

# **SDMX STANDARDS: SECTION 6**

## **TECHNICAL NOTES**

**Version 3.1**

**May 2025**

## Revision History

Revision	Date	Contents
DRAFT 1.0	December 2024	Draft release updated for SDMX 3.1 for public consultation
1.0	May 2025	Public release for SDMX 3.1

## Contents

<b>1</b>	<b>Purpose and Structure</b> .....	<b>6</b>
1.1	Purpose .....	6
1.2	Structure .....	6
<b>2</b>	<b>General Notes on This Document</b> .....	<b>7</b>
<b>3</b>	<b>Guide for SDMX Format Standards</b> .....	<b>8</b>
3.1	Introduction .....	8
3.2	SDMX Information Model for Format Implementers .....	8
3.2.1	Introduction .....	8
3.3	SDMX Formats: Expressive Capabilities and Function .....	8
3.3.1	Format Optimizations and Differences .....	8
3.4	SDMX Best Practices .....	10
3.4.1	Reporting and Dissemination Guidelines .....	10
3.4.2	Best Practices for Batch Data Exchange .....	13
<b>4</b>	<b>General Notes for Implementers</b> .....	<b>15</b>
4.1	Representations .....	15
4.1.1	Data Types .....	17
4.2	Time and Time Format .....	18
4.2.1	Introduction .....	18
4.2.2	Observational Time Period .....	18
4.2.3	Standard Time Period .....	18
4.2.4	Gregorian Time Period .....	19
4.2.5	Date Time .....	19
4.2.6	Standard Reporting Period .....	19
4.2.7	Distinct Range .....	22
4.2.8	Time Format .....	23
4.2.9	Time Zones .....	23
4.2.10	Representing Time Spans Elsewhere .....	24
4.2.11	Notes on Formats .....	24
4.2.12	Effect on Time Ranges .....	24
4.2.13	Time in Query Messages .....	24
4.3	Versioning .....	26
4.3.1	Non-versioned artefacts .....	26
4.3.2	Semantically versioned artefacts .....	27
4.3.3	Legacy-versioned artefacts .....	28
4.3.4	Dependency management and references .....	28
4.4	Structural Metadata Querying Best Practices .....	29
<b>5</b>	<b>Reference Metadata</b> .....	<b>30</b>
5.1	Scope of the Metadata Structure Definition (MSD) .....	30
5.2	Identification of the Object(s) to which the Metadata is to be attached .....	30
5.3	Metadata Structure Definition .....	31
5.4	Metadata Set .....	32
5.5	Reference Metadata in Data Structure Definition and Dataset .....	33
<b>6</b>	<b>Codelist</b> .....	<b>34</b>

6.1	Codelist extension and discriminated unions.....	34
6.1.1	Prefixing Code Ids .....	35
6.1.2	Including / Excluding Specific Codes.....	35
6.1.3	Parent Ids .....	36
6.1.4	Discriminated Unions.....	38
6.2	Linking Hierarchies .....	39
<b>7</b>	<b>Geospatial information support .....</b>	<b>41</b>
7.1	Indirect Reference to Geospatial Information. ....	41
7.2	Geographic Coordinates.....	42
7.3	A Geographic Codelist.....	45
<b>8</b>	<b>Maintenance Agencies and Metadata Providers .....</b>	<b>48</b>
<b>9</b>	<b>Concept Roles.....</b>	<b>51</b>
9.1	Overview .....	51
9.2	Information Model .....	51
9.3	Technical Mechanism.....	51
9.4	SDMX-ML Examples in a DSD .....	52
9.5	SDMX standard roles Concept Scheme .....	53
<b>10</b>	<b>Constraints.....</b>	<b>55</b>
10.1	Introduction .....	55
10.2	Availability Constraint.....	55
10.3	Dimension Constraint.....	55
10.4	Reporting Constraints .....	57
10.4.1	Data Constraint.....	57
10.4.2	Metadata Constraint .....	57
10.4.3	Scope of a Constraint.....	58
10.4.4	Multiple Constraints .....	58
10.4.5	Versioning.....	59
10.4.6	Inheritance.....	60
10.4.7	Constraints Examples.....	62
<b>11</b>	<b>Transforming between versions of SDMX.....</b>	<b>69</b>
11.1	Scope .....	69
11.2	Compatibility and new DSD features .....	69
<b>12</b>	<b>Validation and Transformation Language (VTL).....</b>	<b>70</b>
12.1	Introduction .....	70
12.2	References to SDMX artefacts from VTL statements .....	71
12.2.1	Introduction.....	71
12.2.2	References through the URN .....	71
12.2.3	Abbreviation of the URN.....	73
12.2.4	User-defined alias.....	76
12.2.5	References to SDMX artefacts from VTL Rulesets .....	76
12.3	Mapping between SDMX and VTL artefacts.....	77
12.3.1	When the mapping occurs.....	77
12.3.2	General mapping of VTL and SDMX data structures .....	78
12.3.3	Mapping from SDMX to VTL data structures.....	78
12.3.4	Mapping from VTL to SDMX data structures.....	81

12.3.5	Declaration of the mapping methods between data structures .....	84
12.3.6	Mapping dataflow subsets to distinct VTL Data Sets .....	85
12.3.7	Mapping variables and value domains between VTL and SDMX .....	90
12.4	Mapping between SDMX and VTL Data Types .....	92
12.4.1	VTL Data types.....	92
12.4.2	VTL basic scalar types and SDMX data types .....	94
12.4.3	Mapping SDMX data types to VTL basic scalar types.....	95
12.4.4	Mapping VTL basic scalar types to SDMX data types.....	97
12.4.5	Null Values .....	99
12.4.6	Format of the literals used in VTL Transformations.....	100
<b>13</b>	<b>Structure Mapping.....</b>	<b>101</b>
13.1	Introduction .....	101
13.2	1-1 structure maps .....	101
13.3	N-n structure maps.....	102
13.4	Ambiguous mapping rules.....	103
13.5	Representation maps .....	103
13.6	Regular expression and substring rules.....	104
13.6.1	Regular expressions.....	105
13.6.2	Substrings.....	105
13.7	Mapping non-SDMX time formats to SDMX formats.....	106
13.7.1	Pattern based dates .....	107
13.7.2	Numerical based datetime.....	110
13.7.3	Mapping more complex time inputs.....	111
13.8	Using TIME_PERIOD in mapping rules .....	111
13.9	Time span mapping rules using validity periods .....	111
13.10	Mapping examples.....	112
13.10.1	Many to one mapping (N-1) .....	112
13.10.2	Mapping other data types to Code Id.....	112
13.10.3	Observation Attributes for Time Period.....	113
13.10.4	Time mapping .....	113
<b>14</b>	<b>ANNEX Semantic Versioning.....</b>	<b>115</b>
14.1	Introduction to Semantic Versioning .....	115
14.2	Semantic Versioning Specification for SDMX 3.0(.0).....	115
14.3	Backus–Naur Form Grammar for Valid SDMX 3.0(.0) Semantic Versions 117	
14.4	Dependency Management in SDMX 3.0(.0): .....	118
14.5	Upgrade and conversions of artefacts defined with previous SDMX standard versions to Semantic Versioning .....	119
14.6	FAQ for Semantic Versioning.....	120

# 1 **1 Purpose and Structure**

## 2 **1.1 Purpose**

3 The intention of Section 6 is to document certain aspects of SDMX that are important  
4 to understand and will aid implementation decisions. The explanations here  
5 supplement the information documented in the SDMX XML/JSON schemas and the  
6 Information Model.

## 7 **1.2 Structure**

8 This document is organized into the following major parts:  
9

- 10 • A guide to the SDMX Information Model relating to Data Structure Definitions and  
11 Data Sets, statement of differences in functionality supported by the different  
12 formats and syntaxes for Data Structure Definitions and Data Sets, and best  
13 practices for use of SDMX formats, including the representation for time period.
- 14 • A guide to the SDMX Information Model relating to Metadata Structure  
15 Definitions, and Metadata Sets.
- 16 • Other structural artefacts of interest: Agencies, Concept Role, Constraint,  
17 Codelist.

## 18 **2 General Notes on This Document**

19 As of version SDMX 2.1, the term "Key family" has been replaced by Data Structure  
20 Definition (also known and referred to as DSD) both in the XML schemas and the  
21 Information Model. The term "Key family" is not familiar to many people and its name  
22 was taken from the model of SDMX-EDI (previously known as GESMES/TS). The  
23 more familiar name "Data Structure Definition" which was used in many documents is  
24 now also the technical artefact in the SDMX-ML and Information Model technical  
25 specifications. The SDMX-EDI specification, that was using the term "Key family", is  
26 deprecated in this version of the specification.

27

28 There has been much work within the SDMX community on the creation of user guides,  
29 tutorials, and other aids to implementation and understanding of the standard. This  
30 document is not intended to duplicate the function of these documents, but instead  
31 represents a short set of technical notes not generally covered elsewhere.

32

## 33 **3 Guide for SDMX Format Standards**

### 34 **3.1 Introduction**

35 This guide exists to provide information to implementers of the SDMX format standards  
36 – SDMX-ML, SDMX-JSON and SDMX-CSV – that are concerned with data, i.e., Data  
37 Structure Definitions and Data Sets. This section is intended to provide information that  
38 will help users of SDMX understand and implement the standards. It is not normative,  
39 and it does not provide any rules for the use of the standards, such as those found in  
40 *SDMX-ML: Schema and Documentation*.

41

### 42 **3.2 SDMX Information Model for Format Implementers**

#### 43 **3.2.1 Introduction**

44 The purpose of this sub-section is to provide an introduction to the SDMX-IM relating  
45 to Data Structure Definitions and Data Sets for those whose primary interest is in the  
46 use of the XML, JSON or CSV formats. For those wishing to have a deeper  
47 understanding of the Information Model, the full SDMX-IM document, and other  
48 sections in this guide provide a more in-depth view, along with UML diagrams and  
49 supporting explanation. For those who are unfamiliar with DSDs, an appendix to the  
50 SDMX-IM provides a tutorial which may serve as a useful introduction.

51

52 The SDMX-IM is used to describe the basic data and metadata structures used in all  
53 of the SDMX data formats. The Information Model concerns itself with statistical data  
54 and its structural metadata, and that is what is described here. Both structural  
55 metadata and data have some additional metadata in common, related to their  
56 management and administration. These aspects of the data model are not addressed  
57 in this section and covered elsewhere in this guide or in the full SDMX-IM document.

58

59 Note that in the descriptions below, text in courier and italics are the names used in  
60 the information model (e.g., *DataSet*).

### 61 **3.3 SDMX Formats: Expressive Capabilities and Function**

62 SDMX offers several equivalent formats for describing data and structural metadata,  
63 optimized for use in different applications. Although all of these formats are derived  
64 directly from the SDMX-IM, and are thus equivalent, the syntaxes used to express the  
65 model place some restrictions on their use. Also, different optimizations provide  
66 different capabilities. This section describes these differences and provides some rules  
67 for applications which may need to support more than one SDMX format or syntax.  
68 This section is constrained to the Data Structure Definition and the DataSet.

#### 69 **3.3.1 Format Optimizations and Differences**

70 The following section provides a brief overview of the differences between the various  
71 SDMX formats.

72

73 Version 2.0 was characterised by 4 data messages, each with a distinct format:  
74 Generic, Compact, Cross-Sectional and Utility. Because of the design, data in some  
75 formats could not always be related to another format. In version 2.1, this issue has  
76 been addressed by merging some formats and eliminating others. As a result, in SDMX  
77 2.1 there were just two types of data formats: *GenericData* and *StructureSpecificData*

78 (i.e., specific to one Data Structure Definition). As of SDMX 3.0, based also on the real-  
79 life usage of 2.1 XML formats but also the new formats introduced (JSON and CSV),  
80 only one XML format remains, i.e., *StructureSpecificData*. Furthermore, the time  
81 specific sub-formats have also been deprecated due to the lack of usage.

82  
83 SDMX-JSON and SDMX-CSV feature also only one flavour, each. It should be noted,  
84 though, that both XML and JSON messages allow for series oriented as well as flat  
85 representations.

86

### 87 **Structure Definition**

88 • The SDMX-ML Structure Message is currently the main way of modelling a DSD.  
89 The SDMX-JSON version follows the same principles, while the SDMX-CSV does  
90 not support structures, yet.

91 • The SDMX-ML Structure Message allows for the structures on which a Data  
92 Structure Definition depends – that is, codelists and concepts – to be either  
93 included in the message or to be referenced by the message containing the data  
94 structure definition. XML syntax is designed to leverage URIs and other Internet-  
95 based referencing mechanisms, and these are used in the SDMX-ML message.  
96 This option is also available in SDMX-JSON. The latter, though, further supports  
97 conveying data with some structural metadata within a single message.

98 • All structures can be inserted, replaced or deleted, unless structural  
99 dependencies are not respected.

### 100 **Validation**

101 • The SDMX-ML structure specific messages will allow validation of XML syntax  
102 and data typing to be performed with a generic XML parser and enforce  
103 agreement between the structural definition and the data to a moderate degree  
104 with the same tool.

105 • Similarly, the SDMX-JSON message can be validated using JSON Schema and  
106 hence may also be generically parsed and validated.

107 • The SDMX-CSV format cannot be validated by generic tools.

### 108 **Update and Delete Messages**

109 • All data messages allow for both append/replace/delete messages.

110 • These messages allow also transmitting only data or only documentation (i.e.,  
111 Attribute values without Observation values).

### 112 **Character Encodings**

113 All formats use the UTF-8 encoding. The SDMX-CSV may use a different encoding if  
114 this is reported properly in the mime type of a web service response.

115

### 116 **Data Typing**

117 The XML syntax and JSON syntax have similar data-typing mechanisms. Hence, there  
118 is no need for conventions in order to allow transition from one format to another, like  
119 those required for EDIFACT in SDMX 2.1. On the other hand, JSON schema has a  
120 simpler set of data types (as explained in section 2, paragraph “3.6.3.3 Representation  
121 Constructs”) but complements its data types with a fixed set of formats or regular  
122 expressions. In addition, the JSON schema has also types that are not natively  
123 supported in XML schema and need to be implemented as complex types in the latter.

124 The section below provides examples of those cases that are not natively supported  
125 by either the XML or JSON data types. More details on the data mapping between  
126 XML and JSON schemas are also explained in section "4.1.1 Data Types".  
127

## 128 **3.4 SDMX Best Practices**

### 129 **3.4.1 Reporting and Dissemination Guidelines**

#### 130 **3.4.1.1 Central Institutions and Their Role in Statistical Data Exchanges**

131 Central institutions are the organisations to which other partner institutions "report"  
132 statistics. These statistics are used by central institutions either to compile aggregates  
133 and/or they are put together and made available in a uniform manner (e.g., on-line or  
134 on a CD-ROM or through file transfers). Therefore, central institutions receive data  
135 from other institutions and, usually, they also "disseminate" data to individual and/or  
136 institutions for end-use. Within a country, a NSI or a national central bank (NCB) plays,  
137 of course, a central institution role as it collects data from other entities and it  
138 disseminates statistical information to end users. In SDMX the role of central institution  
139 is very important: every statistical message is based on underlying structural definitions  
140 (statistical concepts, code lists, DSDs) which have been devised by a particular  
141 agency, usually a central institution. Such an institution plays the role of the reference  
142 "structural definitions maintenance agency" for the corresponding messages which are  
143 exchanged. Of course, two institutions could exchange data using/referring to  
144 structural information devised by a third institution.

145  
146 Central institutions can play a double role:

- 147 • collecting and further disseminating statistics;
- 148 • devising structural definitions for use in data exchanges.

#### 149 **3.4.1.2 Defining Data Structure Definitions (DSDs)**

150 The following guidelines are suggested for building a DSD. However, it is expected  
151 that these guidelines will be considered by central institutions when devising new  
152 DSDs.

##### 153 Dimensions, Attributes and Codelists

- 154  
155  
156 • **Avoid dimensions that are not appropriate for all the series in the data  
157 structure definition.** If some dimensions are not applicable (this is evident from  
158 the need to have a code in a code list which is marked as "not applicable", "not  
159 relevant" or "total") for some series then consider moving these series to a new  
160 data structure definition in which these dimensions are dropped from the key  
161 structure. This is a judgement call as it is sometimes difficult to achieve this  
162 without increasing considerably the number of DSDs.
- 163 • **Devise DSDs with a small number of Dimensions for public viewing of data.**  
164 A DSD with the number dimensions in excess 6 or 7 is often difficult for non-  
165 specialist users to understand. In these cases, it is better to have a larger number  
166 of DSDs with smaller "cubes" of data, or to eliminate dimensions and aggregate  
167 the data at a higher level. Dissemination of data on the web is a growing use case  
168 for the SDMX standards: the differentiation of observations by dimensionality,

169 which are necessary for statisticians and economists, are often obscure to public  
170 consumers who may not always understand the semantic of the differentiation.

171 • **Avoid composite dimensions.** Each dimension should correspond to a single  
172 characteristic of the data, not to a combination of characteristics.

173 • Consider the inclusion of the following attributes. Once the key structure of a data  
174 structure definition has been decided, then the set of (preferably mandatory)  
175 attributes of this data structure definition has to be defined. In general, some  
176 statistical concepts are deemed necessary across all Data Structure Definitions  
177 to qualify the contained information. Examples of these are:

178 ○ A descriptive title for the series (this is most useful for dissemination of data for  
179 viewing e.g., on the web).

180 ○ Collection (e.g., end of period, averaged or summed over period).

181 ○ Unit (e.g., currency of denomination).

182 ○ Unit multiplier (e.g., expressed in millions).

183 ○ Availability (which institutions can a series become available to).

184 ○ Decimals (i.e., number of decimal digits used in numerical observations).

185 ○ Observation Status (e.g., estimate, provisional, normal).

186

187 Moreover, additional attributes may be considered as mandatory when a specific data  
188 structure definition is defined.

189

190 • **Avoid creating a new code list where one already exists.** It is highly  
191 recommended that structural definitions and code lists be consistent with  
192 internationally agreed standard methodologies, wherever they exist, e.g., System  
193 of National Accounts 1993; Balance of Payments Manual, Fifth Edition; Monetary  
194 and Financial Statistics Manual; Government Finance Statistics Manual, etc.  
195 When setting-up a new data exchange, the following order of priority is suggested  
196 when considering the use of code lists:

197 ○ international standard code lists;

198 ○ international code lists supplemented by other international and/or regional  
199 institutions;

200 ○ standardised lists used already by international institutions;

201 ○ new code lists agreed between two international or regional institutions;

202 ○ new code lists which extend existing code lists, by adding only missing codes;

203 ○ new specific code lists.

204

205 The same code list can be used for several statistical concepts, within a data structure  
206 definition or across DSDs. Note that SDMX has recognised that these classifications  
207 are often quite large and the usage of codes in any one DSD is only a small extract of  
208 the full code list. In this version of the standard, it is possible to exchange and  
209 disseminate a **partial code list** which is extracted from the full code list and which  
210 supports the dimension values valid for a particular DSD.

211

## 212 Data Structure Definition Structure

- 213 • The following items have to be specified by a structural definitions maintenance  
214 agency when defining a new data structure definition:
- 215 • Data structure definition (DSD) identification:
- 216 • DSD identifier
- 217 • DSD name
- 218 • A list of metadata concepts assigned as dimensions of the data structure  
219 definition. For each:
- 220 • (statistical) concept identifier
- 221 • code list identifier (id, version, maintenance agency) if the  
222 representation is coded
- 223 • A list of (statistical) concepts assigned as attributes for the data structure  
224 definition. For each:
- 225 • (statistical) concept identifier
- 226 • code list identifier if the concept is coded
- 227 • usage: mandatory, optional
- 228 • relationship to dimensions and measures
- 229 • maximum text length for the uncoded concepts
- 230 • maximum code length for the coded concepts
- 231 • A list of the code lists used in the data structure definition. For each:
- 232 • code list identifier
- 233 • code list name
- 234 • code values and descriptions
- 235 • Definition of Dataflow. Two (or more) partners performing data exchanges in a  
236 certain context need to agree on:
- 237 • the list of dataset identifiers they will be using;
- 238 • for each Dataflow:
- 239 • its content (e.g., by Constraints) and description
- 240 • the relevant DSD that defines the structure of the data reported or  
241 disseminated according the Dataflow

### 242 3.4.1.3 Exchanging Attributes

#### 243 3.4.1.3.1 *Attributes on series and group levels*

- 244 • Static properties.
- 245 • Upon creation of a series the sender has to provide to the receiver values for all  
246 mandatory attributes. In case they are available, values for conditional attributes  
247 should also be provided. Whereas initially this information may be provided by  
248 means other than SDMX-ML/JSON/CSV messages (e.g., paper, telephone) it is

- 249 expected that partner institutions will be in a position to provide this information in  
250 the available formats over time.
- 251 • A centre may agree with its data exchange partners special procedures for  
252 authorising the setting of attributes' initial values.
  - 253 • Communication of changes to the centre.
  - 254 • Following the creation of a series, the attribute values do not have to be reported  
255 again by senders, as long as they do not change.
  - 256 • Whenever changes in attribute values for a series (or group) occur, the reporting  
257 institutions should report either all attribute values again (this is the recommended  
258 option) or only the attribute values which have changed. This applies both to the  
259 mandatory and the conditional attributes. For example, if a previously reported  
260 value for a conditional attribute is no longer valid, this has to be reported to the  
261 centre.
  - 262 • A centre may agree with its data exchange partners special procedures for  
263 authorising modifications in the attribute values.
  - 264 • Communication of observation level attributes "observation status", "observation  
265 confidentiality", "observation pre-break" is recommended.
  - 266 • Whenever an observation is exchanged, the corresponding observation status is  
267 recommended to also be exchanged attached to the observation, regardless of  
268 whether it has changed or not since the previous data exchange.
  - 269 • If the "observation status" changes and the observation remains unchanged, both  
270 components would have to be reported (unless the observation is deleted).
- 271 For Data Structure Definitions having also the observation level attributes  
272 "observation confidentiality" and "observation pre-break" defined, this rule  
273 applies to these attributes as well: if an institution receives from another  
274 institution an observation with an observation status attribute only attached, this  
275 means that the associated observation confidentiality and pre-break  
276 observation attributes either never existed or from now they do not have a value  
277 for this observation.

