified Julian Date and Milliseconds (TVA BiM codec, clause 4.4.2.4.2)
integer	0x0202	one bit to indicate sign (0:positive, 1: negative), followed by abs(value) using vluimsbf5.
unsignedShort	0x0101	unsigned binary - Big-Endian
unsignedLong	0x0203	vluimsbf8
unsignedInt	0x0201	unsigned binary - Big-Endian
nonNegativeInteger	0x0203	vluimsbf8
positiveInteger	0x0203	vluimsbf8
float	0x0300	signed float
double	0x0302	signed double

## 4.8.7 Indexes based on Classification Schemes

When creating an index based on a Classification Scheme it is recommended that the key field used is the `href` attribute within the `ControlledTermType` and that its value is a full de-referenced URI i.e. not an Alias.

It is also recommended that all entries within the index use a single classification scheme. The indexing solution can support the use of mixed classification schemes but it may present problems to an application, which wishes to search the index. For example, there may be more than one reference value for genre "Sport".

## 4.9 Notion of Validation

*TV-Anytime* deals with metadata and in particular with large and highly structured metadata.

An XML Schema document is called a schema. The TVA schema is thus an XML schema document, namely the data model defining the rules to be respected to edit "valid" TVA metadata description. This includes information about default values, element types, attributes types and type hierarchies.

The validation process ensures that the descriptions transmitted to the application respect all the rules defined in the associated schema and thus that it is conformant to the standard having specified this schema. The nature (i.e. the type) of each description item (element or attributes) are also determined and controlled during this process. Validation is the way to make data more explicit to an application. It is a transformation from raw un-typed well-formed (syntactically correct according to the schema definitions) information into typed useful information. The generic nature of the validation process is spelled out by W3C in the XML schema specification in [2].

Because the *TV-Anytime* encoding mechanism produces by nature only valid encoded metadata description and due to the design of the *TV-Anytime* schema and of the associated fragmentation mechanism, each partial description shall be valid according to the TVA schema, first, after the decoding of the TVAMain fragments and, afterwards, of any of its associated TVA fragments.

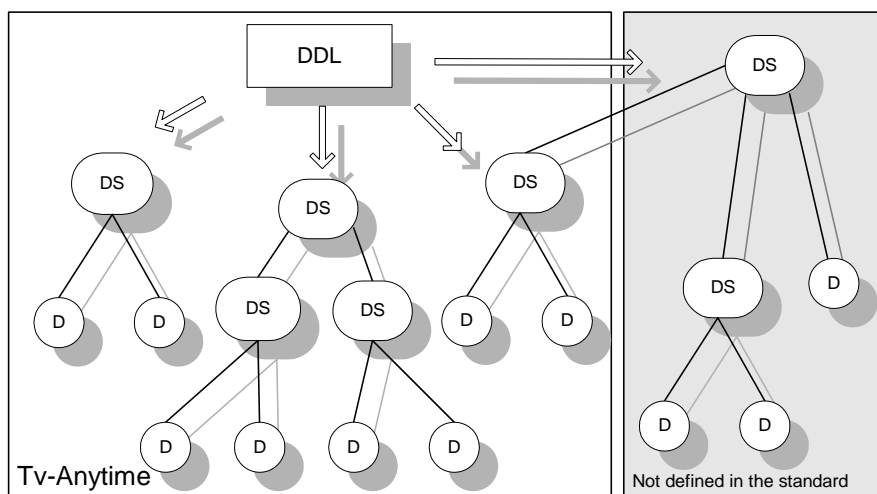
## 4.10 Extensibility of the *TV-Anytime* schema

### 4.10.1 Introduction

A TVA metadata system includes a common core set of metadata as defined in TS 102 822-3-1 [9] to ensure a minimum level of interoperability. Scalability and extensibility are key *TV-Anytime* features. Backward and possibly forward compatibility shall be maintained.

The MPEG-7 Definition Description Language (DDL) that is used as the *TV-Anytime* representation language for metadata is the main instrument for this extensibility.

As shown in figure 33, mechanisms need to be defined to allow the extension of the specification with new *TV-Anytime* definitions, or for private extensions.



**Figure 33: Representation of a TV-Anytime extension**

The associated TVA extensibility rules are presented in the following clause.

#### 4.10.2 Extensibility rules

The following clause defines rules to support specification and private extensions in a backward and forward compatible manner:

- Forward compatibility means a decoder that is only aware of a previous version of a schema is able to partially decode a description conformant to an updated version of that schema.
- Backward compatibility means a decoder that is only aware of a new version of the schema is able to partially decode a description conformant to a previous version of that schema.

With BIM, backward compatibility is provided by the unique reference of the used schema in the `DecoderInit`. Forward compatibility is ensured by a specific syntax defined in MPEG-7 Specification [1] clauses 7 and 8. Its main principle is to use the namespace of the schema. The binary format allows one to keep parts of a description related to different schemas in separate chunks of the binary description stream, so that parts related to an unknown schema may be skipped by the decoder. The Decoder Initialization identifies schema versions with which compatibility is preserved by listing their Schema URIs. A decoder that knows at least one of the Schema URIs will be able to decode at least part of the binary description stream.

The following rules shall be applied so as to define a valid extension of the standard, in particular to allow the compatibility mechanisms described above. They constrain the extensibility of *TV-Anytime* schemas:

- Extension must be defined using the TVA schema representation language (i.e. MPEG-7 DDL). The way these extended schemas are transmitted is out of the scope of the present document.
- The module definition must have a prose definition that describes the syntactic and semantic requirements of the elements, attributes and/or content models that it declares.
- Existing element names should never be re-used. New elements names should be defined under their own namespace (e.g. for another version of the TVA specification or for private extensions).
- The module definition's elements and attributes must be part of an XML namespace. If the module is defined by an organization other than TVA and MPEG (for imported MPEG datatypes and description schemes), this namespace must NOT be the same as the namespace in which other TVA and MPEG standards are defined.
- The namespace under which extensions are defined will need to be clearly identified.
- Any extensions to existing schema should not obscure existing functionality. Thus existing functionality should not be contained within a new element that an earlier decoder will not understand.