## 278 **3.4.2 Best Practices for Batch Data Exchange**

### 279 **3.4.2.1 Introduction**

280 Batch data exchange is the exchange and maintenance of entire databases between  
281 counterparties. It is an activity that often employs SDMX-CSV format, and might also  
282 use the SDMX-ML dataset. The following points apply equally to both formats.

### 283 **3.4.2.2 Positioning of the Dimension "Frequency"**

284 In SDMX 3.0, the "frequency" dimension is not special in the data structure definition.  
285 Many central institutions devising structural definitions have decided to assign to this  
286 dimension the first position in the key structure. Nevertheless, a standard role (i.e., that  
287 of 'Frequency') may facilitate the easy identification of this dimension. This is  
288 necessary to frequency's crucial role in several database systems and in attaching  
289 attributes at the "sibling" group level.

### 290 3.4.2.3 Identification of Data Structure Definitions (DSDs)

291 In order to facilitate the easy and immediate recognition of the structural definition  
292 maintenance agency that defined a data structure definition, some central institutions  
293 devising structural definitions use the first characters of the data structure definition  
294 identifiers to identify their institution: e.g., BIS\_EER, EUROSTAT\_BOP\_01,  
295 ECB\_BOP1, etc. Nevertheless, using the AgencyId may disambiguate any Artefact in  
296 a more efficient and machine readable way.

### 297 3.4.2.4 Identification of the Dataflows

298 In order to facilitate the easy and immediate recognition of the institution administrating  
299 a Dataflow, some central institutions prefer to use the first characters of the Dataflow  
300 identifiers to identify their institution: e.g. BIS\_EER, ECB\_BOP1, ECB\_BOP1, etc.  
301 Nevertheless, using the AgencyId may disambiguate any Artefact in a more efficient  
302 and machine readable way.

303  
304 The statistical information in SDMX is broken down into two fundamental parts –  
305 structural metadata (comprising the `DataStructureDefinition`, and associated  
306 `Concepts` and `CodeLists`) – see `Framework for Standards` – and observational data  
307 (the `DataSet`). This is an important distinction, with specific terminology associated  
308 with each part. Data, which is typically a set of numeric observations at specific points  
309 in time, is organised into data sets (`DataSet`). These data sets are structured  
310 according to a specific `DataStructureDefinition` and are described in the  
311 `Dataflow` (via `Constraints`). The `DataStructureDefinition` describes the  
312 metadata that allows an understanding of what is expressed in the `DataSet`, whilst  
313 the `Dataflow` provides the identifier and other important information (such as the  
314 periodicity of reporting) that is common to all its `Components`.

315  
316 Note that the role of the `Dataflow` and `DataSet` is very specific in the model, and the  
317 terminology used may not be the same as used in all organisations, and specifically  
318 the term `DataSet` is used differently in SDMX than in GESMES/TS. Essentially the  
319 GESMES/TS term "Data Set" is, in SDMX, the "Dataflow" whilst the term "Data Set" in  
320 SDMX is used to describe the "container" for an instance of the data.

### 321 3.4.2.5 Special Issues

#### 322 3.4.2.5.1 "Frequency" related issues

- 323 • **Special frequencies.** The issue of data collected at special (regular or irregular)  
324 intervals at a lower than daily frequency (e.g., 24 or 36 or 48 observations per  
325 year, on irregular days during the year) is not extensively discussed here.  
326 However, for data exchange purposes:
- 327 • such data can be mapped into a series with daily frequency; this daily series  
328 will only hold observations for those days on which the measured event takes  
329 place;
  - 330 • if the collection intervals are regular, additional values to the existing  
331 frequency code list(s) could be added in the future.
- 332 • **Tick data.** The issue of data collected at irregular intervals at a higher than daily  
333 frequency (e.g., tick-by-tick data) is not discussed here either.

334 **4 General Notes for Implementers**

335 This section discusses a number of topics other than the exchange of data sets in  
 336 SDMX formats. Supported only in SDMX-ML (and some in SDMX-JSON), these topics  
 337 include the use of the reference metadata mechanism in SDMX, the use of Structure  
 338 Sets and Reporting Taxonomies, the use of Processes, a discussion of time and data-  
 339 typing, and the conventional mechanisms within the SDMX-ML Structure message  
 340 regarding versioning and referencing.

341 **4.1 Representations**

342 This section does not go into great detail on these topics but provides a useful overview  
 343 of these features to assist implementors in further use of the parts of the specification  
 344 which are relevant to them.

345 There are several different representations in SDMX-ML, taken from XML Schemas  
 346 and common programming languages. The table below describes the various  
 347 representations, which are found in SDMX-ML, and their equivalents.  
 348  
 349

SDMX-ML Data Type	XML Schema Data Type	.NET Framework Type	Java Data Type
String	xsd:string	System.String	java.lang.String
Big Integer	xsd:integer	System.Decimal	java.math.BigInteger
Integer	xsd:int	System.Int32	int
Long	xsd:long	System.Int64	long
Short	xsd:short	System.Int16	short
Decimal	xsd:decimal	System.Decimal	java.math.BigDecimal
Float	xsd:float	System.Single	float
Double	xsd:double	System.Double	double
Boolean	xsd:boolean	System.Boolean	boolean
URI	xsd:anyURI	System.Uri	Java.net.URI or java.lang.String
DateTime	xsd:dateTime	System.DateTime	javax.xml.datatype.XMLGregorianCalendar
Time	xsd:time	System.DateTime	javax.xml.datatype.XMLGregorianCalendar
GregorianYear	xsd:gYear	System.DateTime	javax.xml.datatype.XMLGregorianCalendar
GregorianMonth	xsd:gYearMonth	System.DateTime	javax.xml.datatype.XMLGregorianCalendar
GregorianDay	xsd:date	System.DateTime	javax.xml.datatype.XMLGregorianCalendar
Day, MonthDay, Month	xsd:g*	System.DateTime	javax.xml.datatype.XMLGregorianCalendar
Duration	xsd:duration	System.TimeSpan	javax.xml.datatype.Duration

350 There are also a number of SDMX-ML data types which do not have these direct  
 351 correspondences, often because they are composite representations or restrictions of  
 352 a broader data type. For most of these, there are simple types which can be referenced  
 353 from the SDMX schemas, for others a derived simple type will be necessary:  
 354

- 355
- 356 • `AlphaNumeric` (`common:AlphaNumericType`, string which only allows A-z and 0-9)
- 357
- 358 • `Alpha` (`common:AlphaType`, string which only allows A-z)
- 359 • `Numeric` (`common:NumericType`, string which only allows 0-9, but is not numeric so that it can have leading zeros)
- 360
- 361 • `Count` (`xs:integer`, a sequence with an interval of "1")
- 362 • `InclusiveValueRange` (`xs:decimal` with the `minValue` and `maxValue` facets supplying the bounds)
- 363
- 364 • `ExclusiveValueRange` (`xs:decimal` with the `minValue` and `maxValue` facets supplying the bounds)
- 365
- 366 • `Incremental` (`xs:decimal` with a specified `interval`; the interval is typically enforced outside of the XML validation)
- 367
- 368 • `TimeRange` (`common:TimeRangeType`, `startDateTime` + `Duration`)
- 369 • `ObservationalTimePeriod` (`common:ObservationalTimePeriodType`, a union of `StandardTimePeriod` and `TimeRange`).
- 370
- 371 • `StandardTimePeriod` (`common:StandardTimePeriodType`, a union of `BasicTimePeriod` and `ReportingTimePeriod`).
- 372
- 373 • `BasicTimePeriod` (`common:BasicTimePeriodType`, a union of `GregorianTimePeriod` and `DateTime`)
- 374
- 375 • `GregorianTimePeriod` (`common:GregorianTimePeriodType`, a union of `GregorianYear`, `GregorianMonth`, and `GregorianDay`)
- 376
- 377 • `ReportingTimePeriod` (`common:ReportingTimePeriodType`, a union of `ReportingYear`, `ReportingSemester`, `ReportingTrimester`, `ReportingQuarter`, `ReportingMonth`, `ReportingWeek`, and `ReportingDay`).
- 378
- 379 • `ReportingYear` (`common:ReportingYearType`)
- 380
- 381 • `ReportingSemester` (`common:ReportingSemesterType`)
- 382
- 383 • `ReportingTrimester` (`common:ReportingTrimesterType`)
- 384
- 385 • `ReportingQuarter` (`common:ReportingQuarterType`)
- 386
- 387 • `ReportingMonth` (`common:ReportingMonthType`)
- 388
- 389 • `ReportingWeek` (`common:ReportingWeekType`)
- 390
- 391 • `ReportingDay` (`common:ReportingDayType`)
- 392
- 393 • `XHTML` (`common:StructuredText`, allows for multi-lingual text content that has XHTML markup)
- 394
- 395 • `KeyValues` (`common:DataKeyType`)
- 396
- 397 • `IdentifiableReference` (types for each `IdentifiableObject`)
- 398
- 399 • `GeospatialInformation` (a geo feature set, according to the pattern in section 7.2)
- 400
- 401
- 402
- 403

Data types also have a set of facets:

- 396 • `isSequence` = `true` | `false` (indicates a sequentially increasing value)
- 397 • `minLength` = `positive integer` (# of characters/digits)
- 398 • `maxLength` = `positive integer` (# of characters/digits)
- 399 • `startValue` = `decimal` (for numeric sequence)
- 400 • `endValue` = `decimal` (for numeric sequence)
- 401 • `interval` = `decimal` (for numeric sequence)
- 402 • `timeInterval` = `duration`
- 403 • `startTime` = `BasicTimePeriod` (for time range)

- 404 • `endTime` = `BasicTimePeriod` (for time range)
- 405 • `minValue` = `decimal` (for numeric range)
- 406 • `maxValue` = `decimal` (for numeric range)
- 407 • `decimal` = `Integer` (# of digits to right of decimal point)
- 408 • `pattern` = (a regular expression, as per W3C XML Schema)
- 409 • `isMultiLingual` = `boolean` (for specifying text can occur in more than one language)

411  
412 Note that code lists may also have textual representations assigned to them, in addition  
413 to their enumeration of codes.

#### 414 4.1.1 Data Types

415 XML and JSON schemas support a variety of data types that, although rich, are not  
416 mapped one-to-one in all cases. This section provides an explanation of the mapping  
417 performed in SDMX 3.0, between such cases.

418  
419 For identifiers, text fields and Codes there are no restriction from either side, since a  
420 generic type (e.g., that of string) accompanied by the proper regular expression works  
421 equally well for both XML and JSON.

422  
423 For example, for the `id` type, this is the XML schema definition:

```
424 <xs:simpleType name="IDType">
425   <xs:restriction base="NestedIDType">
426     <xs:pattern value="[A-Za-z0-9_@$\-]+" />
427   </xs:restriction>
428 </xs:simpleType>
```

429 Where the `NestedIDType` is also a restriction of `string`.

430  
431 The above looks like this, in JSON schema:

```
432 "idType": {
433   "type": "string",
434   "pattern": "^[A-Za-z0-9_@$\-]+$"
435 }
```

436  
437 There are also cases, though, that data types cannot be mapped like above. One such  
438 case is the array data type, which was introduced in SDMX 3.0 as a new  
439 representation. In JSON schema an array is already natively foreseen, while in the  
440 XML schema, this has to be defined as a complex type, with an SDMX specific  
441 definition (i.e., specific element/attribute names for SDMX). Beyond that, the minimum  
442 and/or maximum number of items within an array is possible in both cases.

443  
444 Further to the above, the mapping between the non-native data types is presented in  
445 the table below:

SDMX Facet	XML Schema	JSON schema "pattern" <sup>1</sup> for "string" type
GregorianYear	xsd:gYear	"^-?([1-9][0-9]{3,} 0[0-9]{3})(Z (\+ -)((0[0-9] 1[0-3]):[0-5][0-9] 14:00))?\$"
GregorianMonth	xsd:gYearMonth	"^-?([1-9][0-9]{3,} 0[0-9]{3})-(0[1-9] 1[0-2])(Z (\+ -)((0[0-9] 1[0-3]):[0-5][0-9] 14:00))?\$"

<sup>1</sup> Regular expressions, as specified in [W3C XML Schema Definition Language \(XSD\) 1.1 Part 2: Datatypes](#).

GregorianDay	xsd:date	"^-[?([1-9][0-9]{3,} 0[0-9]{3})-(0[1-9] 1[0-2])-(0[1-9] 12 [0-9] 3[01]) (Z (\+ -) ((0[0-9] 1[0-3]):[0-5][0-9] 14:00))?\$"
Day	xsd:gDay	"^---(0[1-9] 12 [0-9] 3[01]) (Z (\+ -) ((0[0-9] 1[0-3]):[0-5][0-9] 14:00))?\$"
MonthDay	xsd:gMonthDay	"^--(0[1-9] 1[0-2])-(0[1-9] 12 [0-9] 3[01]) (Z (\+ -) ((0[0-9] 1[0-3]):[0-5][0-9] 14:00))?\$"
Month	xsd:Month	"^--(0[1-9] 1[0-2]) (Z (\+ -) ((0[0-9] 1[0-3]):[0-5][0-9] 14:00))?\$"
Duration	xsd:duration	"^-[?P[0-9]+Y?([0-9]+M)?([0-9]+D)?(T([0-9]+H)?([0-9]+M)?([0-9]+(\.[0-9]+)?S)?)?\$"

446  
447

## 448 **4.2 Time and Time Format**

449 This section does not go into great detail on these topics but provides a useful overview  
450 of these features to assist implementors in further use of the parts of the specification  
451 which are relevant to them.

### 452 **4.2.1 Introduction**

453 First, it is important to recognize that most observation times are a period. SDMX  
454 specifies precisely how Time is handled.

455  
456 The representation of time is broken into a hierarchical collection of representations. A  
457 data structure definition can use of any of the representations in the hierarchy as the  
458 representation of time. This allows for the time dimension of a particular data structure  
459 definition allow for only a subset of the default representation.

460  
461 The hierarchy of time formats is as follows (**bold** indicates a category which is made  
462 up of multiple formats, *italic* indicates a distinct format):

- 463
- 464 • **Observational Time Period**
    - 465 ○ **Standard Time Period**
      - 466 ▪ **Basic Time Period**
        - 467 • **Gregorian Time Period**
        - 468 • *Date Time*
      - 469 ▪ **Reporting Time Period**
    - 470 ○ *Time Range*

471  
472 The details of these time period categories and of the distinct formats which make them  
473 up are detailed in the sections to follow.

### 474 **4.2.2 Observational Time Period**

475 This is the superset of all time representations in SDMX. This allows for time to be  
476 expressed as any of the allowable formats.

### 477 **4.2.3 Standard Time Period**

478 This is the superset of any predefined time period or a distinct point in time. A time  
479 period consists of a distinct start and end point. If the start and end of a period are  
480 expressed as date instead of a complete date time, then it is implied that the start of

481 the period is the beginning of the start day (i.e. 00:00:00) and the end of the period is  
482 the end of the end day (i.e. 23:59:59).

#### 483 **4.2.4 Gregorian Time Period**

484 A Gregorian time period is always represented by a Gregorian year, year-month, or  
485 day. These are all based on ISO 8601 dates. The representation in SDMX-ML  
486 messages and the period covered by each of the Gregorian time periods are as follows:  
487

##### 488 **Gregorian Year:**

489 Representation: xs:gYear (YYYY)

490 Period: the start of January 1 to the end of December 31

##### 491 **Gregorian Year Month:**

492 Representation: xs:gYearMonth (YYYY-MM)

493 Period: the start of the first day of the month to end of the last day of the month

##### 494 **Gregorian Day:**

495 Representation: xs:date (YYYY-MM-DD)

496 Period: the start of the day (00:00:00) to the end of the day (23:59:59)

#### 497 **4.2.5 Date Time**

498 This is used to unambiguously state that a date-time represents an observation at a  
499 single point in time. Therefore, if one wants to use SDMX for data which is measured  
500 at a distinct point in time rather than being reported over a period, the date-time  
501 representation can be used.

502 Representation: xs:dateTime (YYYY-MM-DDThh:mm:ss)<sup>2</sup>

#### 503 **4.2.6 Standard Reporting Period**

504 Standard reporting periods are periods of time in relation to a reporting year. Each of  
505 these standard reporting periods has a duration (based on the ISO 8601 definition)  
506 associated with it. The general format of a reporting period is as follows:  
507

508 [REPORTING\_YEAR]-[PERIOD\_INDICATOR][PERIOD\_VALUE]

509 Where:

511 REPORTING\_YEAR represents the reporting year as four digits (YYYY)

512 PERIOD\_INDICATOR identifies the type of period which determines the duration  
513 of the period

514 PERIOD\_VALUE indicates the actual period within the year

515

516 The following section details each of the standard reporting periods defined in SDMX:  
517

##### 518 **Reporting Year:**

519 Period Indicator: A

520 Period Duration: P1Y (one year)

521 Limit per year: 1

522 Representation: common:ReportingYearType (YYYY-A1, e.g. 2000-A1)

##### 523 **Reporting Semester:**

524 Period Indicator: S

525 Period Duration: P6M (six months)

526 Limit per year: 2

527 Representation: common:ReportingSemesterType (YYYY-Ss, e.g. 2000-S2)

---

<sup>2</sup> The seconds can be reported fractionally

528 **Reporting Trimester:**  
 529 Period Indicator: T  
 530 Period Duration: P4M (four months)  
 531 Limit per year: 3  
 532 Representation: common:ReportingTrimesterType (YYYY-Tt, e.g. 2000-T3)  
 533 **Reporting Quarter:**  
 534 Period Indicator: Q  
 535 Period Duration: P3M (three months)  
 536 Limit per year: 4  
 537 Representation: common:ReportingQuarterType (YYYY-Qq, e.g. 2000-Q4)  
 538 **Reporting Month:**  
 539 Period Indicator: M  
 540 Period Duration: P1M (one month)  
 541 Limit per year: 1  
 542 Representation: common:ReportingMonthType (YYYY-Mmm, e.g. 2000-M12)  
 543 Notes: The reporting month is always represented as two digits, therefore 1-9  
 544 are 0 padded (e.g. 01). This allows the values to be sorted chronologically using  
 545 textual sorting methods.  
 546 **Reporting Week:**  
 547 Period Indicator: W  
 548 Period Duration: P7D (seven days)  
 549 Limit per year: 53  
 550 Representation: common:ReportingWeekType (YYYY-Www, e.g. 2000-W53)  
 551 Notes: There are either 52 or 53 weeks in a reporting year. This is based on the  
 552 ISO 8601 definition of a week (Monday - Saturday), where the first week of a  
 553 reporting year is defined as the week with the first Thursday on or after the  
 554 reporting year start day.<sup>3</sup> The reporting week is always represented as two digits,  
 555 therefore 1-9 are 0 padded (e.g. 01). This allows the values to be sorted  
 556 chronologically using textual sorting methods.  
 557 **Reporting Day:**  
 558 Period Indicator: D  
 559 Period Duration: P1D (one day)  
 560 Limit per year: 366  
 561 Representation: common:ReportingDayType (YYYY-Dddd, e.g. 2000-D366)  
 562 Notes: There are either 365 or 366 days in a reporting year, depending on  
 563 whether the reporting year includes leap day (February 29). The reporting day is  
 564 always represented as three digits, therefore 1-99 are 0 padded (e.g. 001). This  
 565 allows the values to be sorted chronologically using textual sorting methods.  
 566  
 567 The meaning of a reporting year is always based on the start day of the year and  
 568 requires that the reporting year is expressed as the year at the start of the period. This  
 569 start day is always the same for a reporting year, and is expressed as a day and a  
 570 month (e.g. July 1). Therefore, the reporting year 2000 with a start day of July 1 begins  
 571 on July 1, 2000.  
 572  
 573 A specialized attribute (reporting year start day) exists for the purpose of  
 574 communicating the reporting year start day. This attribute has a fixed identifier  
 575 (REPORTING\_YEAR\_START\_DAY) and a fixed representation (xs:gMonthDay) so

---

<sup>3</sup> ISO 8601 defines alternative definitions for the first week, all of which produce equivalent results. Any of these definitions could be substituted so long as they are in relation to the reporting year start day.

576 that it can always be easily identified and processed in a data message. Although this  
 577 attribute exists in specialized sub-class, it functions the same as any other attribute  
 578 outside of its identification and representation. It must takes its identity from a concept  
 579 and state its relationship with other components of the data structure definition. The  
 580 ability to state this relationship allows this reporting year start day attribute to exist at  
 581 the appropriate levels of a data message. In the absence of this attribute, the reporting  
 582 year start date is assumed to be January 1; therefore if the reporting year coincides  
 583 with the calendar year, this Attribute is not necessary.

584  
 585 Since the duration and the reporting year start day are known for any reporting period,  
 586 it is possible to relate any reporting period to a distinct calendar period. The actual  
 587 Gregorian calendar period covered by the reporting period can be computed as follows  
 588 (based on the standard format of [REPORTING\_YEAR]-  
 589 [PERIOD\_INDICATOR][PERIOD\_VALUE] and the reporting year start day as  
 590 [REPORTING\_YEAR\_START\_DAY]):

591  
 592 **1. Determine [REPORTING\_YEAR\_BASE]:**

593 Combine [REPORTING\_YEAR] of the reporting period value (YYYY) with  
 594 [REPORTING\_YEAR\_START\_DAY] (MM-DD) to get a date (YYYY-MM-DD).  
 595 This is the [REPORTING\_YEAR\_START\_DATE]

596 **a) If the [PERIOD\_INDICATOR] is W:**

- 597 **1. If [REPORTING\_YEAR\_START\_DATE] is a Friday,**  
 598 **Saturday, or Sunday:**

599 Add<sup>4</sup> (P3D, P2D, or P1D respectively) to the  
 600 [REPORTING\_YEAR\_START\_DATE]. The result is the  
 601 [REPORTING\_YEAR\_BASE].

- 602 **2. If [REPORTING\_YEAR\_START\_DATE] is a Monday,**  
 603 **Tuesday, Wednesday, or Thursday:**

604 Add<sup>4</sup> (P0D, -P1D, -P2D, or -P3D respectively) to the  
 605 [REPORTING\_YEAR\_START\_DATE]. The result is the  
 606 [REPORTING\_YEAR\_BASE].

607 **b) Else:**

608 The [REPORTING\_YEAR\_START\_DATE] is the  
 609 [REPORTING\_YEAR\_BASE].

610 **2. Determine [PERIOD\_DURATION]:**

- 611 a) If the [PERIOD\_INDICATOR] is A, the [PERIOD\_DURATION] is P1Y.  
 612 b) If the [PERIOD\_INDICATOR] is S, the [PERIOD\_DURATION] is P6M.  
 613 c) If the [PERIOD\_INDICATOR] is T, the [PERIOD\_DURATION] is P4M.  
 614 d) If the [PERIOD\_INDICATOR] is Q, the [PERIOD\_DURATION] is P3M.  
 615 e) If the [PERIOD\_INDICATOR] is M, the [PERIOD\_DURATION] is P1M.  
 616 f) If the [PERIOD\_INDICATOR] is W, the [PERIOD\_DURATION] is P7D.  
 617 g) If the [PERIOD\_INDICATOR] is D, the [PERIOD\_DURATION] is P1D.

618 **3. Determine [PERIOD\_START]:**

619 Subtract one from the [PERIOD\_VALUE] and multiply this by the  
 620 [PERIOD\_DURATION]. Add<sup>4</sup> this to the [REPORTING\_YEAR\_BASE]. The  
 621 result is the [PERIOD\_START].

622 **4. Determine the [PERIOD\_END]:**

---

<sup>4</sup> The rules for adding durations to a date time are described in the W3C XML Schema specification. See <http://www.w3.org/TR/xmlschema-2/#adding-durations-to-dateTimes> for further details.

623 Multiply the [PERIOD\_VALUE] by the [PERIOD\_DURATION]. Add<sup>4</sup> this to  
 624 the [REPORTING\_YEAR\_BASE] add<sup>4</sup> -P1D. The result is the  
 625 [PERIOD\_END].  
 626

627 For all of these ranges, the bounds include the beginning of the [PERIOD\_START]  
 628 (i.e. 00:00:00) and the end of the [PERIOD\_END] (i.e. 23:59:59).  
 629

630 **Examples:**

631

632 **2010-Q2, REPORTING\_YEAR\_START\_DAY = --07-01 (July 1)**

633 1. [REPORTING\_YEAR\_START\_DATE] = 2010-07-01

634 b) [REPORTING\_YEAR\_BASE] = 2010-07-01

635 2. [PERIOD\_DURATION] = P3M

636 3. (2-1) \* P3M = P3M

637 2010-07-01 + P3M = 2010-10-01

638 [PERIOD\_START] = 2010-10-01

639 4. 2 \* P3M = P6M

640 2010-07-01 + P6M = 2010-13-01 = 2011-01-01

641 2011-01-01 + -P1D = 2010-12-31

642 [PERIOD\_END] = 2010-12-31  
 643

644 The actual calendar range covered by 2010-Q2 (assuming the reporting year  
 645 begins July 1) is 2010-10-01T00:00:00/2010-12-31T23:59:59  
 646

647 **2011-W36, REPORTING\_YEAR\_START\_DAY = --07-01 (July 1)**

648 1. [REPORTING\_YEAR\_START\_DATE] = 2010-07-01

649 a) 2011-07-01 = Friday

650 2011-07-01 + P3D = 2011-07-04

651 [REPORTING\_YEAR\_BASE] = 2011-07-04

652 2. [PERIOD\_DURATION] = P7D

653 3. (36-1) \* P7D = P245D

654 2011-07-04 + P245D = 2012-03-05

655 [PERIOD\_START] = 2012-03-05

656 4. 36 \* P7D = P252D

657 2011-07-04 + P252D = 2012-03-12

658 2012-03-12 + -P1D = 2012-03-11

659 [PERIOD\_END] = 2012-03-11  
 660

661 The actual calendar range covered by 2011-W36 (assuming the reporting year  
 662 begins July 1) is 2012-03-05T00:00:00/2012-03-11T23:59:59  
 663

#### 664 **4.2.7 Distinct Range**

665 In the case that the reporting period does not fit into one of the prescribe periods above,  
 666 a distinct time range can be used. The value of these ranges is based on the ISO 8601  
 667 time interval format of start/duration. Start can be expressed as either an ISO 8601  
 668 date or a date-time, and duration is expressed as an ISO 8601 duration. However, the  
 669 duration can only be positive.  
 670

671 **4.2.8 Time Format**

672 In version 2.0 of SDMX there is a recommendation to use the time format attribute to  
 673 gives additional information on the way time is represented in the message. Following  
 674 an appraisal of its usefulness this is no longer required. However, it is still possible, if  
 675 required , to include the time format attribute in SDMX-ML.  
 676

Code	Format
OTP	Observational Time Period: Superset of all SDMX time formats (Gregorian Time Period, Reporting Time Period, and Time Range)
STP	Standard Time Period: Superset of Gregorian and Reporting Time Periods
GTP	Superset of all Gregorian Time Periods and date-time
RTP	Superset of all Reporting Time Periods
TR	Time Range: Start time and duration (YYYY-MM-DD(Thh:mm:ss)?/<duration>)
GY	Gregorian Year (YYYY)
GTM	Gregorian Year Month (YYYY-MM)
GD	Gregorian Day (YYYY-MM-DD)
DT	Distinct Point: date-time (YYYY-MM-DDThh:mm:ss)
RY	Reporting Year (YYYY-A1)
RS	Reporting Semester (YYYY-Ss)
RT	Reporting Trimester (YYYY-Tt)
RQ	Reporting Quarter (YYYY-Qq)
RM	Reporting Month (YYYY-Mmm)
RW	Reporting Week (YYYY-Www)
RD	Reporting Day (YYYY-Dddd)

677 **Table 1: SDMX-ML Time Format Codes**

678 **4.2.9 Time Zones**

679 In alignment with ISO 8601, SDMX allows the specification of a time zone on all time  
 680 periods and on the reporting year start day. If a time zone is provided on a reporting  
 681 year start day, then the same time zone (or none) should be reported for each reporting  
 682 time period. If the reporting year start day and the reporting period time zone differ, the  
 683 time zone of the reporting period will take precedence. Examples of each format with  
 684 time zones are as follows (time zone indicated in bold):  
 685

- 686 • Time Range (start date): 2006-06-05-**05:00**/P5D
- 687 • Time Range (start date-time): 2006-06-05T00:00:00-**05:00**/P5D
- 688 • Gregorian Year: 2006-**05:00**
- 689 • Gregorian Month: 2006-06-**05:00**
- 690 • Gregorian Day: 2006-06-05-**05:00**
- 691 • Distinct Point: 2006-06-05T00:00:00-**05:00**
- 692 • Reporting Year: 2006-A1-**05:00**
- 693 • Reporting Semester: 2006-S2-**05:00**

- 694 • Reporting Trimester: 2006-T2-05:00
- 695 • Reporting Quarter: 2006-Q3-05:00
- 696 • Reporting Month: 2006-M06-05:00
- 697 • Reporting Week: 2006-W23-05:00
- 698 • Reporting Day: 2006-D156-05:00
- 699 • Reporting Year Start Day: --07-01-05:00

700 According to ISO 8601, a date without a time-zone is considered "local time". SDMX  
 701 assumes that local time is that of the sender of the message. In this version of SDMX,  
 702 an optional field is added to the sender definition in the header for specifying a time  
 703 zone. This field has a default value of 'Z' (UTC). This determination of local time applies  
 704 for all dates in a message.

#### 705 **4.2.10 Representing Time Spans Elsewhere**

706 It has been possible since SDMX 2.0 for a Component to specify a representation of a  
 707 time span. Depending on the format of the data message, this resulted in either an  
 708 element with 2 XML attributes for holding the start time and the duration or two  
 709 separate XML attributes based on the underlying Component identifier. For example,  
 710 if REF\_PERIOD were given a representation of time span, then in the Compact data  
 711 format, it would be represented by two XML attributes; REF\_PERIODStartTime  
 712 (holding the start) and REF\_PERIOD (holding the duration). If a new simple type is  
 713 introduced in the SDMX schemas that can hold ISO 8601 time intervals, then this will  
 714 no longer be necessary. What was represented as this:

```
715  

  716 <Series REF_PERIODStartTime="2000-01-01T00:00:00" REF_PERIOD="P2M"/>
```

717  
 718 can now be represented with this:

```
719  

  720 <Series REF_PERIOD="2000-01-01T00:00:00/P2M"/>
```

#### 721 **4.2.11 Notes on Formats**

722 There is no ambiguity in these formats so that for any given value of time, the category  
 723 of the period (and thus the intended time period range) is always clear. It should also  
 724 be noted that by utilizing the ISO 8601 format, and a format loosely based on it for the  
 725 report periods, the values of time can easily be sorted chronologically without  
 726 additional parsing.

#### 727 **4.2.12 Effect on Time Ranges**

728 All SDMX-ML data messages are capable of functioning in a manner similar to SDMX-  
 729 EDI if the Dimension at the observation level is time: the time period for the first  
 730 observation can be stated and the rest of the observations can omit the time value as  
 731 it can be derived from the start time and the frequency. Since the frequency can be  
 732 determined based on the actual format of the time value for everything but distinct  
 733 points in time and time ranges, this makes it even simpler to process as the interval  
 734 between time ranges is known directly from the time value.

#### 735 **4.2.13 Time in Query Messages**

736 When querying for time values, the value of a time parameter can be provided as any  
 737 of the Observational Time Period formats and must be paired with an operator. This

738 section will detail how systems processing query messages should interpret these  
 739 parameters.

740  
 741 Fundamental to processing a time value parameter in a query message is  
 742 understanding that all time periods should be handled as a distinct range of time. Since  
 743 the time parameter in the query is paired with an operator, this also effectively  
 744 represents a distinct range of time. Therefore, a system processing the query must  
 745 simply match the data where the time period for requested parameter is encompassed  
 746 by the time period resulting from value of the query parameter. The following table  
 747 details how the operators should be interpreted for any time period provided as a  
 748 parameter.

749

Operator	Rule
Greater Than	Any data after the last moment of the period
Less Than	Any data before the first moment of the period
Greater Than or Equal To	Any data on or after the first moment of the period
Less Than or Equal To	Any data on or before the last moment of the period
Equal To	Any data which falls on or after the first moment of the period and before or on the last moment of the period

750

751 Reporting Time Periods as query parameters are handled like this: any data within the  
 752 bounds of the reporting period for the year is matched, regardless of the actual start  
 753 day of the reporting year. In addition, data reported against a normal calendar period  
 754 is matched if it falls within the bounds of the time parameter based on a reporting year  
 755 start day of January 1. When determining whether another reporting period falls within  
 756 the bounds of a report period query parameter, one will have to take into account the  
 757 actual time period to compare weeks and days to higher order report periods. This will  
 758 be demonstrated in the examples to follow.

759

760 **Examples:**

761

762 **Gregorian Period**

763

Query Parameter: Greater than 2010

764

Literal Interpretation: Any data where the start period occurs after 2010-12-31T23:59:59.

765

766

Example Matches:

767

- 2011 or later
- 2011-01 or later
- 2011-01-01 or later
- 2011-01-01/P[Any Duration] or any later start date
- 2011-[Any reporting period] (any reporting year start day)
- 2010-S2 (reporting year start day --07-01 or later)
- 2010-T3 (reporting year start day --07-01 or later)
- 2010-Q3 or later (reporting year start day --07-01 or later)
- 2010-M07 or later (reporting year start day --07-01 or later)
- 2010-W28 or later (reporting year start day --07-01 or later)
- 2010-D185 or later (reporting year start day --07-01 or later)

768

769

770

771

772

773

774

775

776

777

778

779

**Reporting Period**

780 Query Parameter: Greater than or equal to 2010-Q3  
 781 Literal Interpretation: Any data with a reporting period where the start period is on  
 782 or after the start period of 2010-Q3 for the same reporting year start day, or and  
 783 data where the start period is on or after 2010-07-01.  
 784 Example Matches:

- 785 • 2011 or later
- 786 • 2010-07 or later
- 787 • 2010-07-01 or later
- 788 • 2010-07-01/P[Any Duration] or any later start date
- 789 • 2011-[Any reporting period] (any reporting year start day)
- 790 • 2010-S2 (any reporting year start day)
- 791 • 2010-T3 (any reporting year start day)
- 792 • 2010-Q3 or later (any reporting year start day)
- 793 • 2010-M07 or later (any reporting year start day)
- 794 • 2010-W27 or later (reporting year start day --01-01)<sup>5</sup>
- 795 • 2010-D182 or later (reporting year start day --01-01)
- 796 • 2010-W28 or later (reporting year start day --07-01)<sup>6</sup>
- 797 • 2010-D185 or later (reporting year start day --07-01)

## 798 **4.3 Versioning**

799 Versioning operates at the level of versionable and maintainable objects in the SDMX  
 800 information model. Within the SDMX Structure and MetadataSet messages, there is a  
 801 well-defined pattern for artefact versioning and referencing. The artefact identifiers are  
 802 qualified by their version numbers – that is, an object with an Agency of "A", and ID of  
 803 "X" and a version of "1.0.0" is a different object than one with an Agency of "A", an ID  
 804 of "X", and a version of "1.1.0".

805  
 806 As of SDMX 3.0, the versioning rules are extended to allow for truly versioned artefacts  
 807 through the implementation of the rules of the well-known practice called "Semantic  
 808 Versioning" (<http://semver.org>), in addition to the legacy non-restrictive versioning  
 809 scheme. In addition, the "isFinal" property is removed from  
 810 *MaintainableArtefact*. According to the legacy versioning, any artefact defined  
 811 without a version is equivalent to following the legacy versioning, thus having version  
 812 '1.0'.

### 813 **4.3.1 Non-versioned artefacts**

814 Indeed, some use cases do not need or are incompatible with versioning for some or  
 815 all their structural artefacts, such as the Agency, Data Providers, Metadata Providers  
 816 and Data Consumer Schemes. These artefacts follow the legacy versioning, with a  
 817 fixed version set to '1.0'.

818  
 819 Many existing organisation's data management systems work with version-less  
 820 structures and apply ad-hoc structural metadata governance processes. The new non-  
 821 versioned artefacts will allow supporting those numerous situations, where  
 822 organisations do not manage version numbers.

---

<sup>5</sup> 2010-Q3 (with a reporting year start day of --01-01) starts on 2010-07-01. This is day 4 of week 26, therefore the first week matched is week 27.

<sup>6</sup> 2010-Q3 (with a reporting year start day of --07-01) starts on 2011-01-01. This is day 6 of week 27, therefore the first week matched is week 28.

823

#### 824 **4.3.2 Semantically versioned artefacts**

825 Since the purpose of SDMX versioning is to allow communicating the structural artefact  
826 changes to data exchange partners and connected systems, SDMX 3.0 offers  
827 Semantic Versioning (aka SemVer) with a clear and unambiguous syntax to all  
828 semantically versioned SDMX 3.0 structural artefacts. Semantic versioning will thus  
829 better respond to situations where the SDMX standard itself is the only structural  
830 contract between data providers and data consumers and where changes in structures  
831 can only be communicated through the version number increases.

832

833 The semantic version number consists of four parts: MAJOR, MINOR, PATCH and  
834 EXTENSION, the first three parts being separated by a dot (.), the last two parts being  
835 separated by a hyphen (-): MAJOR.MINOR.PATCH-EXTENSION. All versions are  
836 ordered.

837

838 The detailed rules for semantic versioning are listed in chapter 14 in the annex for  
839 “Semantic Versioning”. In short, they define:

840

841 Given a version number MAJOR.MINOR.PATCH (without EXTENSION), when making  
842 changes to that semantically versioned SDMX artefact, then one must increment the:

843

1. MAJOR version when backwards incompatible artefact changes are  
844 made,

845

2. MINOR version when artefact elements are added in a backwards  
846 compatible manner, or

847

3. PATCH version when backwards compatible artefact property changes  
848 are made.

849

850 When incrementing a version part, the right-hand side parts are 0-ed (reset to ‘0’).

851

852 Extensions can be added, changed or dropped.

853

854 Given an extended version number MAJOR.MINOR.PATCH-EXTENSION, when  
855 making changes to that versioned artefact, then one is not required to increment the  
856 version if those changes are within the allowed scope of the version increment from  
857 the previous version (if that existed); otherwise, the above version increment rules  
858 apply. EXTENSIONS can be used e.g., for drafting or a pre-release.

859 Semantically versioned SDMX artefacts will thus be safe to use. Specific version  
860 patterns allow them to become either immutable, i.e., the maintainer commits to never  
861 change their content, or changeable only within a well-defined scope. If any further  
862 change is required, a new version must be created first. Furthermore, the impact of the  
863 further change is communicated using a clear version increment. The built-in version  
864 extension facility allows for eased drafting of new SDMX artefact versions.

865

866 The production versions of identifiable artefacts are assumed stable, i.e., they do not  
867 have an EXTENSION. This is because once in production, an artefact cannot change  
868 in any way, or it must change the version. For cases where an artefact is not static,  
869 like during the drafting, the version must indicate this by including an EXTENSION. Draft  
870 artefacts should not be used outside of a specific system designed to accommodate

871 them. For most purposes, all artefacts should become stable before being used in  
872 production.

### 873 **4.3.3 Legacy-versioned artefacts**

874 Organisations wishing to keep a maximum of backwards compatibility with existing  
875 implementations can continue using the previous 2-digit convention for version  
876 numbers (`MAJOR.MINOR`) as in the past, such as '2.3', but without the 'isFinal' property.  
877 The new SDMX 3.0 standard does not add any strict rules or guarantees about  
878 changes in those artefacts, since the legacy versioning rules were rather loose and  
879 non-binding, including the meaning of the 'isFinal' property, and their implementations  
880 were varying.

881  
882 In order to make artefacts immutable or changes truly predictable, a move to the new  
883 semantic versioning syntax is required.

### 884 **4.3.4 Dependency management and references**

885 New flexible dependency specifications with wildcarding allow for easier data model  
886 maintenance and enhancements for semantically versioned SDMX artefacts. This  
887 allows implementing a smart referencing mechanism, whereby an artefact may  
888 reference:

- 889 - a fixed version of another artefact
  - 890 - the **latest available** version of another artefact
  - 891 - the **latest backward compatible** version of another artefact, or
- 892 the **latest backward and forward compatible** version of another artefact.

893  
894 References not representing a strict artefact dependency, such as the target artefacts  
895 defined in a `MetadataProvisionAgreement` allow for linking to **all currently**  
896 **available** versions of another artefact. Another illustrative case for such loose  
897 referencing is that of Constraints and flows. A Constraint may reference many  
898 Dataflows or Metadataflows, the addition of more references to flow objects does not  
899 version the Constraint. This is because the Constraints are not properties of the flows  
900 – they merely make references to them.

901  
902 Semantically versioned artefacts must only reference other semantically versioned  
903 artefacts, which may include extended versions. Non-versioned and legacy-versioned  
904 artefacts can reference any other non-versioned or versioned (whether semantic or  
905 legacy) artefacts. The scope of wildcards in references adapts correspondingly.

906  
907 The mechanism named "early binding" refers to a dependency on a stable versioned  
908 artefact – everything with a stable versioned identity is a known quantity and will not  
909 change. The "late binding" mechanism is based on a wildcarded reference, and it  
910 resolves that reference and determines the currently related artefact at runtime.

911  
912 One area which is much impacted by this versioning scheme is the ability to reference  
913 external objects. With the many dependencies within the various structural objects in  
914 SDMX, it is useful to have a scheme for external referencing. This is done at the level  
915 of maintainable objects (DSDs, Codelists, Concept Schemes, etc.) In an SDMX  
916 Structure Message, whenever an "isExternalReference" attribute is set to true,  
917 then the application must resolve the address provided in the associated "uri"  
918 attribute and use the SDMX Structure Message stored at that location for the full

919 definition of the object in question. Alternately, if a registry "urn" attribute has been  
920 provided, the registry can be used to supply the full details of the object.

921

922 The detailed rules for dependency management and references are listed in chapter  
923 14 in the annex for "Semantic Versioning".

924

925 In order to allow resolving the described new forms of dependencies, the SDMX 3.0  
926 Rest API supports retrievals legacy-versioned, wildcarded and extended artefact  
927 versions:

- 928 - Artefact queries for a **specific** version (X.Y, X.Y.Z or X.Y.Z-EXT).
- 929 - Artefact queries for **latest available** semantic versions within the wildcard scope  
930 (X+.Y.Z, X.Y+.Z or X.Y.Z+).
- 931 - Queries for **non-versioned** artefacts.
- 932 - Artefact queries for **all available** semantic versions within the wildcard scope  
933 (\*, X.\* or X.Y.\*), where only the first form is required for resolving wildcarded  
934 loose references.

935 The combination of wildcarded queries with a specific version extension is not  
936 permitted.

937 Full details can be found in the SDMX RESTful web services specification.

#### 938 **4.4 Structural Metadata Querying Best Practices**

939 When querying for structural metadata, the ability to state how references should be  
940 resolved is quite powerful. However, this mechanism is not always necessary and can  
941 create an undue burden on the systems processing the queries if it is not used properly.

942

943 Any structural metadata object which contains a reference to an object can be queried  
944 based on that reference. For example, a categorisation references both a category and  
945 the object is it categorising. As this is the case, one can query for categorisations which  
946 categorise a particular object or which categorise against a particular category or  
947 category scheme. This mechanism should be used when the referenced object is  
948 known.

949

950 When the referenced object is not known, then the reference resolution mechanism  
951 could be used. For example, suppose one wanted to find all category schemes and  
952 the related categorisations for a given maintenance agency. In this case, one could  
953 query for the category scheme by the maintenance agency and specify that parent and  
954 sibling references should be resolved. This would result in the categorisations which  
955 reference the categories in the matched schemes to be returned, as well as the object  
956 which they categorise.