- Wherever possible, an extended schema should only add functionality and not replace existing functionality. This will allow a version 1 decoder to maximally understand a version 2 document.
- An application should ignore any elements or attributes they do not need, do not understand or cannot use.

Table 13 provides the list of conditions under which the extensions of *TV-Anytime* metadata definitions are supported or not.

**Table 13: Types of extension permitted in future versions of *TV-Anytime***

Condition/Type of extension of <i>TV-Anytime</i> metadata definitions	Status
Condition 1: A new global element of existing type	NOT PERMITTED
Condition 2: A new global attributes added to existing type	PERMITTED
new type (simple or complex - but see below for limitations on derivation etc)	PERMITTED
Polymorphism of existing type by Inheritance with restriction	PERMITTED (But see rules above)
Polymorphism of existing type by Inheritance with extension	PERMITTED (But see rules above)
Polymorphism of existing type by Redefining types during import	NOT PERMITTED
Substitution Groups	NOT PERMITTED. Instead of using substitution groups, explicit derivation can be used. This is safer for future extensions.

---

## Annex A (informative): Bibliography

- XML Path Language, W3C Recommendation, 16 November 1999.

NOTE: Available at: <http://www.w3.org/TR/xpath.html>.

- Namespaces in XML, W3C Recommendation, 14 January 1999.

NOTE 1: Available at: <http://www.w3.org/TR/REC-xml-names/>.

NOTE 2: These documents are maintained by the W3C (<http://www.w3.org>)

- IETF RFC 3986: "Uniform Resource Identifier (URI): Generic Syntax".
- XML, Extensible Markup Language (XML) 1.0 second edition, October 2000.

NOTE: Available at: <http://www.w3.org/TR/2000/REC-xml-20001006>.

- IETF RFC 3629: "UTF-8, a transformation format of ISO/IEC 10646 (2003)".

NOTE: Available at <http://www.ietf.org/rfc/rfc3629.txt>.

---

## List of figures

Figure 1: Overview of the scope of TVA Specification on Metadata and System.....	8
Figure 2: Unidirectional environment .....	12
Figure 3: Processes associated with delivery of metadata.....	13
Figure 4: Functional metadata processing architecture in a unidirectional environment .....	15
Figure 5: Fragmentation of a <i>TV-Anytime</i> metadata description .....	16
Figure 6: UML-like representation of a TVAMain fragment.....	17
Figure 7: UML-like representation of a MetadataOriginationInformation fragment .....	19
Figure 8: UML-like representation of a ProgramInformation fragment.....	19
Figure 9: UML-like representation of a GroupInformation fragment .....	20
Figure 10: UML-like representation of a BroadcastEvent fragment .....	21
Figure 11: UML-like representation of a Schedule fragment.....	22
Figure 12: UML-like representation of a ServiceInformation fragment .....	23
Figure 13: UML-like representation of PersonNameType .....	23
Figure 14: UML-like representation of a PurchaseInformation fragment .....	24
Figure 15: UML-like representation of a ProgramReviews fragment .....	24
Figure 16: UML-like representation of a ClassificationScheme fragment .....	25
Figure 17: UML-like representation of a SegmentInformation fragment .....	26
Figure 18: UML-like representation of a SegmentGroup fragment .....	26
Figure 19: UML-like representation of a Package fragment .....	27
Figure 20: UML - like representation of a InterstitialCampaign fragment.....	27
Figure 21: UML - like representation of a RMPIDescription fragment .....	27
Figure 22: UML - like representation of a CouponDescription fragment.....	28
Figure 23: UML - like representation of a TargetInformation fragment .....	28
Figure 24: UML - like representation of an InterstitialBreak fragment.....	29
Figure 25: UML - like representation of a Rule fragment.....	29
Figure 26: UML - like representation of the ContentListType used by the following fragments .....	29
Figure 27: BiM bitstream without Zlib optimized decoder .....	38
Figure 28: BiM stream with Zlib optimized decoder .....	39
Figure 29: Schematic representation of interrelationship between structures within a container.....	43
Figure 30: Schematic representation of interrelationship between Index containers and Data containers.....	48
Figure 31: Indexing structure .....	50
Figure 32: Example multi_field_sub_index structure (using multi-layer syntax) for an index with three key fields (Genre, Language and Title) .....	61

Figure 33: Representation of a *TV-Anytime* extension ..... 67

---

## List of tables

Table 1: Table of values for the EncodingVersion parameter .....	32
Table 2: Character encoding and their termination values .....	32
Table 3: Table of values for the IndexingVersion field.....	33
Table 4: Structure_type assignments.....	41
Table 5: Structure_type and their matching valid structure_id .....	42
Table 6: Valid fragment_reference_formats.....	44
Table 7: Fragment_type assignments .....	54
Table 8: field_identifier Assignments .....	55
Table 9: Encoding and interpretation of the field_value, low_field_value and high_field_value field.....	55
Table 10: Encoding types and their respective sizes .....	56
Table 11: Defined index order for primitive simple types .....	57
Table 12: Fragment reference types .....	58
Table 13: Types of extension permitted in future versions of <i>TV-Anytime</i> .....	68



---

## History

<b>Document history</b>		
V1.1.1	October 2003	Publication
V1.2.1	September 2004	Publication
V1.3.1	January 2006	Publication
V1.4.1	November 2007	Publication