957 **5 Reference Metadata**

958 **5.1 Scope of the Metadata Structure Definition (MSD)**

959 The scope of the MSD is reduced in SDMX 3.0, by means of simplifying its structure,  
 960 but also in the way referenced Artefacts are targeted. In fact, the MSD is restricted to  
 961 play the role of a single container, without targeting any specific Artefact. The possible  
 962 targets of Metadata Set are specified in the Metadataflows or Metadata Provision  
 963 Agreements relating to that MSD. To achieve that, the structure of the Metadataflow  
 964 has changed in this version of the standard. Moreover, the Metadata Provision  
 965 Agreement Artefact is introduced to include this feature.

966  
 967 Two more changes, introduced in this version, are the following:

- 968 • The Metadata Set becomes a Maintainable Artefact but maintained by a Metadata  
 969 Provider (another new Artefact in this version).
- 970 • Metadata Attributes may also be used in Data Structure Definitions, as long as  
 971 the latter reference the Metadata Structure Definition that specify those Metadata  
 972 Attributes.

973

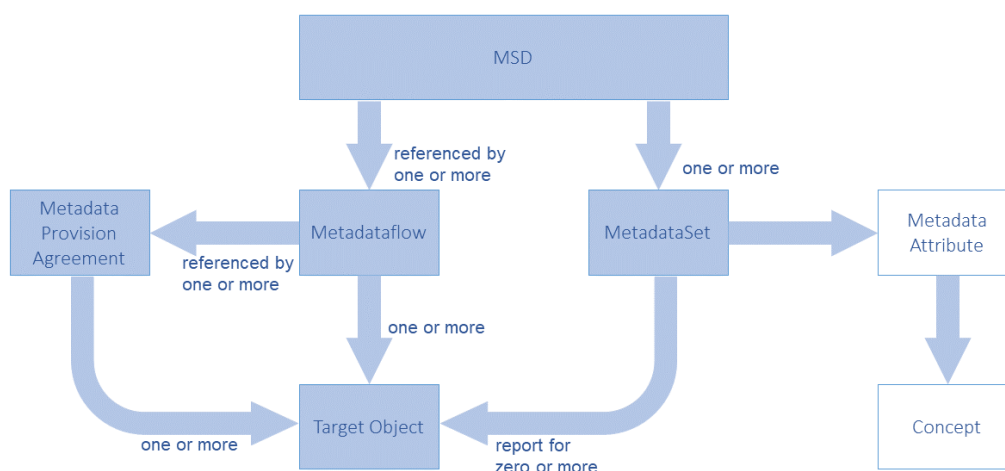
974 **5.2 Identification of the Object(s) to which the Metadata is to**  
 975 **be attached**

976 The following example shows the structure and naming of the MSD and related  
 977 components for creating reference metadata.

978

979 The schematic structure of an MSD is shown below.

980



981  
 982

**Figure 1: Schematic of the Metadata Structure Definition**

983 The MSD contains one Metadata Attribute Descriptor comprising the Metadata  
 984 Attributes that identify the Concepts for which metadata may be reported in the  
 985 Metadata Set. The Metadataflow and Metadata Provision Agreement comprise the

986 specification of the objects to which metadata can be reported in a Metadata Set  
987 (Metadata Target(s)).

988

989 The high-level view of the MSD, as well as the way the Metadataflow and Metadata  
990 Provision Agreement specify the Targets:

991

```
992 <str:MetadataStructure agencyID="SDMX" id="MSD" version="1.0.0-draft">
993   <com:Name>MSD 3.0 sample</com:Name>
994   <str:MetadataAttributeDescriptor id="MetadataAttributeDescriptor">
995     ...
996   </str: MetadataAttributeDescriptor>
997 </str:MetadataStructure>
```

998 **Figure 2: The high-level view of the MSD containing one Metadata Attribute Descriptor**

999

```
1000 <str:Metadataflow agencyID="OECD" id="GENERAL_METADATA" version="1.0.0-
1001 draft">
1002   <com:Name xml:lang="en">Metadataflow 3.0 sample</com:Name>
1003   <str:Structure>urn:sdmx:org.sdmx.infomodel.metadatastructure.
1004     MetadataStructure=OECD:MSD(1.0.0-draft)</str:Structure>
1005   <!-- Attach to any Dataflows maintained by the OECD -->
1006   <str:Targets>urn:sdmx:org.sdmx.infomodel.datastructure.
1007     Dataflow=OECD:* (*)</str:Targets>
1008 </str:Metadataflow>
```

1009

**Figure 3: Wildcarded Target Objects as specified in a Metadataflow**

1010

```
1011 <str:MetadataProvisionAgreement agencyID="OECD" id="ABS_INDICATORS"
1012 version="1.0.0-draft">
1013   <com:Name xml:lang="en">Metadata Provision Agreement 3.0 sample</com:Name>
1014   <str:StructureUsage>urn:sdmx:org.sdmx.infomodel.metadatastructure.
1015     Metadataflow=OECD:GENERAL_METADATA(1.0.0-draft)</str:StructureUsage>
1016   <str:MetadataProvider>urn:sdmx:org.sdmx.infomodel.base.
1017     MetadataProvider=OECD:METADATA_PROVIDERS(1.0).ABS</str:MetadataProvider>
1018   <!-- Attach to specific Dataflows maintained by the OECD -->
1019   <str:Target>urn:sdmx:org.sdmx.infomodel.datastructure.Dataflow=
1020     OECD:GDP (*)</str:Target>
1021   <str:Target>urn:sdmx:org.sdmx.infomodel.datastructure.Dataflow=
1022     OECD:EXR (*)</str:Target>
1023   <str:Target>urn:sdmx:org.sdmx.infomodel.datastructure.Dataflow=
1024     OECD:ABC (*)</str:Target>
1025 </str:MetadataProvisionAgreement>
```

1026

**Figure 4: Specific Target Objects as specified in a Metadata Provision Agreement**

1027 Note that the SDMX-ML schemas have specific XML elements for each identifiable  
1028 object type because identifying, for instance, a Maintainable Object has different  
1029 properties from an Identifiable Object which must also include the agencyId, version,  
1030 and id of the Maintainable Object in which it resides.

### 1031 **5.3 Metadata Structure Definition**

1032 An example is shown below.

1033

```
1034 <str:MetadataStructure agencyID="SDMX" id="MSD" version="1.0.0-draft">
1035   <com:Name>MSD 3.0 sample</com:Name>
1036   <str:MetadataAttributeDescriptor id="MetadataAttributeDescriptor">
1037     <str:MetadataAttribute id="CONTACT" isPresentational="true">
1038       <str:ConceptIdentity>urn:sdmx:org.sdmx.infomodel.conceptscheme.
```

```

1039     Concept=SDMX:CONCEPTS (1.0.0) .CONTACT</str:ConceptIdentity>
1040     <str:MetadataAttribute id="CONTACT_NAME" minOccurs="1" maxOccurs="1">
1041       <str:ConceptIdentity>urn:sdmx:org.sdmx.infomodel.conceptscheme.
1042         Concept=SDMX:CONCEPTS (1.0.0) .CONTACT_NAME</str:ConceptIdentity>
1043       <str:LocalRepresentation>
1044         <str:TextFormat textType="String"/>
1045       </str:LocalRepresentation>
1046     </str:MetadataAttribute>
1047     <str:MetadataAttribute id="ADDRESS" minOccurs="1" maxOccurs="3"
1048 isPresentational="true">
1049       <str:ConceptIdentity>urn:sdmx:org.sdmx.infomodel.conceptscheme.
1050         Concept=SDMX:CONCEPTS (1.0.0) .ADDRESS</str:ConceptIdentity>
1051       <str:MetadataAttribute id="HOUSE_NUMBER" minOccurs="1"
1052 maxOccurs="1">
1053         <str:ConceptIdentity>urn:sdmx:org.sdmx.infomodel.conceptscheme.
1054           Concept=SDMX:CONCEPTS (1.0.0) .HOUSE_NUMBER</str:ConceptIdentity>
1055         <str:LocalRepresentation>
1056           <str:TextFormat textType="Integer"/>
1057         </str:LocalRepresentation>
1058       </str:MetadataAttribute>
1059     </str:MetadataAttribute>
1060   </str:MetadataAttributeDescriptor>
1061 </str:MetadataStructure>
1062

```

1063 **Figure 5: Example MSD showing specification of some Metadata Attributes**

1064 This example shows the following hierarchy of Metadata Attributes:

- 1065 • Contact – this is presentational; no metadata is expected to be reported at this
 1066 level
  - 1067 ○ Contact Name
  - 1068 ○ Address – this is also presentational; up to 3 addresses are allowed
 1069 ▪ House Number

## 1070 5.4 Metadata Set

1071 An example of reporting metadata according to the MSD described above, is shown
 1072 below.

```

1073
1074 <msg:MetadataSet id="ALB" metadataProviderID="OECD" version="1.0.0">
1075   <str:MetadataProvision>urn:sdmx:org.sdmx.infomodel.registry.MetadataProvis
1076 ionAgreement=OECD:ABS_INDICATORS (1.0.0-draft)</str:MetadataProvision>
1077   <str:Target>urn:sdmx:org.sdmx.infomodel.datastructure.Dataflow=OECD:GDP (1.
1078 0.0)</str:Target>
1079   <md:AttributeSet>
1080     <md:ReportedAttribute id="CONTACT">
1081       <md:AttributeSet>
1082         <md:ReportedAttribute id="CONTACT_NAME">John Doe
1083         </md:ReportedAttribute>
1084         <md:ReportedAttribute id="ADDRESS">
1085           <md:AttributeSet>
1086             <md:ReportedAttribute id="STREET_NAME">
1087               <com:Text xml:lang="en">5th Avenue</com:Text>
1088             </md:ReportedAttribute>
1089             <md:ReportedAttribute id="HOUSE_NUMBER">12
1090             </md:ReportedAttribute>
1091           </md:AttributeSet>
1092         </md:ReportedAttribute>
1093         <md:ReportedAttribute id="HTML_ATTR">
1094           <com:StructuredText xml:lang="en">
1095             <div xmlns="http://www.w3.org/1999/xhtml">

```

```
1096         <p>Lorem Ipsum</p>
1097     </div>
1098 </com:StructuredText>
1099 </md:ReportedAttribute>
1100 </md:AttributeSet>
1101 </md:ReportedAttribute>
1102 </md:AttributeSet>
1103 </msg:MetadataSet>
```

1104 **Figure 6: Example Metadata Set**

1105 This example shows:

- 1106 1. The reference to the Metadata Provision Agreement and Metadata Target
- 1107 2. The reported metadata attributes (AttributeSet)

## 1108 **5.5 Reference Metadata in Data Structure Definition and**

### 1109 **Dataset**

1110 An important change of SDMX 3.0 is the ability to reference an MSD within a DSD, in  
1111 order to report any Metadata Attributes defined in the former to Datasets of the latter.

1112 This is achieved by the following:

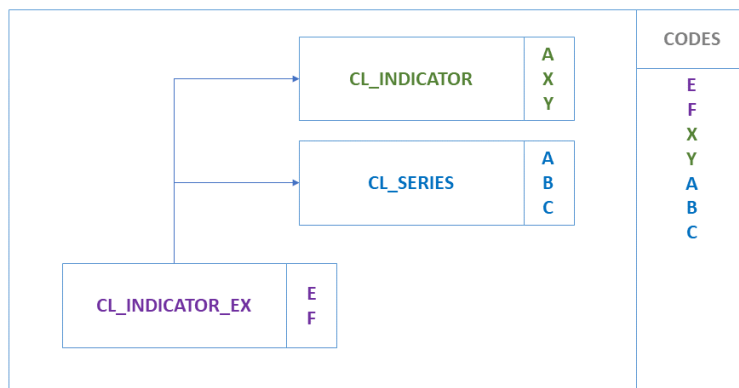
- 1113 • In a DSD, the user may add a reference to one MSD.
- 1114 • In the Attribute Descriptor of the DSD, the user may include any Metadata  
1115 Attributes defined in the linked MSD.
  - 1116 ○ For each link to a Metadata Attribute, an Attribute Relationship may be  
1117 specified (similarly to that for Data Attributes).
- 1118 • In any Dataset complying with this DSD, Metadata Attributes may be reported  
1119 according to the specified Attribute Relationship.
  - 1120 ○ The hierarchy of the Metadata Attributes defined in the MSD must be  
1121 respected and they are reported in the same way as in a Metadataset,  
1122 under the level they are related within the DSD, via their Attribute  
1123 Relationship.
- 1124 • In Data Constraints, the user is allowed to restrict values for Metadata  
1125 Attributes, in the same way as Data Attributes (more on this in section “10  
1126 Constraints”).

1127 **6 Codelist**

1128 As of SDMX 3.0, Codelists have gained new features like geospatial properties,  
 1129 inheritance and extension. Moreover, hierarchies (used to build complex hierarchies  
 1130 of one or more Codelists) are now linked to other Artefacts in order to facilitate the  
 1131 formers' usage in dissemination or other scenarios. For all geospatial related features,  
 1132 as well as the new Geographical Codelist, please refer to section 7.  
 1133

1134 **6.1 Codelist extension and discriminated unions**

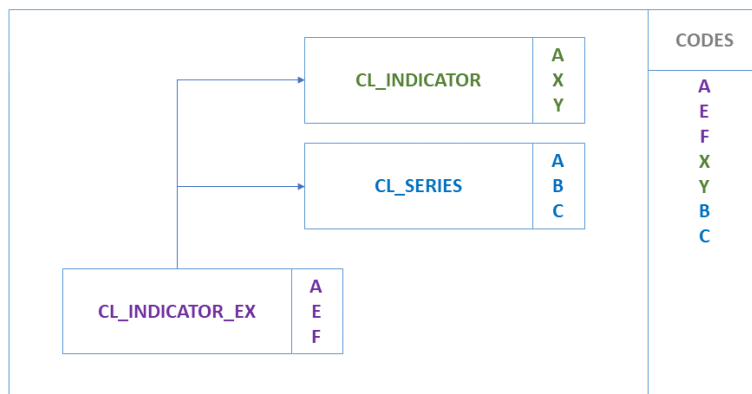
1135 A Codelist can extend one or more Codelists. Codelist extensions are defined  
 1136 as a list of references to parent Codelists. The order of the references is important  
 1137 when it comes to conflict resolution on Code Ids. When two Codelists have the  
 1138 same Code Id, the Codelist referenced later takes priority. In the example below,  
 1139 the code 'A', exists in both CL\_INDICATOR and CL\_SERIES. The Codelist  
 1140 CL\_INDICATOR\_EX will contain the code 'A' from CL\_SERIES as this was the second  
 1141 Codelist to be referenced in the sequence of references.



1142  
 1143

**Figure 7: Codelist extension**

1144 As the extended Codelist, CL\_INDICATOR\_EX in this example, may also define its  
 1145 own Codes, these take the ultimate priority over any referenced Codelists. If  
 1146 CL\_INDICATOR\_EX defines Code 'A', then this will be used instead of Code 'A' from  
 1147 CL\_INDICATOR and CL\_SERIES, as shown below:

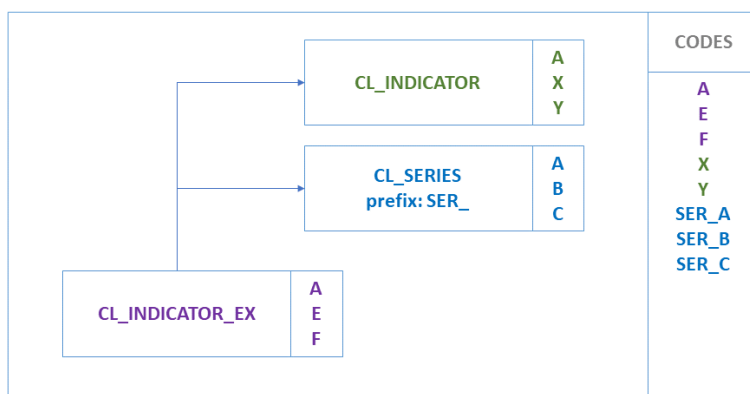


1148  
1149  
1150

**Figure 8: Codelist extension with new Codes**

1151 **6.1.1 Prefixing Code Ids**

1152 A reference to a Codelist may contain a prefix. If a prefix is provided, this prefix will  
1153 be applied to all the codes in the Codelist before they are imported into the extended  
1154 Codelist. Following the above example if the CL\_SERIES reference includes a prefix  
1155 of 'SER\_' then the resulting Codelist would contain 7 codes, A, E, F, X, Y, SER\_A,  
1156 SER\_B, SER\_C.

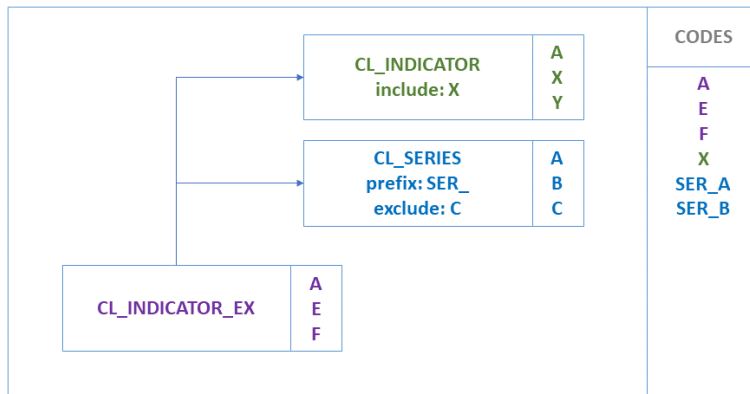


1157  
1158

**Figure 9: Extended Codelist with prefix**

1159 **6.1.2 Including / Excluding Specific Codes**

1160 The default behaviour of extending another Codelist is to import all Codes. However,  
1161 an explicit list of Code Ids may be provided for explicit inclusion or exclusion. This list  
1162 of Ids may contain wildcards using the same notation as Constraints (%).  
1163 Cascading values is also supported using the same syntax as the Constraints. The  
1164 list of Ids is either a list of excluded items, or included items, exclusion and inclusion  
1165 is not supported against a single Codelist.

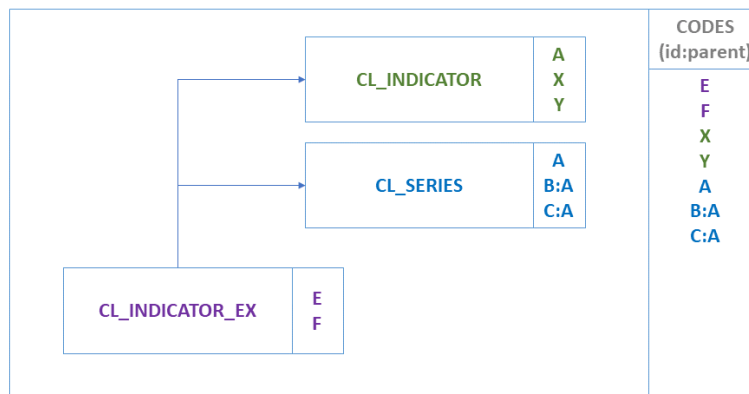


1166  
1167

**Figure 10: Extended Codelist with include/exclude terms**

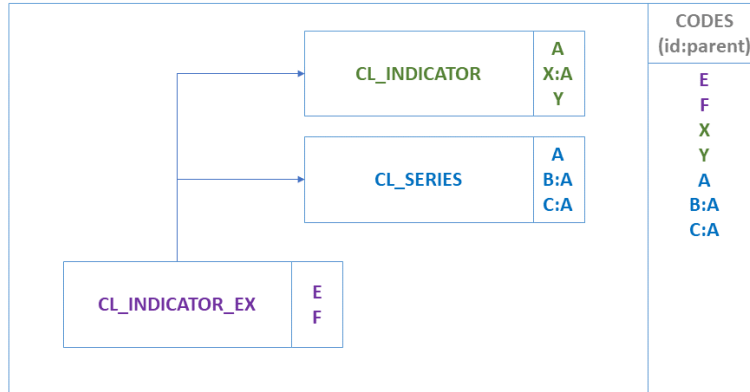
1168 **6.1.3 Parent Ids**

1169 Parent Ids are preserved in the extended Codelist if they can be. If a Code is inherited  
 1170 but its parent Code is excluded, then the Code's parent Id will be removed. This rule is  
 1171 also true if the parent Code is excluded because it is overwritten by another Code with the  
 1172 same Id from another Codelist. This ensures the parent Ids do not inadvertently link to  
 1173 Codes originating from different Codelists, and also prevents circular references from  
 1174 occurring.



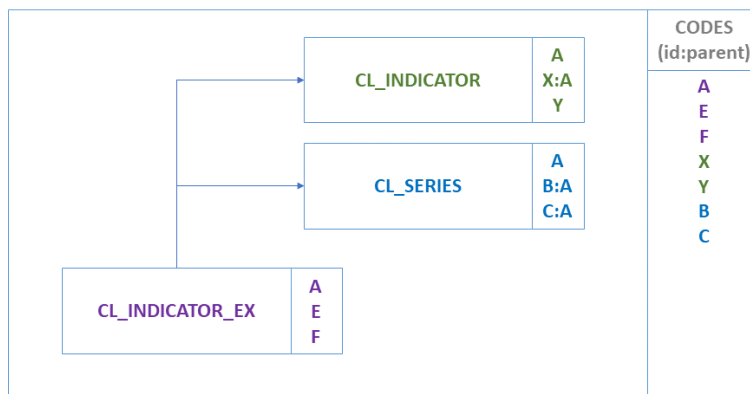
1175  
1176

**Figure 11: Parent Code included**



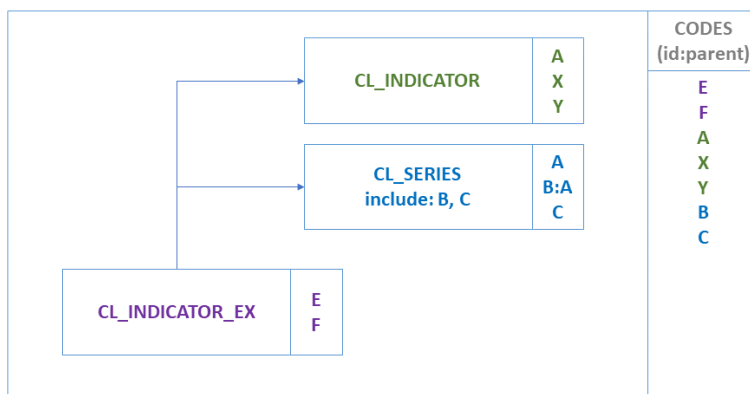
1177  
1178

**Figure 12: Parent Code from different extended Codelist**



1179  
1180

**Figure 13: Parent Code overridden by local Code**



1181  
1182

**Figure 14: Parent Code not included**

1183 **6.1.4 Discriminated Unions**

1184 A common use case solved in SDMX 3.0 is that of discriminated unions, i.e., dealing  
 1185 with classification or breakdown "variants" which are all valid but mutually exclusive.  
 1186 For example, there are many versions of the international classification for economic  
 1187 activities "ISIC". In SDMX, classifications are enumerated concepts, normally modelled  
 1188 as dimensions corresponding to breakdowns. Each enumerated concept is associated  
 1189 to one and only one code list.

1190  
 1191 To support this use case, the following have to be considered:

- 1192 • **Independent Codelists per variant:** Having each variant in a separate  
 1193 `Codelist` facilitates the maintenance and allows keeping the original codes,  
 1194 even if different versions of the classification have the same code for different  
 1195 concepts. For example, in ISIC Rev. 4 the code "A" represents "Agriculture,  
 1196 forestry and fishing", while in ISIC 3.1 "A" means "Agriculture, hunting and  
 1197 forestry".
- 1198 • **Prefixing Code Ids:** When extending `Codelists`, the reference to an extension  
 1199 `Codelist` may contain a prefix. If a prefix is provided, this prefix will be applied  
 1200 to all the codes in the `Codelist` before they are imported into the extended  
 1201 `Codelist`. In this case, the reference to `CL_ISIC4` includes a prefix of  
 1202 "ISIC4\_" and the reference to `ISIC3` includes "ISIC3\_", so the resulting  
 1203 `Codelist` will have no conflict for the "A" items which will become "ISIC3\_A"  
 1204 and "ISIC4\_A".
- 1205 • **Including / Excluding Specific Codes:** As explained above, there will be  
 1206 independent DFs/PAs with specific `Constraint` attached, in order to keep the  
 1207 proper items according to the variant in use by each data provider.

1208 For example, assuming:

- 1209 • `DSD DSD_EXDU` contains a `Dimension: ACTIVITY` enumerated by  
 1210 `CL_ACTIVITY`.
- 1211 • `CL_ACTIVITY` has no items and is extended by:
- 1212 • `CL_ISIC4, prefix="ISIC4_"`
- 1213 • `CL_ISIC3, prefix="ISIC3_"`
- 1214 • `CL_NACE2, prefix="NACE2_"`
- 1215 • `CL_AGGR, prefix="AGGR_"`
- 1216 • **Dataflow DF1, with a `DataConstraint CC_NACE2, CubeRegion` for `ACTIVITY`  
 1217 and `Value="NACE2_%"`**
- 1218 • **Dataflow DF2, with a `DataConstraint CC_ISIC3, CubeRegion` for `ACTIVITY`  
 1219 and `Value="ISIC3_%"`**
- 1220 • **Dataflow DF3, with a `DataConstraint CC_ISIC4, CubeRegion` for `ACTIVITY`  
 1221 and `Value="ISIC4_%"`, `Value="AGGR_TOTAL"`, `Value="AGGR_Z"`**

1222  
 1223 The discriminated unions are achieved, by requesting any of the above `Dataflows`  
 1224 with `references="all"` and `detail="referencepartial"`: returns  
 1225 `CL_ACTIVITY` with the corresponding extensions resolved and the

1226 DataConstraint, referencing the Dataflow, applied. Thus, the CL\_ACTIVITY will  
1227 only include Codes prefixed according to the Dataflow, i.e.:

- 1228 • Prefix "NACE2\_%" for DF1;
- 1229 • Prefix "ISIC3\_%" for DF2;
- 1230 • Prefix "ISIC4\_%" for DF3; note that Codes "AGGR\_TOTAL" and "AGGR\_Z" are  
1231 also included in this case.

1232

## 1233 **6.2 Linking Hierarchies**

1234 To facilitate the usage of Hierarchy within other SDMX Artefacts, the  
1235 HierarchyAssociation is defined to link any Hierarchy with any  
1236 IdentifiableArtefact within a specific context.

1237

1238 The HierarchyAssociation is a simple Artefact operating like a  
1239 Categorisation. The former specifies three references:

- 1240 • The link to a Hierarchy;
- 1241 • The link to the IdentifiableArtefact that the Hierarchy is linked (e.g., a  
1242 Dimension);
- 1243 • The link to the context that the linking is taking place (e.g., a DSD).

1244 As an example, let's assume:

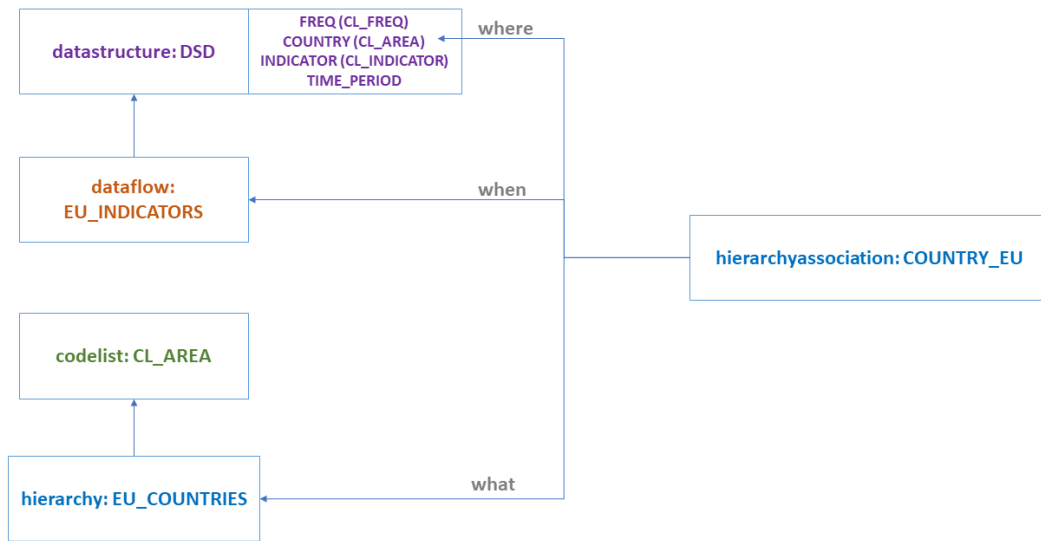
1245 • A DSD with a COUNTRY Dimension that uses Codelist CL\_AREA as  
1246 representation.

1247 • A Hierarchy (e.g., EU\_COUNTRIES) that builds a hierarchy for the CL\_AREA  
1248 Codelist.

1249 • In order to use this Hierarchy for data of a Dataflow (e.g., EU\_INDICATORS),  
1250 we need to build the following HierarchyAssociation:

- 1251 • Links to the Hierarchy **EU\_COUNTRIES (what is associated?)**
- 1252 • Links to the Dimension **COUNTRY (where is it associated?)**
- 1253 • Links to the context: Dataflow **EU\_INDICATORS (when is it**  
1254 **associated?)**

1255 • The above are also shown in the schematic below:



- 1256 •
- 1257 •
- 1258 •

**Figure 15: Hierarchy Association**

1259 **7 Geospatial information support**

1260 SDMX recognizes that statistics refers to units or facts sited in places or areas that  
 1261 may be referenced to geodesic coordinates. This section presents the technical  
 1262 specifications to "geo-reference" those objects and facts in SDMX, by establishing  
 1263 ways to make relations to geographic features over the Earth using a defined  
 1264 coordinates system.

1265  
 1266 SDMX can support three different ways for referencing geospatial data:

- 1267 1. Indirect Reference to Geospatial Information. Including a link to an external file  
 1268 containing the geospatial information. This is the only backwards compatible  
 1269 approach. Since this representation of geospatial information is not included  
 1270 inside the data message, the main use case would be connecting  
 1271 dissemination systems for making use of external tools, like GIS software.
- 1272 2. Geographic Coordinates. Including the coordinates of a specific geospatial  
 1273 feature as a set of coordinates. This is suitable for any statistical information  
 1274 that needs to be georeferenced especially for the exchange of microdata.
- 1275 3. A Geographic Codelist. Includes a type of Codelist, listing predefined  
 1276 geographies that are represented by geospatial information. These  
 1277 geographies could be administrative (including administrative boundaries or  
 1278 enumeration areas), lines, points, or gridded geographies. Regardless, the  
 1279 geospatial information used to represent the geography would contain both  
 1280 dimensions and/or attributes; therefore, representing an advantage for the data  
 1281 modellers as it provides a clear way to identify those dimensions developing a  
 1282 "Geospatial" role.

1283 **7.1 Indirect Reference to Geospatial Information.**

1284 This option provides a way to include external references to geospatial information  
 1285 through a file containing it. The external content may be geographical or thematic maps  
 1286 with different levels of precision. All the processing of geospatial data is made through  
 1287 external applications that can interpret the information in different formats.  
 1288

1289 The reference to the external files containing geospatial information is made using  
 1290 some recommended SDMX Attributes, with the following content:

- 1291 • **GEO\_INFO\_TEXT**. A description of the kind of information being referenced.
- 1292 • **GEO\_INFO\_URL**. A URL which points to the resource containing the referred  
 1293 geospatial information. The resource might be a file with static geodesic  
 1294 information or a web service providing dynamic construction of geometries.
- 1295 • **GEO\_INFO\_TYPE**. Coded information describing a standard format of the file  
 1296 that contains the geospatial information. The format types are taken from the list  
 1297 of Format descriptions for Geospatial Data managed by the US Library of the  
 1298 Congress ([https://www.loc.gov/preservation/digital/formats/fdd/gis\\_fdd.shtml](https://www.loc.gov/preservation/digital/formats/fdd/gis_fdd.shtml)).  
 1299 Allowed types in SDMX are listed in the **Geographical Formats** code list  
 1300 (**CL\_GEO\_FORMATS**). Examples of the codes contained in the document are:

• Code	• Description
• <b>GML</b>	• Geography Markup Language

• <b>GeoTIFF</b>	• GeoTIFF
• <b>KML_2_2</b>	• KML Version 2.2
• <b>GEOJSON_1_1</b>	• GeoJSON Version 1.1

1301  
1302  
1303  
1304  
1305  
1306  
1307  
1308

Depending on the intended use, these attributes may be attached at the dataflow level, the series level or the observation level.

The indirect reference to geospatial information in SDMX is recommended to be used for dissemination purposes, where the statistical information is complemented by geographical representations of places or regions.

## 1309 **7.2 Geographic Coordinates**

1310 This option to represent geospatial information in SDMX provides an efficient way for  
1311 including geographic information with different levels of granularity, due to its flexibility.  
1312 Geospatial information is represented using the `GeospatialInformation` type, as  
1313 defined in the data types of the SDMX Information Model. A "GEO\_FEATURE\_SET" role  
1314 should be assigned to any Component of this type.

1315  
1316 The `GeospatialInformation` data type can be assigned to a `Dimension`,  
1317 `DataAttribute`, `MetadataAttribute` or a `Measure` with the  
1318 "GEO\_FEATURE\_SET" role assigned; it can be included in a dataset or metadataset.

1319  
1320 Any `Component` used for representing a Geographical Feature Set, i.e., used to  
1321 describe geographical characteristics, must have a "GEO\_FEATURE\_SET" role. Its  
1322 `Representation` would be of `textType="GeospatialInformation"`. The  
1323 `GeospatialInformation` type is not intended to replace standard geospatial  
1324 information formats, but instead provide a simple way to describe precise geographical  
1325 characteristics in SDMX data sets agnostic of any particular geospatial software  
1326 product or use case.

1327  
1328 The `GeospatialInformation` type should be used to describe geographical  
1329 features like points (e.g., locations of places, landmarks, buildings, etc.), lines (e.g.,  
1330 rivers, roads, streets, etc.), or areas (e.g., geographical regions, countries, islands,  
1331 land lots, etc.). A mix of different features is possible too, e.g., combining polygons and  
1332 geographical points to describe a country and the location of its capital.

1333  
1334 The components that conform to the structure of the `GeospatialInformation` type  
1335 are:

- 1336 • `X_COORDINATE`: The horizontal (longitude) value of a pair of coordinates  
1337 expressed in the Coordinate Reference System (CRS), mandatory.
- 1338 • `Y_COORDINATE`: The vertical value (latitude) of a pair of coordinates expressed  
1339 in the CRS units, mandatory.
- 1340 • `ALT`: The height (altitude) from the reference surface is expressed in meters,  
1341 optional.

1342       • CRS: The code of the Coordinate Reference System is used to reference the  
1343 coordinates in the flow, optional.

1344       The code of the CRS will be as it appears in the EPSG Geodetic Parameter  
1345 Registry (<http://www.epsg-registry.org/>) maintained by the International  
1346 Association of Oil and Gas Producers. If this element is omitted, by default, the  
1347 CRS will be the World Geodetic System 1984 (WGS 84, EPSG:4326).

1348       • PRECISION: Precision of the coordinates, expressing the possible deviation in  
1349 meters from the exact geodesic point, optional.

1350       This component is only allowed if the CRS is specified too. If omitted, it will be  
1351 interpreted as limited it to the expected measurement accuracy (e. g. a  
1352 standard GPS has an accuracy of ~ 10m).

1353       • GEO\_DESCRIPTION: Text for additional information about the place,  
1354 geographical feature, or set of geographical features, optional.

1355  
1356 Geographical features (GEO\_FEATURES) are collections of geographical features  
1357 intended to be used to represent geographical areas like countries, regions, etc., or  
1358 objects, like water bodies (e. g. rivers, lakes, oceans, etc.), roads (streets, highways,  
1359 etc.), hospitals, schools, and the like. They are represented in the following way:

1360  
1361 **(GEO\_FEATURE, GEO\_FEATURE) : GEO\_DESCRIPTION**  
1362

1363       • GEO\_FEATURE represents a set of points defining a feature following the  
1364 ISO/IEC 13249-3:2016 standard to conform Well-known Text (WKT) for the  
1365 representation of geometries in a format defined in the following way:

1366  
1367 **GEOMETRY\_TYPE (GEOMETRY\_REP)**  
1368

1369       • GEOMETRY\_TYPE: A string with a closed vocabulary defining the type of the  
1370 geometry that represents a geographical component of the GEO\_FEATURES  
1371 collection, mandatory.

1372  
1373 Three types are allowed:  
1374       1. **Point**, a specific geodesic point, like the centroid of a city or a hospital.  
1375       It is represented with the string "POINT"  
1376       2. **Line**, a feature defining a line like a road, a river, or similar. It is  
1377       represented with the string "LINESTRING"  
1378       3. **Area**, a polygon defining a closed area. It is represented with the string  
1379       "POLYGON"

1380  
1381 If the GEOMETRY\_REP is going to be including the height (ALT) component, a  
1382 "Z" must be added after the string qualifying the GEOMETRY\_TYPE. In this way,  
1383 we will have: "POINT Z", "LINESTRING Z" and "POLYGON Z"

1384  
1385 Other feature types (e.g. Triangular irregular networks, "TIN") are not supported  
1386 yet directly, except grids that are detailed in 7.3.

1387  
1388       • GEOMETRY\_REP: Representation of each of the types The way to represent  
1389 each GEO\_FEATURE\_TYPE will be:

- 1390           ○ A point (POINT): "COORDINATES"  
 1391           ○ A line (LINESTRING): "COORDINATES, COORDINATES, ..."  
 1392           ○ An area (POLYGON): "(COORDINATES, COORDINATES, ...),  
 1393           (COORDINATES, COORDINATES, ...)"

1394 Where:

- 1395       • COORDINATES: Represents an individual set of coordinates composed by the  
 1396       X\_COORDINATE (X), Y\_COORDINATE (Y), and ALT (Z) in the following  
 1397       way "X Y Z" or "X Y" defining a single point of the polygon. Altitude is to be  
 1398       reported in meters.  
 1399

1400 In an expanded way, GEO\_FEATURE may be represented in the following ways:

1401  
 1402 POINT (X\_COORDINATE Y\_COORDINATE): GEO\_DESCRIPTION  
 1403 POINT Z (X\_COORDINATE Y\_COORDINATE ALT): GEO\_DESCRIPTION  
 1404 LINESTRING (X\_COORDINATE Y\_COORDINATE, X\_COORDINATE  
 1405 Y\_COORDINATE, ...): GEO\_DESCRIPTION  
 1406 LINESTRING Z (X\_COORDINATE Y\_COORDINATE ALT, X\_COORDINATE  
 1407 Y\_COORDINATE ALT, ...): GEO\_DESCRIPTION  
 1408 POLYGON ((X\_COORDINATE Y\_COORDINATE, X\_COORDINATE  
 1409 Y\_COORDINATE, ...), (X\_COORDINATE Y\_COORDINATE, X\_COORDINATE  
 1410 Y\_COORDINATE, ...), ...): GEO\_DESCRIPTION  
 1411 POLYGON Z ((X\_COORDINATE Y\_COORDINATE ALT, X\_COORDINATE  
 1412 Y\_COORDINATE ALT, ...), (X\_COORDINATE Y\_COORDINATE ALT,  
 1413 X\_COORDINATE Y\_COORDINATE ALT, ...), ...): GEO\_DESCRIPTION  
 1414

1415 An example of how GEO\_FEATURES may be represented in an expanded way would  
 1416 be:

1417  
 1418 (POLYGON Z ((X\_COORDINATE Y\_COORDINATE ALT, X\_COORDINATE  
 1419 Y\_COORDINATE ALT, ...), (X\_COORDINATE Y\_COORDINATE ALT,  
 1420 X\_COORDINATE Y\_COORDINATE ALT, ...), ...), POLYGON Z  
 1421 ((X\_COORDINATE Y\_COORDINATE ALT, X\_COORDINATE Y\_COORDINATE  
 1422 ALT, ...), (X\_COORDINATE Y\_COORDINATE ALT, X\_COORDINATE  
 1423 Y\_COORDINATE ALT, ...), ...), POLYGON Z ((X\_COORDINATE  
 1424 Y\_COORDINATE ALT, X\_COORDINATE Y\_COORDINATE ALT, ...),  
 1425 (X\_COORDINATE Y\_COORDINATE ALT, X\_COORDINATE Y\_COORDINATE ALT,  
 1426 ...), ...), ...): GEO\_DESCRIPTION  
 1427

1428 Accordingly to this logic, an example of an expanded expression representing a value  
 1429 of the GeospatialInformation may be the following:

1430  
 1431 "CRS, PRECISION: {(POLYGON Z ((X\_COORDINATE Y\_COORDINATE ALT,  
 1432 X\_COORDINATE Y\_COORDINATE ALT, ...), (X\_COORDINATE Y\_COORDINATE  
 1433 ALT, X\_COORDINATE Y\_COORDINATE ALT, ...), ...), POLYGON Z  
 1434 ((X\_COORDINATE Y\_COORDINATE ALT, X\_COORDINATE Y\_COORDINATE  
 1435 ALT, ...), (X\_COORDINATE Y\_COORDINATE ALT, X\_COORDINATE  
 1436 Y\_COORDINATE ALT, ...), ...), POLYGON Z ((X\_COORDINATE  
 1437 Y\_COORDINATE ALT, X\_COORDINATE Y\_COORDINATE ALT, ...),  
 1438 (X\_COORDINATE Y\_COORDINATE ALT, X\_COORDINATE Y\_COORDINATE ALT,  
 1439 ...), ...), ...): GEO\_DESCRIPTION}, {(POLYGON Z ((X\_COORDINATE  
 1440 Y\_COORDINATE ALT, X\_COORDINATE Y\_COORDINATE ALT, ...),  
 1441 (X\_COORDINATE Y\_COORDINATE ALT, X\_COORDINATE Y\_COORDINATE ALT,

1442 ...), ...) , POLYGON Z ((X\_COORDINATE Y\_COORDINATE ALT,  
 1443 X\_COORDINATE Y\_COORDINATE ALT, ...) , (X\_COORDINATE Y\_COORDINATE  
 1444 ALT, X\_COORDINATE Y\_COORDINATE ALT, ...) , ...) , POLYGON Z  
 1445 ((X\_COORDINATE Y\_COORDINATE ALT, X\_COORDINATE Y\_COORDINATE  
 1446 ALT, ...) , (X\_COORDINATE Y\_COORDINATE ALT, X\_COORDINATE  
 1447 Y\_COORDINATE ALT, ...) , ...) : GEO\_DESCRIPTION} , ...:  
 1448 GEO\_DESCRIPTION"

1449

1450 Validation rules must be added to the XML Schema to ensure the integrity of the  
 1451 specification according to the proposed syntax.

1452

### 1453 **7.3 A Geographic Codelist**

1454 Geography is represented by geospatial information. Within SDMX, geospatial  
 1455 information is conceptually represented by the "GEO\_FEATURE\_SET"  
 1456 role/specification. This approach uses a specialized form of SDMX Codelist, named  
 1457 "GeoCodelist", which is a Codelist containing the Geography used to demarcate the  
 1458 geographic extent. This is implemented in two ways:

- 1459 1. *Geographic*. It is a regular codelist that has been extended to add a  
 1460 geographical feature set to each of its items, typically, this would include all  
 1461 types of administrative geographies;
- 1462 2. *Grid*. As a codelist that has defined a geographical grid composed of cells  
 1463 representing regular squared portions of the Earth.

1464 A *GeoCodelist* is a Codelist as defined in the SDMX Information Model that has the  
 1465 *GeoType* property added. *GeoType* can take one of two values "Geographic" or  
 1466 "GeoGrid".

1467

1468 "Geographic" corresponds to the first way to implement a *GeoCodelist*. When the  
 1469 *GeoCodelist* includes a *GeoType*="Geographic" property, a *GeoFeatureSet*  
 1470 property is added to each of the items in the code list to implement a *Geographic*  
 1471 *GeoCodelist*.

1472

1473 On the other hand, when *GeoType*="GeoGrid" it is defining a gridded  
 1474 *GeoCodelist*, and it is necessary to add a grid definition to the Codelist identifier  
 1475 using the *gridDefinition* property. The components needed to define a  
 1476 geographical grid are the following:

- 1477 • *CRS*: The code of the Coordinate Reference System is used to reference the  
 1478 coordinates in the flow, optional. The code of the CRS will be as it appears in the  
 1479 EPSG Geodetic Parameter Registry (<http://www.epsg-registry.org/>) maintained  
 1480 by the International Association of Oil and Gas Producers. If this component is  
 1481 omitted, by default the CRS will be the World Geodetic System 1984 (WGS 84,  
 1482 EPSG:4326).
- 1483 • *REFERENCE\_CORNER*: A code composed of two characters to define the position  
 1484 of the coordinates that will be used as a starting reference to locate the cells. The  
 1485 possible values of this code can be *UL* (Upper Left), *UR* (Upper Right), *LL* (Lower  
 1486 Left), or *LR* (Lower Right). If this component is omitted the value *LL* (Lower Left)  
 1487 will be taken by default. This element is optional.

- 1488 • **REFERENCE\_COORDINATES:** Represents the starting point to reference the cells
- 1489 of the grid, accordingly to the `CRS` and the `REFERENCE_CORNER`. It is represented
- 1490 by an individual set of coordinates composed by the `X_COORDINATE` (X) and
- 1491 `Y_COORDINATE` (Y) in the following way "X, Y". This element is mandatory if
- 1492 `GEO_STD` is omitted.
  
- 1493 • **CELL\_WIDTH:** The size in meters of a horizontal side of the cells in the grid. This
- 1494 element is mandatory if `GEO_STD` is omitted.
  
- 1495 • **CELL\_HEIGHT:** The size in meters of a vertical side of the cells in the grid. This
- 1496 element is mandatory if `GEO_STD` is omitted .
  
- 1497 • **GEO\_STD:** A restricted text value expressing that the cells in the grid will provide
- 1498 information about matching codes existing in another reference system that
- 1499 establishes a mechanism to define the grid. This element is optional.

1500 Accepted values for this component are included in the Geographical Grids  
 1501 Codelist (`CL_GEO_GRIDS`). Examples contained in the mentioned document  
 1502 are:  
 1503

Value	Description
GEOHASH	GeoHash
GEOREF	World Geographic Reference System
MGRS	Military Grid Reference System
OLC	Open Location Code / Plus Code
QTH	Maidenhead Locator System /QTH Locator / IAURU Locator
W3W	What3words™
WOEID	Where On Earth Identifier

1504  
 1505 The `GRID_DEFINITION` element will contain a regular expression string  
 1506 corresponding to the following format:  
 1507 "`CRS: REFERENCE_CORNER; REFERENCE_COORDINATES; CELL_WIDTH,`  
 1508 `CELL_HEIGHT: GEO_STD`"

1509  
 1510 In an expanded way we would have:  
 1511 "`CRS:REFERENCE_CORNER; X_COORDINATE, Y_COORDINATE; CELL_WIDTH,`  
 1512 `CELL_HEIGHT: GEO_STD`"

1513  
 1514 If the grid will be fully adhering to a standard declared in the `GEO_STD`, the definition  
 1515 of each code in the code list will be optional. In other case, each item in the code list  
 1516 must be assigned to one cell in the grid, which is made by adding the `GEO_CELL`  
 1517 element to each item of the code list that will contain a regular expression string  
 1518 composed of the following components:

- 1519 • **GEO\_COL:** The number of the column in the grid starting by zero.
- 1520 • **GEO\_ROW:** The number of the row in the grid starting by zero.
- 1521 • **GEO\_TAG:** An optional text to include additional information to the cell.

- 1522     • GEO\_CELL will have values with the following format: "GEO\_COL, GEO\_ROW:  
1523       GEO\_TAG"
- 1524     When using a gridded `GeoCodeList` we may use the `GEO_TAG` to integrate the cells  
1525     in the grid to the codes used by other standard defined grids. As an example, `GEO_TAG`  
1526     can take the values of the Open Location Codes, GeoHash, etc. If this is done, the  
1527     `GEO_STD` component must have been added to the definition of the grid. If the  
1528     `GEO_STD` is omitted, the `GEO_TAG` contents will be taken just as free text.
- 1529     •

1530 **8 Maintenance Agencies and Metadata Providers**

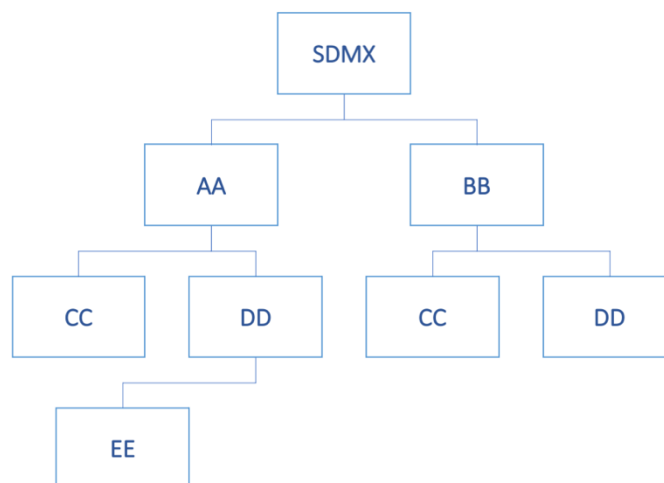
1531 All structural metadata in SDMX is owned and maintained by a maintenance agency  
 1532 (Agency identified by `agencyID` in the schemas). Similarly, all reference metadata  
 1533 (i.e., `MetadataSets`) is owned and maintained by a metadata provider organisation  
 1534 (`MetadataProvider` identified by `metadataProviderID` in the schemas). It is vital to  
 1535 the integrity of the structural metadata that there are no conflicts in `agencyID` and  
 1536 `metadataProviderID`. In order to achieve this, SDMX adopts the following rules:

- 1537
- 1538 1. Agencies are maintained in an `AgencyScheme` (which is a sub class of  
 1539 `OrganisationScheme`); `Metadata Providers` are maintained in a  
 1540 `MetadataProviderScheme`.
  - 1541 2. The maintenance agency of the `Agency/Metadata Provider Scheme` must also  
 1542 be declared in a (different) `AgencyScheme`.
  - 1543 3. The "top-level" agency is SDMX and this agency scheme is maintained by  
 1544 SDMX.
  - 1545 4. Agencies registered in the top-level scheme can themselves maintain a single  
 1546 `AgencyScheme` and a single `MetadataProviderScheme`. SDMX is an  
 1547 agency in the SDMX `AgencyScheme`. Agencies in any `AgencyScheme` can  
 1548 themselves maintain a single `AgencyScheme` and so on.
  - 1549 5. The `AgencyScheme` and `MetadataProvideScheme` cannot be versioned  
 1550 and thus have a fixed version set to '1.0'.
  - 1551 6. There can be only one `AgencyScheme` maintained by any one Agency. It has  
 1552 a fixed Id of 'AGENCIES'. Similarly, only one `MetadataProvideScheme` is  
 1553 maintained by one Agency and has a fixed id of 'METADATA\_PROVIDERS'.
  - 1554 7. The format of the agency identifier is `agencyId.agencyID` etc. The top-level  
 1555 agency in this identification mechanism is the agency registered in the SDMX  
 1556 agency scheme. In other words, SDMX is not a part of the hierarchical ID  
 1557 structure for agencies. SDMX is, itself, a maintenance agency.

1558  
 1559 This supports a hierarchical structure of `agencyID`.

1560  
 1561 An example is shown below.

1562



1563

1564 **Figure 16: Example of Hierarchic Structure of Agencies**

1565 Each agency is identified by its full hierarchy excluding SDMX.

1566  
1567 The XML representing this structure is shown below.

```

1568
1569 <str:AgencySchemes>
1570   <str:AgencyScheme agencyID="SDMX" id="AGENCIES">
1571     <com:Name xml:lang="en">Top-level Agency Scheme</com:Name>
1572     <str:Agency id="AA">
1573       <com:Name xml:lang="en">AA Name</com:Name>
1574     </str:Agency>
1575     <str:Agency id="BB">
1576       <com:Name xml:lang="en">BB Name</com:Name>
1577     </str:Agency>
1578   </str:AgencyScheme>
1579
1580   <str:AgencyScheme agencyID="AA" id="AGENCIES">
1581     <com:Name xml:lang="en">AA Agencies</com:Name>
1582     <str:Agency id="CC">
1583       <com:Name xml:lang="en">CC Name</com:Name>
1584     </str:Agency>
1585     <str:Agency id="DD">
1586       <com:Name xml:lang="en">DD Name</com:Name>
1587     </str:Agency>
1588   </str:AgencyScheme>
1589
1590   <str:AgencyScheme agencyID="BB" id="AGENCIES">
1591     <com:Name xml:lang="en">BB Agencies</com:Name>
1592     <str:Agency id="CC">
1593       <com:Name xml:lang="en">CC Name</com:Name>
1594     </str:Agency>
1595     <str:Agency id="DD">
1596       <com:Name xml:lang="en">DD Name</com:Name>
1597     </str:Agency>
1598   </str:AgencyScheme>
1599
1600   <str:AgencyScheme agencyID="AA.DD" id="AGENCIES">
1601     <com:Name xml:lang="en">AA.DD Agencies</com:Name>
1602     <str:Agency id="EE">
1603       <com:Name xml:lang="en">EE Name</com:Name>
1604     </str:Agency>
1605   </str:AgencyScheme>
1606 </str:AgencySchemes>
1607

```

1608 **Figure 17: Example Agency Schemes Showing a Hierarchy**

1609 Examples of Structure definitions that show how Agencies are used, are presented  
1610 below:

```

1611 <str:Codelist agencyID="SDMX" id="CL_FREQ" version="1.0.0"
1612   urn="urn:sdmx:org.sdmx.infomodel.codelist.Codelist=SDMX:CL_FREQ(1.0.0)">
1613   <com:Name xml:lang="en">Frequency</com:Name>
1614 </str:Codelist>
1615 <str:Codelist agencyID="AA" id="CL_FREQ" version="1.0.0"
1616   urn="urn:sdmx:org.sdmx.infomodel.codelist.Codelist=AA:CL_FREQ(1.0.0)">
1617   <com:Name xml:lang="en">Frequency</com:Name>
1618 </str:Codelist>
1619 <str:Codelist agencyID="AA.CC" id="CL_FREQ" version="1.0.0"
1620   urn="urn:sdmx:org.sdmx.infomodel.codelist.Codelist=AA.CC:CL_FREQ(1.0.0)">
1621   <com:Name xml:lang="en">Frequency</com:Name>
1622 </str:Codelist>
1623 <str:Codelist agencyID="BB.CC" id="CL_FREQ" version="1.0.0"
1624   urn="urn:sdmx:org.sdmx.infomodel.codelist.Codelist=BB.CC:CL_FREQ(1.0.0)">

```

```
1625 <com:Name xml:lang="en">Frequency</com:Name>  
1626 </str:Codelist>
```

1627 **Figure 18: Example Showing Use of Agency Identifiers**

1628

1629 Each of these maintenance agencies has a `Codelist` with an identical id 'CL\_FREQ'.

1630 However, each is uniquely identified by means of the hierarchic agency structure.

1631 **9 Concept Roles**

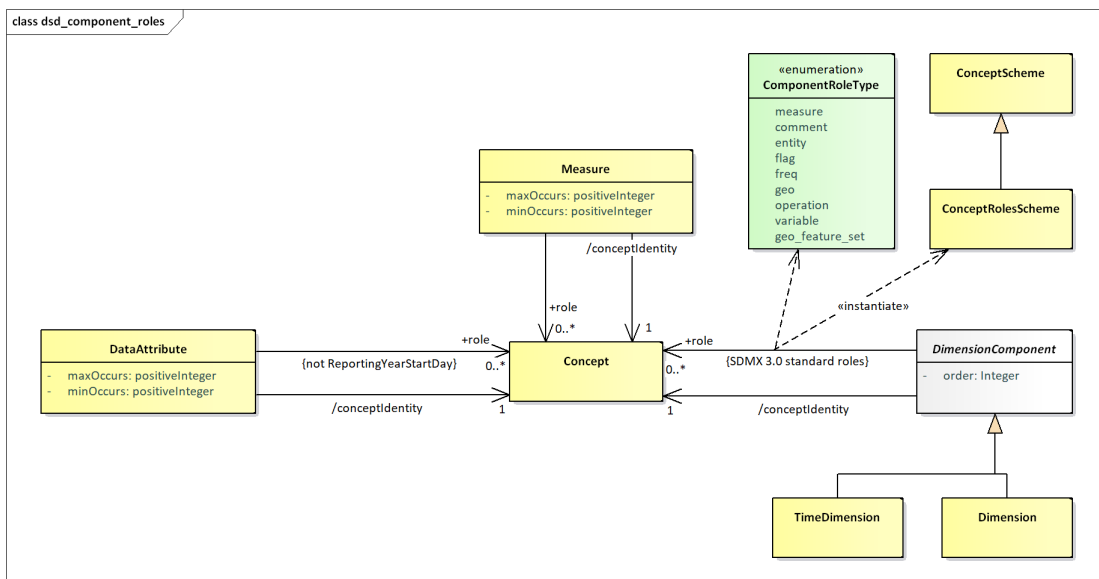
1632 **9.1 Overview**

1633 The DSD Components of Dimension and Attribute can play a specific role in the DSD  
 1634 and it is important to some applications that this role is specified. For instance, the  
 1635 following roles are some examples:

- 1636 • **Frequency** – in a data set the content of this Component contains information on  
 1637 the frequency of the observation values.
- 1638 • **Geography** – in a data set the content of this Component contains information  
 1639 on the geographic location of the observation values.

1640 **9.2 Information Model**

1641 The Information Model for this is shown below:  
 1642



1643 **Figure 19: Information Model Extract for Concept Role**  
 1644

1645 It is possible to specify zero or more concept roles for a Dimension, Measure and Data  
 1646 Attribute. The Time Dimension has explicitly defined roles and cannot be further  
 1647 specified with additional concept roles.

1648 **9.3 Technical Mechanism**

1649 The mechanism for maintain and using concept roles is as follows:

- 1650 4. A standard Concept Scheme maintained in the Global Registry, with the  
 1651 following identification: `SDMX:CONCEPT_ROLES(1.0.0)`, shall include  
 1652 the default roles, specified by the SDMX SWG (as detailed in 9.5).
- 1653 5. Any recognized Agency can have a concept scheme that contains  
 1654 concepts that identify concept roles. Indeed, from a technical  
 1655 perspective any agency can have more than one of these schemes,  
 1656 though this is not recommended.

- 1657 6. The concept scheme that contains the "role" concepts can contain  
1658 concepts that do not play a role.
- 1659 7. There is no explicit indication on the Concept whether it is a 'role'  
1660 concept.
- 1661 8. Therefore, any concept in any concept scheme is capable of being a  
1662 'role' concept.
- 1663 9. It is the responsibility of Agencies to ensure their community knows  
1664 which concepts in which concept schemes play a 'role' and the  
1665 significance and interpretation of this role. In other words, such  
1666 concepts must be known by applications, there is no technical  
1667 mechanism that can inform an application on how to process such a  
1668 'role'.
- 1669 10. If the concept referenced in the Concept Identity in a DSD component  
1670 (Dimension, Measure Dimension, Attribute) is contained in the concept  
1671 scheme containing concept roles then the DSD component could play  
1672 the role implied by the concept, if this is understood by the processing  
1673 application.
- 1674 11. If the concept referenced in the Concept Identity in a DSD component  
1675 (Dimension, Measure Dimension, Attribute) is not contained in the  
1676 concept scheme containing concept roles, and the DSD component is  
1677 playing a role, then the concept role is identified by the Concept Role in  
1678 the schema.

#### 1679 **9.4 SDMX-ML Examples in a DSD**

1680 The standard roles Concept Scheme, is still a normal Concept Scheme, thus it may be  
1681 used also for the concept identity of a Component, e.g., the 'FREQ':

```
1682 <str:Dimension id="FREQ">
1683   <str:ConceptIdentity>urn:sdmx:org.sdmx.infomodel.conceptscheme.Concept=
1684     SDMX:CONCEPT_ROLES(1.0.0).FREQ</str:ConceptIdentity>
1685 </str:Dimension>
```

1686  
1687 Given this is the standard roles Concept Scheme, any application should interpret the  
1688 above Dimension to have the role of Frequency.

1689  
1690 Using a Concept Scheme that is not the standard roles Concept Scheme where it is  
1691 required to assign a role using the standard roles Concept Scheme. Again, FREQ is  
1692 chosen as the example.

```
1693 <str:Dimension id="FREQ">
1694   <str:ConceptIdentity>urn:sdmx:org.sdmx.infomodel.conceptscheme.Concept=
1695     SDMX:CONCEPTS(1.0.0).FREQ</str:ConceptIdentity>
1696   <str:ConceptRole>urn:sdmx:org.sdmx.infomodel.conceptscheme.Concept=
1697     SDMX:CONCEPT_ROLES(1.0.0).FREQ</str:ConceptRole>
1698 </str:Dimension>
```

1699  
1700 This explicitly states that this Dimension is playing a role identified by the FREQ  
1701 concept in the standard roles Concept Scheme. Again, the application must interpret  
1702 this as a Frequency role.

1703  
1704 In other cases where a role from a non-standard roles Concept Scheme is used, then  
1705 the application has to know how to interpret the provided roles, e.g., like in the case  
1706 below:

```

1707 <str:Dimension id="FREQ">
1708   <str:ConceptIdentity>urn:sdmx:org.sdmx.infomodel.conceptscheme.Concept=
1709     SDMX:CONCEPTS(1.0.0).FREQ</str:ConceptIdentity>
1710   <str:ConceptRole>urn:sdmx:org.sdmx.infomodel.conceptscheme.Concept=
1711     SDMX:MY_CONCEPT_ROLES(1.0.0).FREQ</str:ConceptRole>
1712 </str:Dimension>

```

1713  
 1714 This is all that is required for interoperability within a community. Having a standard  
 1715 roles Concept Scheme, maintained by the SDMX SWG, allows the SDMX community  
 1716 to have a common understanding of the roles, while also being able to extend the roles  
 1717 in bilateral (or multilateral) agreements, by maintaining their own roles Concept  
 1718 Scheme. This will then ensure there is interoperability between systems that  
 1719 understand the use of these concepts.

1720  
 1721 Note that each of the Components (Data Attribute, Measure, Dimension, Time  
 1722 Dimension) has a mandatory identity association (Concept Identity) and if this Concept  
 1723 also identifies the role then it must be interpreted accordingly.

1724  
 1725 In order for these roles to be extensible and also to enable user communities to  
 1726 maintain community-specific roles, the roles are maintained in a controlled vocabulary  
 1727 which is implemented in SDMX as Concepts in a Concept Scheme. The Component  
 1728 optionally references this Concept if it is required to declare the role explicitly. Note  
 1729 that a Component can play more than one role and therefore multiple "role" concepts  
 1730 can be referenced.

### 1731 **9.5 SDMX standard roles Concept Scheme**

1732 As of SDMX 3.0, there is a predefined Concept Scheme, with a set of Concepts that  
 1733 are considered the standard roles for SDMX. Beyond that, a user is free to add other  
 1734 roles, using custom Concept Schemes. This predefined Concept Scheme is the result  
 1735 of the SWG guidelines for Concept Roles, plus that for Measure, and includes the  
 1736 following Concepts:

COMMENT	Comment	Descriptive text which can be attached to data or metadata.
ENTITY	Entity	Describes the subject of the data set (e.g., a country).
FLAG	Flag	Coded attribute in a data set that represents qualitative information for the cell or partial key (e.g. series) value.
FREQ	Frequency	Time interval at which the source data are collected.
GEO	Geographical	Geographic area to which the measured statistical phenomenon relates.
OPERATION	Statistical operation	Signifies statistical operations have been done on the observations.
VARIABLE	Variable	Characteristic of a unit being observed that may assume more than one of a set of values to which a numerical measure or a category from a classification can be assigned.
MEASURE	Measure	Used for emulating the functionality of the deprecated MeasureDimension.

GEO_FEATU RE_SET	Geographical Feature Set	Georeferencing information to describe the location or the shape of a statistical unit, recognizable object or geographical area.
PRIMARY	Primary Measure	Used for backwards compatibility with SDMX 2.1 and back, or when the "Primary Measure" concept is needed.

1738

## 1739 **10 Constraints**

### 1740 **10.1 Introduction**

1741 Constraints are used as a way to restrict what data can be reported, or to report what  
1742 data exists in a given context. There are three types of Constraint, which serve  
1743 different purposes

- 1744
- 1745 • Availability Constraint
- 1746 • Dimension Constraint
- 1747 • Reporting Constraints
- 1748

1749 An Availability Constraint defines the data that exists in the context of a data query.  
1750 They form part of the response message from the Availability REST API. Availability  
1751 Constraints are dynamically generated by a system based on the data that exists and  
1752 the query context. Availability Constraints are therefore not Identifiable structures (they  
1753 have no URN).

1754

1755 A Dimension Constraint is a property of a Dataflow, they are used to fix the Dimensions  
1756 that they use in the Data Structure Definition which they use. Dimension Constraints  
1757 enable Data Structure Definitions to evolve over time by having new Dimensions  
1758 added, without having to undergo a major version change.

1759

1760 A Reporting Constraint is used to define the set of allowed and/or disallowed values  
1761 that can be reported in a data or metadata set.  
1762

### 1763 **10.2 Availability Constraint**

1764 An Availability Constraint is not a maintained structure, instead it is generated  
1765 dynamically as a response to the availability REST API. The purpose of the Availability  
1766 Constraint is to define the distinct set of values that have data over 1 or more  
1767 Dimensions. Unlike a Data and Metadata Constraint, which can attach to multiple  
1768 Constraining structures (of the same type), an Availability Constraint can only attach  
1769 to only one structure. The attachment defines the context of the response (data exists  
1770 for components in the context of). The subset of Constraining structures the  
1771 Availability Constraint can attach to are:

- 1772 • Data Structure Definition
- 1773 • Dataflow
- 1774 • Provision Agreement
- 1775

### 1776 **10.3 Dimension Constraint**

1777 A Dimension Constraint is a property of a Dataflow; its purpose is to explicitly list the  
1778 Dimensions from the corresponding DSD that are being used by the Dataflow.  
1779

1780 Dimension Constraints were introduced in SDMX 3.1 and are not required for most  
1781 Dataflows where the dataset must always contain the full complement of Dimensions  
1782 as defined by the corresponding DSD. However, for some complex data collections,  
1783 which may span long periods and where the full complement of required Dimensions

1784 are not necessarily known at design time, the DSD is subject to increasing its  
 1785 Dimensionality over time. In this scenario it is possible to define the DSD as an  
 1786 evolving structure, this property tells the user that the DSD can have new Dimensions  
 1787 added without having to undergo a major version change; a DSD at version 1.0.0 for  
 1788 example would be able to add a new Dimension and move to version 1.1.0; a change  
 1789 that would not ordinarily be allowed. A minor version change on the addition of a new  
 1790 Dimension is only possible if the DSD defines itself as an evolving structure. This is  
 1791 a new property of the DSD introduced in version 3.1 to satisfy this use case. The  
 1792 evolving structure property is either true or false, defaulting to false if not specified.  
 1793 Setting the evolving structure property to true requires a major version change, and  
 1794 therefore can only be introduced on an x.0.0 release (e.g. 1.0.0). The evolving  
 1795 structure property can be set to false to indicate that there will be no additional  
 1796 Dimensions added to the Data Structure under the same major version number; setting  
 1797 the evolving structure property to false does not require require a major version change  
 1798 on the Data Structure.

1800 When a Dataflow references a DSD, late binding on the minor release, and the DSD  
 1801 has the evolving structure property set to true, then the Dataflow must contain a  
 1802 Dimension Constraint to protect its Dimensionality from changing over time without a  
 1803 version change.

1805 The Dimension Constraint provides the explicit list of Dimensions that the Dataflow  
 1806 uses from the DSD that it references. This enables the DSD to evolve over time without  
 1807 breaking the compatibility of datasets against the Dataflow.

1809 **Rules for a Dimension Constraint**

- 1810 • A Dataflow must contain a Dimension Constraint if the DSD which it uses states  
 1811 that it is an evolving structure and the Dataflow is late binding on the minor  
 1812 release (latest minor release of a given major version, e.g. 1.0+.0)
- 1813 • The Dimension Constraint can only include Dimensions from the DSD that is  
 1814 referenced by the Dataflow.
- 1815 • A Dimension Constraint can only be changed if the Dataflow undergoes a major  
 1816 version change
- 1817 • Datasets reported against the Dataflow must only contain reported values for  
 1818 the Dimensions specified in the Dimension Constraint.
- 1819 • When exporting data for the Dataflow, the dataset should only include the  
 1820 Dimensions specified by the Dimension Constraint.
- 1821 • When exporting data for the DSD the dataset must contain the full set of  
 1822 Dimensions as specified by the DSD. The tilde '~' character is used to represent  
 1823 a value which is not present due to the Dimension not being included in the  
 1824 corresponding Dataflow.

1826 **Example Datasets with Evolving Structures**

1827 A dataset is built against a Data Structure Definition. The dataset contains data for  
 1828 two Dataflows. Dataflows 'DF\_POP' uses a Dimension Constraint which fixes its  
 1829 Dimensions to `FREQ` and `REF_AREA`. Dataflow 'DF\_POP\_SA' does not reference a  
 1830 Dimension Constraint, and as such includes all Dimensions as specified by the DSD.

1832 The resulting dataset contains values '~' for both the `SEX` and `AGE` Dimension for the  
 1833 series related to `DF_POP`.

Dataflow	FREQ	REF_AREA	SEX	AGE	OBS_VALUE	TIME_PERIOD	UNIT
----------	------	----------	-----	-----	-----------	-------------	------

DF_POP	A	UK	~	~	65	2022	6
DF_POP	A	FR	~	~	50	2022	6
DF_POP_SA	A	UK	M	1	1.2	2022	6

1835

1836

## 1837 **10.4 Reporting Constraints**

1838 A Reporting Constraint is a Maintainable Artefact which restricts the values that can  
1839 be reported in a dataset or metadata set based on one or more inclusion or exclusion  
1840 rules.

1841

1842 A reporting constraint is one of the following concrete types:

- 1843 • Data Constraint
- 1844 • Metadata Constraint

1845

1846

### 1847 **10.4.1 Data Constraint**

1848 A Data Constraint is used to add additional restrictions to the allowable values reported  
1849 in a dataset. Data Constraints can be applied to the follow structures which are  
1850 collectively known as Constrainable structures:

- 1851 • Data Structure Definition
- 1852 • Dataflow
- 1853 • Provision Agreement
- 1854 • Data Provider

1855 **Note** regardless of the Constrainable structure, the restricted values relate to the  
1856 allowable content for the Component of the DSD to which the constrained object  
1857 relates.

### 1858 **10.4.2 Metadata Constraint**

1859 A Metadata Constraint is used to add additional restrictions to the allowable values  
1860 reported in a metadataset. Metadata Constraints can be applied to the follow  
1861 structures which are collectively known as Constrainable structures:

- 1862 • Metadata Structure Definition
- 1863 • Metadataflow
- 1864 • Metadata Provision Agreement
- 1865 • Metadata Provider

1866 **Note** regardless of the Constrainable structure, the restricted values relate to the  
1867 allowable content for the Component of the MSD to which the constrained object  
1868 relates.

1869

1870 **10.4.3 Scope of a Constraint**

1871 A Constraint is used specify the content of a data or metadata source in terms of the  
1872 component values or the keys.

1873

1874 In terms of data the components are:

1875

- Dimension

1876

- Time Dimension

1877

- Data Attribute

1878

- Measure

1879

- Metadata Attribute

1880

- DataKeySets: the keys are the content of the KeyDescriptor – i.e., the series keys composed, for each key, by a value for each Dimension.

1881

1882

1883 In terms of reference metadata the components are:

1884

- Metadata Attribute

1885

1886 For a Constraint based on a DSD the Constraint can reference one or more of:

1887

- Data Structure Definition

1888

- Dataflow

1889

- Provision Agreement

1890

- Data Provider

1891

1892 For a Constraint based on an MSD the Constraint can reference one or more of:

1893

- Metadata Structure Definition

1894

- Metadataflow

1895

- Metadata Provision Agreement

1896

- Metadata Provider

1897

- Metadata Set

1898

1899 Furthermore, there can be more than one Constraint specified for a specific object e.g.,  
1900 more than one Constraint for a specific DSD.

1901

1902 In view of the flexibility of constraints attachment, clear rules on their usage are  
1903 required. These are elaborated below.

1904 **10.4.4 Multiple Constraints**

1905 There can be many Constraints for any Constraining Artefact (e.g., DSD), subject to  
1906 the following restrictions:

1907 **10.4.4.1 Cube Region**

1908 A Constraint can contain multiple Member Selections (e.g., Dimensions).

1909 • A specific Member Selection (e.g., Dimension `FREQ`) can only be contained in  
1910 one Cube Region for any one attached object (e.g., a specific DSD or specific  
1911 Dataflow).

1912 • Component values within a Member Selection may define a validity period.  
1913 Otherwise, the value is valid for the whole validity of the Cube Region.

1914 • For partial reference resolution purposes (as per the SDMX REST API), the latest  
1915 non-draft Constraint must be considered.

1916 • A Member Selection may include wildcarding of values (using character `'%'` to  
1917 represent zero or more occurrences of any character), as well as cascading  
1918 through hierarchic structures (e.g., parents in Codelist), or localised values (e.g.,  
1919 text for English only). Lack of locale means any language may match. Cascading  
1920 values are mutual exclusive to localised values, as the former refer to coded  
1921 values, while the latter refer to uncoded values.

1922 • Any values included in a Member Selection for Components with an array data  
1923 type (i.e., Measures, Attributes or Metadata Attributes), will be applied as single  
1924 values and will not be assessed combined with other values to match all possible  
1925 array values. For example, including the Code `'A'` for an Attribute will allow any  
1926 instance of the Attribute that includes `'A'`, like `['A', 'B']` or `['A', 'C', 'D']`. Similarly, if  
1927 Code `'A'` was excluded, all those arrays of values would also be excluded.

1928

1929 **10.4.4.2 Key Set**

1930 Key Sets will be processed in the order they appear in the Constraint and wildcards  
1931 can be used (e.g., any key position not reference explicitly is deemed to be "all  
1932 values").

1933

1934 As the Key Sets can be "included" or "excluded" it is recommended that Key Sets with  
1935 wildcards are declared before KeySets with specific series keys. This will minimize the  
1936 risk that keys are inadvertently included or excluded.

1937

1938 In addition, Attribute, Measure and Metadata Attribute constraints may accompany  
1939 KeySets, in order to specify the allowed values per Key. Those are expressed following  
1940 the rules for Cube Regions, as explained above.

1941

1942 Finally, a validity period may be specified per Key.

1943

1944 **10.4.5 Versioning**

1945 When Data and Metadata Constraints are versioned, the latest version of the  
1946 Constraint is used to generate the reporting restriction rules; all previous versions are  
1947 for historical information only.

1948

1949 If restrictions are applicable to certain periods in time, the `validFrom` and `validTo`  
1950 properties can be set on the specific values. This allows Constraints to evolve over  
1951 time, increasing their version number as they do so, whilst being able to maintain a  
1952 complete set of reporting restrictions for current and past datasets.

1953  
1954  
1955  
1956

Example:

Data Constraint 1.0.0

Component	Valid Value	Valid from	Valid to
COUNTRY	UK		
	FR		
	DE		

1957  
1958

Data Constraint 1.1.0

Component	Valid Value	Valid from	Valid to
COUNTRY	UK		
	FR		2012
	DE		

1959  
1960  
1961  
1962  
1963

When both versions of the Data Constraint are in a system, an observation value reported against COUNTRY FR for time period 2013 would be deemed invalid as the 1.1.0 rule would be applied.

1964

**10.4.6 Inheritance**

1965

**10.4.6.1 Attachment levels of a Constraint**

1966  
1967  
1968  
1969  
1970

There are three levels of constraint attachment for which these inheritance rules apply:

- DSD/MSD – top level
  - Dataflow/Metadataflow – second level
    - Provision Agreement – third level

1971  
1972  
1973

It is not necessary for a Constraint to be attached to a higher level artefact. e.g., it is valid to have a Constraint for a Provision Agreement where there are no constraints attached the relevant Dataflow or DSD.

1974

**10.4.6.2 Cascade rules for processing Constraints**

1975  
1976  
1977  
1978  
1979

The processing of the constraints on either Dataflow/Metadataflow or Provision Agreement must take into account the constraints declared at higher levels. The rules for the lower-level constraints (attached to Dataflow/ Metadataflow and Provision Agreement) are detailed below.

1980  
1981  
1982  
1983  
1984  
1985  
1986

Note that there can be a situation where a constraint is specified at a lower level before a constraint is specified at a higher level. Therefore, it is possible that a higher-level constraint makes a lower-level constraint invalid. SDMX makes no rules on how such a conflict should be handled when processing the constraint for attachment. However, the cascade rules on evaluating constraints for usage are clear – the higher-level constraint takes precedence in any conflicts that result in a less restrictive specification at the lower level.

1987

**10.4.6.3 Cube Region**

1988  
1989  
1990  
1991

It is not necessary to have a Constraint on the higher-level artefact (e.g., DSD referenced by the Dataflow), but if there is such a Constraint at the higher level(s) then:

- The lower-level Constraint cannot be less restrictive than the Constraint specified for the same Member Selection (e.g. Dimension) at the next higher level, which

1992           constrains that Member Selection. For example, if the Dimension FREQ is  
1993           constrained to A, Q in a DSD, then the Constraint at the Dataflow or Provision  
1994           Agreement cannot be A, Q, M or even just M – it can only further constrain A, Q.

1995           • The Constraint at the lower level for any one Member Selection further constrains  
1996           the content for the same Member Selection at the higher level(s).

1997           • Any Member Selection, which is not referenced in a Constraint, is deemed to be  
1998           constrained according to the Constraint specified at the next higher level which  
1999           constrains that Member Selection.

2000           • If there is a conflict when resolving the Constraint in terms of a lower-level  
2001           Constraint being less restrictive than a higher-level Constraint, then the  
2002           Constraint at the higher-level is used.

2003  
2004           Note that it is possible for a Constraint at a higher level to constrain, say, four  
2005           Dimensions in a single Constraint, and a Constraint at a lower level to constrain the  
2006           same four in two, three, or four Constraints.  
2007

#### 2008           **10.4.6.4 Key Set**

2009           It is not necessary to have a Constraint on the higher-level artefact (e.g., DSD  
2010           referenced by the Dataflow), but if there is such a Constraint at the higher level(s) then:

2011           • The lower-level Constraint cannot be less restrictive than the Constraint specified  
2012           at the higher level.

2013           • The Constraint at the lower level for any one Member Selection further constrains  
2014           the keys specified at the higher level(s).

2015           • Any Member Selection, which is not referenced in a Constraint, is deemed to be  
2016           constrained according to the Constraint specified at the next higher level which  
2017           constrains that Member Selection.

2018           • If there is a conflict when resolving the keys in the Constraint at two levels, in  
2019           terms of a lower-level constraint being less restrictive than a higher-level  
2020           Constraint, then the offending keys specified at the lower level are not deemed  
2021           part of the Constraint.

2022  
2023           Note that a Key in a Key Set can have wildcarded Components. For instance, the  
2024           Constraint may simply constrain the Dimension FREQ to "A", and all keys where the  
2025           FREQ="A" are therefore valid.  
2026

2027           The following logic explains how the inheritance mechanism works. Note that this is  
2028           conceptual logic and actual systems may differ in the way this is implemented.  
2029

- 2030           1. Determine all possible keys that are valid at the higher level.
- 2031           2. These keys are deemed to be inherited by the lower-level constrained object,  
2032           subject to the Constraints specified at the lower level.
- 2033           3. Determine all possible keys that are possible using the Constraints specified at  
2034           the lower level.
- 2035           4. At the lower level inherit all keys that match with the higher-level Constraint.
- 2036           5. If there are keys in the lower-level Constraint that are not inherited then the key  
2037           is invalid (i.e., it is less restrictive).

2038 **10.4.7 Constraints Examples**

2039 **10.4.7.1 Data Constraint and Cascading**

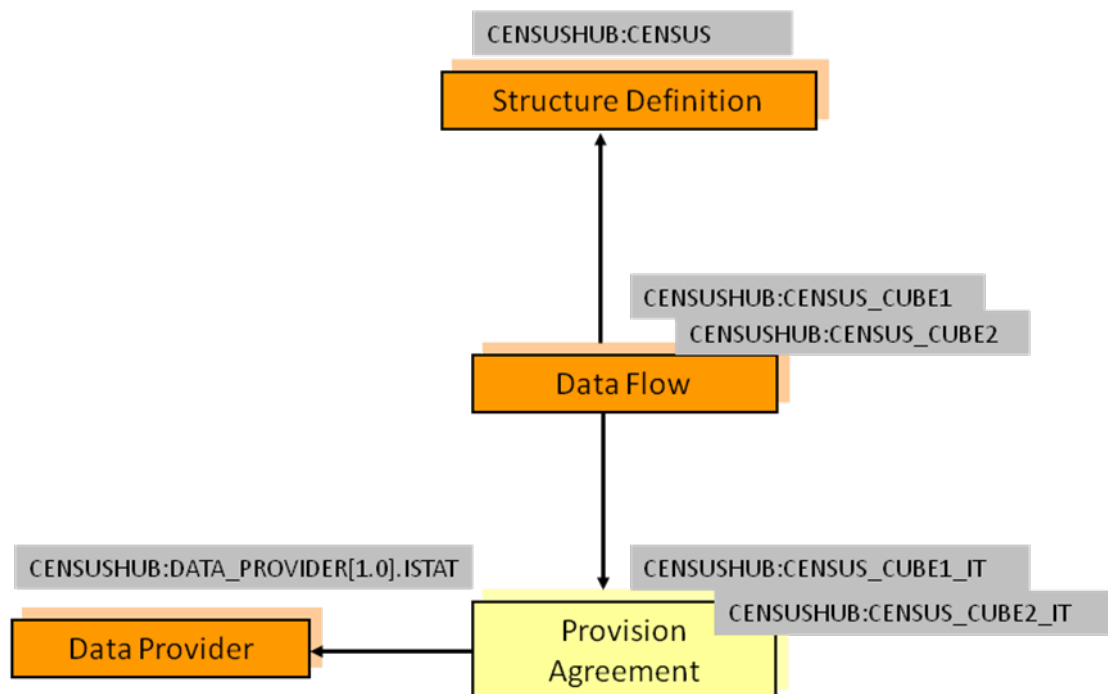
2040 The following scenario is used.

2041

2042 A DSD contains the following Dimensions:

- 2043 • GEO – Geography
- 2044 • SEX – Sex
- 2045 • AGE – Age
- 2046 • CAS – Current Activity Status

2047 In the DSD, common code lists are used and the requirement is to restrict these at  
 2048 various levels to specify the actual code that are valid for the object to which the  
 2049 Constraint is attached.



2050

2051

**Figure 20: Example Scenario for Constraints**

2052 Constraints are declared as follows:

Exclude AGE code 001 `CENSUSHUB:CENSUS`  
 Structure Definition

Restrict CAS to codes 003 and 004 `CENSUSHUB:CENSUS_CUBE1`  
 Restrict AGE to 002 and 003

Restrict CAS to codes TOT and NAP `CENSUSHUB:CENSUS_CUBE2`  
 Data Flow

Restrict GEO to codes IT and children `CENSUSHUB:CENSUS_CUBE1_IT`  
 Restrict GEO to codes IT and children `CENSUSHUB:CENSUS_CUBE2_IT`  
 Provision Agreement

**Figure 21: Example Constraints**

2053  
2054

2055 Notes:

2056 AGE is constrained for the DSD and is further restricted for the Dataflow  
 2057 CENSUS\_CUBE1.

- 2058 • The same Constraint applies to both Provision Agreements.

2059

2060 The cascade rules elaborated above result as follows:

2061

2062 DSD

- 2063 • Constrained by eliminating code 001 from the code list for the AGE Dimension.

2064

2065 Dataflow CENSUS\_CUBE1

- 2066 • Constrained by restricting the code list for the AGE Dimension to codes 002  
 2067 and 003 (note that this is a more restrictive constraint than that declared for the  
 2068 DSD which specifies all codes except code 001).
  - 2069 ○ Restricts the CAS codes to 003 and 004.

2070

2071 Dataflow CENSUS\_CUBE2

- 2072 • Restricts the code list for the CAS Dimension to codes TOT and NAP.
  - 2073 ○ Inherits the AGE constraint applied at the level of the DSD.

2074

2075 Provision Agreement CENSUS\_CUBE1\_IT

- 2076 • Restricts the codes for the GEO Dimension to IT and its children.
  - 2077 ○ Inherits the constraints from Dataflow CENSUS\_CUBE1 for the AGE  
 2078 and CAS Dimensions.

2079

2080 Provision Agreement CENSUS\_CUBE2\_IT

- 2081 • Restricts the codes for the GEO Dimension to IT and its children.

- 2082           ○ Inherits the constraints from Dataflow CENSUS\_CUBE2 for the CAS
- 2083           Dimension.
- 2084           ○ Inherits the AGE constraint applied at the level of the DSD.
- 2085

2086 The Constraints are defined as follows:

2087 DSD Constraint

```

2088 <str:DataConstraint agencyID="SDMX" id="DATA_CONSTRAINT" version="1.0.0-
2089 draft" type="Allowed">
2090   <com:Name xml:lang="en">SDMX 3.0 Data Constraint sample</com:Name>
2091   <str:ConstraintAttachment>
2092     <str:DataStructure>urn:sdmx:org.sdmx.infomodel.datastructure.
2093       DataStructure=CENSUSHUB:CENSUS(1.0.0)</str:DataStructure>
2094   </str:ConstraintAttachment>
2095   <str:CubeRegion include="true">
2096     <!-- the ability to exclude values is illustrated - i.e., all values
2097     valid except this one -->
2098     <com:KeyValue id="AGE" include="false">
2099       <com:Value>001</com:Value>
2100     </com:KeyValue>
2101   </str:CubeRegion>
2102 </str:DataConstraint>

```

2103

2104 Dataflow Constraints

```

2105 <str:DataConstraint agencyID="SDMX" id="DATA_CONSTRAINT_2" version="1.0.0-
2106 draft" type="Allowed">
2107   <com:Name xml:lang="en">SDMX 3.0 Data Constraint sample</com:Name>
2108   <str:ConstraintAttachment>
2109     <str>Dataflow>urn:sdmx:org.sdmx.infomodel.datastructure.Dataflow=
2110       CENSUSHUB:CENSUS_CUBE1(1.0.0)</str>Dataflow>
2111   </str:ConstraintAttachment>
2112   <str:CubeRegion include="true">
2113     <com:KeyValue id="AGE" include="true">
2114       <com:Value>002</com:Value>
2115       <com:Value>003</com:Value>
2116     </com:KeyValue>
2117     <com:KeyValue id="CAS">
2118       <com:Value>003</com:Value>
2119       <com:Value>004</com:Value>
2120     </com:KeyValue>
2121   </str:CubeRegion>
2122 </str:DataConstraint>

```

```

2123
2124 <str:DataConstraint agencyID="SDMX" id="DATA_CONSTRAINT_3" version="1.0.0-
2125 draft" type="Allowed">
2126   <com:Name xml:lang="en">SDMX 3.0 Data Constraint sample</com:Name>
2127   <str:ConstraintAttachment>
2128     <str>Dataflow>urn:sdmx:org.sdmx.infomodel.datastructure.Dataflow=
2129       CENSUSHUB:CENSUS_CUBE2(1.0.0)</str>Dataflow>
2130   </str:ConstraintAttachment>
2131   <str:CubeRegion include="true">
2132     <com:KeyValue id="CAS" include="true">
2133       <com:Value>TOT</com:Value>
2134       <com:Value>NAP</com:Value>
2135     </com:KeyValue>
2136   </str:CubeRegion>
2137 </str:DataConstraint>

```

2138

2139 Provision Agreement Constraint

```

2140 <str:DataConstraint agencyID="SDMX" id="DATA_CONSTRAINT_4" version="1.0.0-
2141 draft" type="Allowed">
2142   <com:Name xml:lang="en">SDMX 3.0 Data Constraint sample</com:Name>

```

```

2143 <str:ConstraintAttachment>
2144 <str:ProvisionAgreement>urn:sdmx:org.sdmx.infomodel.registry.
2145   ProvisionAgreement=CENSUSHUB:CENSUS_CUBE1_IT(1.0.0)
2146 </str:ProvisionAgreement>
2147 <str:ProvisionAgreement>urn:sdmx:org.sdmx.infomodel.registry.
2148   ProvisionAgreement=CENSUSHUB:CENSUS_CUBE2_IT(1.0.0)
2149 </str:ProvisionAgreement>
2150 </str:ConstraintAttachment>
2151 <str:CubeRegion include="true">
2152   <com:KeyValue id="GEO" include="true">
2153     <com:Value cascadeValues="true">IT</com:Value>
2154   </com:KeyValue>
2155 </str:CubeRegion>
2156 </str:DataConstraint

```

2157

2158 **10.4.7.2 Combination of Constraints**

2159 The possible combination of constraining terms are explained in this section, following  
 2160 a few examples.

2161

2162 Let's assume a DSD with the following Components:

Dimension	FREQ
Dimension	JD_TYPE
Dimension	JD_CATEGORY
Dimension	VIS_CTY
TimeDimension	TIME_PERIOD
Attribute	OBS_STATUS
Attribute	UNIT
Attribute	COMMENT
MetadataAttribute	CONTACT
Measure	MULTISELECT
Measure	CHOICE

2163

2164 On the above, let's assume the following use cases with their constraining  
 2165 requirements:

2166

2167 **Use Case 1: A Constraint on allowed values for some Dimensions**

2168 R1: Allow *monthly* and *quarterly* data

2169 R2: Allow *Mexico* for vis-à-vis country

2170

2171 This is expressed with the following *CubeRegion*:

FREQ	M, Q
VIS_CTY	MX

2172

2173 **Use Case 2: A Constraint on allowed combinations for some Dimensions**

2174 R1: Allow *monthly* data for *Germany*

2175 R2: Allow *quarterly* data for *Mexico*

2176

2177 This is expressed with the following *DataKeySet*:

Key1	FREQ	M
------	------	---

	VIS_CTY	DE
Key2	FREQ	Q
	VIS_CTY	MX

2178  
2179  
2180  
2181  
2182  
2183  
2184  
2185

**Use Case 3: A Constraint on allowed values for some Dimensions combined with allowed values for some Attributes**

- R1: Allow `monthly` and `quarterly` data
- R2: Allow `Mexico` for vis-à-vis country
- R3: Allow `present` for status

This may be expressed with the following `CubeRegion`:

FREQ	M, Q
VIS_CTY	MX
OBS_STATUS	A

2186  
2187  
2188  
2189  
2190  
2191  
2192

**Use Case 4: A Constraint on allowed combinations for some Dimensions combined with specific Attribute values**

- R1: Allow `monthly` data, for `Germany`, with unit `euro`
- R2: Allow `quarterly` data, for `Mexico`, with unit `usd`

This may be expressed with the following `DataKeySet`:

Key1	FREQ	M
	VIS_CTY	DE
	UNIT	EUR
Key2	FREQ	Q
	VIS_CTY	MX
	UNIT	USD

2193  
2194  
2195  
2196  
2197  
2198  
2199  
2200

**Use Case 5: A Constraint on allowed values for some Dimensions together with some combination of Dimension values**

- R1: For `annually` and `quarterly` data, for `Mexico` and `Germany`, only `A` status is allowed
- R2: For `monthly` data, for `Mexico` and `Germany`, only `F` status is allowed

Considering the above examples, the following `CubeRegions` would be created:

CubeRegion1	FREQ	Q, A
	VIS_CTY	MX, DE
	OBS_STATUS	A
CubeRegion2	FREQ	M
	VIS_CTY	MX, DE
	OBS_STATUS	F

2201  
2202  
2203  
2204  
2205  
2206

The problem with this approach is that according to the business rule for `Constraints`, only one should be specified per `Component`. Thus, if a software would perform some conflict resolution would end up with empty sets for `FREQ` and `OBS_STATUS` (as they do not share any values).

2207 Nevertheless, there is a much easier approach to that; this is the cascading  
 2208 mechanism of `Constraints` (as shown in 10.4.7.1). Hence, these rules would be  
 2209 expressed into two levels of `Constraints`, e.g., `DSD` and `Dataflows`:

2210  
 2211 `DSD CubeRegion`:

FREQ	M, Q, A
VIS_CTY	MX, DE
OBS_STATUS	A, F

2212  
 2213 `Dataflow1 CubeRegion`:

FREQ	Q, A
VIS_CTY	MX, DE
OBS_STATUS	F

2214  
 2215 `Dataflow2 CubeRegion`:

FREQ	M
VIS_CTY	MX, DE
OBS_STATUS	A

2216  
 2217 **Use case 6: A Constraint on allowed values for some Dimensions combined with**  
 2218 **allowed values for Measures**

2219 R1: Allow `monthly` data, for `Germany`, with unit `euro`, and measure choice is 'A'

2220 R2: Allow `quarterly` data, for `Mexico`, with unit `usd`, and measure choice is 'B'

2221  
 2222 This may be expressed with the following `DataKeySet`:

Key1	FREQ	M
	VIS_CTY	DE
	UNIT	EUR
	CHOICE	A
Key2	FREQ	Q
	VIS_CTY	MX
	UNIT	USD
	CHOICE	B

2223  
 2224 **Use Case 7: A Constraint with wildcards for Codes and removePrefix property**  
 2225 For this example, we assume that the `VIS_CTY` representation has been prefixed with  
 2226 prefix 'AREA\_'. In this Constraint, we need to remove the prefix.

2227 R1: Allow `monthly` and `quarterly` data

2228 R2: Allow vis-à-vis countries that start with M

2229 R3: Remove the prefix 'AREA\_'

2230  
 2231 This may be expressed with the following `CubeRegion`:

FREQ	M, Q
VIS_CTY (removePrefix='AREA_')	M%

2232  
 2233 **Use Case 8: A Constraint with multilingual support on Attributes**

2234 R1: Allow `monthly` and `quarterly` data

2235 R2: Allow `Mexico` for vis-à-vis country

2236 R3: Allow a comment, in English, which includes the term `adjusted` for status

2237

2238 This may be expressed with the following `CubeRegion`:

<code>FREQ</code>	<code>M, Q</code>
<code>VIS_CTY</code>	<code>MX</code>
<code>COMMENT (lang='en')</code>	<code>%adjusted%</code>

2239

2240 **Use Case 9: A Constraint on allowed values for Dimensions combined with**  
 2241 **allowed values for Metadata Attributes**

2242 R1: Allow `monthly` and `quarterly` data

2243 R2: Allow `Mexico` for vis-à-vis country

2244 R3: Allow `John Doe` for contact

2245

2246 This may be expressed with the following `CubeRegion`:

<code>FREQ</code>	<code>M, Q</code>
<code>VIS_CTY</code>	<code>MX</code>
<code>CONTACT</code>	<code>John Doe</code>

2247

2248 **11 Transforming between versions of SDMX**

2249 **11.1 Scope**

2250 The scope of this section is to define both best practices and mandatory behaviour for  
2251 specific aspects of transformation between different versions of SDMX.

2252 **11.2 Compatibility and new DSD features**

2253 The following table provides an overview of the backwards compatibility between  
2254 SDMX 3.0 and 2.1.  
2255

SDMX 3.0 feature	SDMX 2.1 compatibility	Comments
Multiple Measures	Create a Measure Dimension Or Model Measures as Attributes	For a Measure Dimensions, all Concepts must reside in the same Concept Scheme
Arrays for values	Cannot be supported	Arrays are always reported in a verbose format, even if one value is reported
Measure Relationship	Can be ignored, as it does not affect dataset format	
Metadata Attributes	Can be created as Data Attributes	Not for extended facets
Multilingual Components	Cannot be supported	
No Measure	Can only be emulated by ignoring the Primary Measure value	
Use extended Codelist	A new Codelist with all Codes must be created	
Sentinel values	Cannot be supported in the DSD	Rules may be supported outside the DSD, in bilateral agreements

2256 The following table illustrates forward compatibility issues from SDMX 2.1 to 3.0.  
2257  
2258

SDMX 2.1 feature	SDMX 3.0 compatibility	Comments
Measure Dimension	Create a Dimension with role 'MEASURE' Or Create multiple Measures from the Measure Dimension Concept Scheme	If the dataset has to resemble that of SDMX 2.1 Structure Specific, then the first option must be used
Primary Measure	Create one Measure with role 'PRIMARY'; use id="OBS_VALUE"	

2259

## 2260 12 Validation and Transformation Language (VTL)

### 2261 12.1 Introduction

2262 The Validation and Transformation Language (VTL) supports the definition of  
2263 Transformations, which are algorithms to calculate new data starting from already  
2264 existing ones<sup>7</sup>. The purpose of the VTL in the SDMX context is to enable the:

- 2265
- 2266 • definition of validation and transformation algorithms, in order to specify how to  
2267 calculate new data from existing ones;
- 2268 • exchange of the definition of VTL algorithms, also together the definition of the  
2269 data structures of the involved data (for example, exchange the data structures  
2270 of a reporting framework together with the validation rules to be applied,  
2271 exchange the input and output data structures of a calculation task together  
2272 with the VTL Transformations describing the calculation algorithms);
- 2273 • compilation and execution of VTL algorithms, either interpreting the VTL  
2274 Transformations or translating them in whatever other computer language is  
2275 deemed as appropriate.
- 2276

2277 It is important to note that the VTL has its own information model (IM), derived from  
2278 the Generic Statistical Information Model (GSIM) and described in the VTL User Guide.  
2279 The VTL IM is designed to be compatible with more standards, like SDMX, DDI (Data  
2280 Documentation Initiative) and GSIM, and includes the model artefacts that can be  
2281 manipulated (inputs and/or outputs of Transformations, e.g. "Data Set", "Data  
2282 Structure") and the model artefacts that allow the definition of the transformation  
2283 algorithms (e.g. "Transformation", "Transformation Scheme").

2284

2285 The VTL language can be applied to SDMX artefacts by mapping the SDMX IM model  
2286 artefacts to the model artefacts that VTL can manipulate<sup>8</sup>. Thus, the SDMX artefacts  
2287 can be used in VTL as inputs and/or outputs of Transformations. It is important to be  
2288 aware that the artefacts do not always have the same names in the SDMX and VTL  
2289 IMs, nor do they always have the same meaning. The more evident example is given  
2290 by the SDMX `Dataset` and the VTL "Data Set", which do not correspond one another:  
2291 as a matter of fact, the VTL "Data Set" maps to the SDMX "Dataflow", while the  
2292 SDMX "Dataset" has no explicit mapping to VTL (such an abstraction is not needed  
2293 in the definition of VTL Transformations). A SDMX "Dataset", however, is an instance  
2294 of a SDMX "Dataflow" and can be the artefact on which the VTL transformations are  
2295 executed (i.e., the Transformations are defined on `Dataflows` and are applied to  
2296 `Dataflow` instances that can be `Datasets`).

2297

2298 The VTL programs (Transformation Schemes) are represented in SDMX through the  
2299 `TransformationScheme` maintainable class which is composed of  
2300 `Transformation` (nameable artefact). Each `Transformation` assigns the  
2301 outcome of the evaluation of a VTL expression to a result.

2302

---

<sup>7</sup> The Validation and Transformation Language is a standard language designed and published under the SDMX initiative. VTL is described in the VTL User and Reference Guides available on the SDMX website <https://sdmx.org>.

<sup>8</sup> In this chapter, in order to distinguish VTL and SDMX model artefacts, the VTL ones are written in the Arial font while the SDMX ones in Courier New

2303 This section does not explain the VTL language or any of the content published in the  
2304 VTL guides. Rather, this is a description of how the VTL can be used in the SDMX  
2305 context and applied to SDMX artefacts.

## 2306 **12.2 References to SDMX artefacts from VTL statements**

### 2307 **12.2.1 Introduction**

2308 The VTL can manipulate SDMX artefacts (or objects) by referencing them through pre-  
2309 defined conventional names (aliases).

2310  
2311 The alias of an SDMX artefact can be its URN (Universal Resource Name), an  
2312 abbreviation of its URN or another user-defined name.

2313  
2314 In any case, the aliases used in the VTL Transformations have to be mapped to the  
2315 SDMX artefacts through the `VtlMappingScheme` and `VtlMapping` classes (see the  
2316 section of the SDMX IM relevant to the VTL). A `VtlMapping` allows specifying the  
2317 aliases to be used in the VTL Transformations, Rulesets<sup>9</sup> or User Defined Operators<sup>10</sup>  
2318 to reference SDMX artefacts. A `VtlMappingScheme` is a container for zero or more  
2319 `VtlMapping`.

2320  
2321 The correspondence between an alias and a SDMX artefact must be one-to-one,  
2322 meaning that a generic alias identifies one and just one SDMX artefact while a SDMX  
2323 artefact is identified by one and just one alias. In other words, within a  
2324 `VtlMappingScheme` an artefact can have just one alias and different artefacts cannot  
2325 have the same alias.

2326  
2327 The references through the URN and the abbreviated URN are described in the  
2328 following paragraphs.

### 2329 **12.2.2 References through the URN**

2330 This approach has the advantage that in the VTL code the URN of the referenced  
2331 artefacts is directly intelligible by a human reader but has the drawback that the  
2332 references are verbose.

2333  
2334 The SDMX URN<sup>11</sup> is the concatenation of the following parts, separated by special  
2335 symbols like dot, equal, asterisk, comma, and parenthesis:

- 2336       • `SDMXprefix`  
2337       • `SDMX-IM-package-name`  
2338       • `class-name`  
2339       • `agency-id`

---

<sup>9</sup> See also the section "VTL-DL Rulesets" in the VTL Reference Manual.

<sup>10</sup> The `VtlMappings` are used also for User Defined Operators (UDO). Although UDOS are envisaged to be defined on generic operands, so that the specific artefacts to be manipulated are passed as parameters at their invocation, it is also possible that an UDO invokes directly some specific SDMX artefacts. These SDMX artefacts have to be mapped to the corresponding aliases used in the definition of the UDO through the `VtlMappingScheme` and `VtlMapping` classes as well.

<sup>11</sup> For a complete description of the structure of the URN see the SDMX 2.1 Standards - Section 5 - Registry Specifications, paragraph 6.2.2 ("Universal Resource Name (URN)").

- 2340 • maintainedobject-id
- 2341 • maintainedobject-version
- 2342 • container-object-id <sup>12</sup>
- 2343 • object-id

2344 The generic structure of the URN is the following:

2345  
2346 `SDMXprefix.SDMX-IM-package-name.class-name=agency-id:maintainedobject-id`  
2347 `(maintainedobject-version).*container-object-id.object-id`

2348  
2349 The **SDMXprefix** is "urn:sdmx:org", always the same for all SDMX artefacts.

2350  
2351 The **SDMX-IM-package-name** is the concatenation of the string "sdmx.infomodel." with  
2352 the package-name, which the artefact belongs to. For example, for referencing a  
2353 `Dataflow` the **SDMX-IM-package-name** is "sdmx.infomodel.datastructure", because the  
2354 class `Dataflow` belongs to the package "datastructure".

2355  
2356 The **class-name** is the name of the SDMX object class, which the SDMX object belongs  
2357 to (e.g., for referencing a `Dataflow` the **class-name** is "Dataflow"). The VTL can  
2358 reference SDMX artefacts that belong to the classes `Dataflow`, `Dimension`,  
2359 `TimeDimension`, `Measure`, `DataAttribute`, `Concept`, `Codelist`.

2360  
2361 The **agency-id** is the acronym of the agency that owns the definition of the artefact, for  
2362 example for the Eurostat artefacts the **agency-id** is "ESTAT"). The **agency-id** can be  
2363 composite (for example `AgencyA.Dept1.Unit2`).

2364  
2365 The **maintainedobject-id** is the name of the maintained object which the artefact  
2366 belongs to, and in case the artefact itself is maintainable<sup>13</sup>, coincides with the name of  
2367 the artefact. Therefore the **maintainedobject-id** depends on the class of the artefact:

- 2368
- 2369 • if the artefact is a `Dataflow`, which is a maintainable class, the
- 2370 **maintainedobject-id** is the `Dataflow` name (`dataflow-id`);
- 2371 • if the artefact is a `Dimension`, `Measure`, `TimeDimension` or
- 2372 `DataAttribute`, which are not maintainable and belong to the
- 2373 `DataStructure` maintainable class, the **maintainedobject-id** is the name of
- 2374 the `DataStructure` (`datastructure-id`) which the artefact belongs to;
- 2375 • if the artefact is a `Concept`, which is not maintainable and belongs to the
- 2376 `ConceptScheme` maintainable class, the **maintainedobject-id** is the name
- 2377 of the `ConceptScheme` (`conceptscheme-id`) which the artefact belongs to;
- 2378 • if the artefact is a `Codelist`, which is a maintainable class, the
- 2379 **maintainedobject-id** is the `Codelist` name (`codelist-id`).

2380  
2381 The **maintainedobject-version** is the version, according to the SDMX versioning  
2382 rules, of the maintained object which the artefact belongs to (for example, possible  
2383 versions might be 1.0, 2.3, 1.0.0, 2.1.0 or 3.1.2).

2384  
2385 The **container-object-id** does not apply to the classes that can be referenced in VTL  
2386 Transformations, therefore is not present in their URN

2387

---

<sup>12</sup> The container-object-id can repeat and may not be present.

<sup>13</sup> i.e., the artefact belongs to a maintainable class

2388 The object-id is the name of the non-maintainable artefact (when the artefact is  
2389 maintainable its name is already specified as the `maintainedobject-id`, see above), in  
2390 particular it has to be specified:

- 2391
- 2392 • if the artefact is a `Dimension`, `TimeDimension`, `Measure` or  
2393 `DataAttribute` (the object-id is the name of one of the artefacts above,  
2394 which are data structure components)
  - 2395 • if the artefact is a `Concept` (the object-id is the name of the `Concept`)

2396  
2397 For example, by using the URN, the VTL Transformation that sums two SDMX  
2398 Dataflows DF1 and DF2 and assigns the result to a third persistent Dataflow DFR,  
2399 assuming that DF1, DF2 and DFR are the `maintainedobject-id` of the three  
2400 Dataflows, that their version is 1.0.0 and their Agency is AG, would be written as<sup>14</sup>:

2401  
2402 `'urn:sdmx:org.sdmx.infomodel.datastructure.Dataflow=AG:DFR(1.0.0)'` <-  
2403 `'urn:sdmx:org.sdmx.infomodel.datastructure.Dataflow=AG:DF1(1.0.0)'` +  
2404 `'urn:sdmx:org.sdmx.infomodel.datastructure.Dataflow=AG:DF2(1.0.0)'`

### 2405 12.2.3 Abbreviation of the URN

2406 The complete formulation of the URN described above is exhaustive but verbose, even  
2407 for very simple statements. In order to reduce the verbosity through a simplified  
2408 identifier and make the work of transformation definers easier, proper abbreviations of  
2409 the URN are possible. Using this approach, the referenced artefacts remain intelligible  
2410 in the VTL code by a human reader.

2411  
2412 The URN can be abbreviated by omitting the parts that are not essential for the  
2413 identification of the artefact or that can be deduced from other available information,  
2414 including the context in which the invocation is made. The possible abbreviations are  
2415 described below.

- 2416
- 2417 • The `SDMXprefix` can be omitted for all the SDMX objects, because it is a  
2418 prefixed string (`urn:sdmx:org`), always the same for SDMX objects.
  - 2419 • The `SDMX-IM-package-name` can be omitted as well because it can be deduced  
2420 from the `class-name` that follows it (the table of the SDMX-IM packages and  
2421 classes that allows this deduction is in the SDMX 2.1 Standards - Section 5 -  
2422 Registry Specifications, paragraph 6.2.3). In particular, considering the object  
2423 classes of the artefacts that VTL can reference, the package is:
    - 2424 ○ `"datastructure"` for the classes `Dataflow`, `Dimension`,
    - 2425 `TimeDimension`, `Measure`, `DataAttribute`,
    - 2426 ○ `"conceptscheme"` for the class `Concept`,
    - 2427 ○ `"codelist"` for the class `Codelist`.
  - 2428 • The `class-name` can be omitted as it can be deduced from the VTL invocation.  
2429 In particular, starting from the VTL class of the invoked artefact (e.g. `dataset`,  
2430 `component`, `identifier`, `measure`, `attribute`, `variable`, `valuedomain`), which is  
2431 known given the syntax of the invoking VTL operator<sup>15</sup>, the SDMX class can be

<sup>14</sup> Since these references to SDMX objects include non-permitted characters as per the VTL ID notation, they need to be included between single quotes, according to the VTL rules for irregular names.

<sup>15</sup> For the syntax of the VTL operators see the VTL Reference Manual

- 2432 deduced from the mapping rules between VTL and SDMX (see the section  
 2433 "Mapping between VTL and SDMX" hereinafter)<sup>16</sup>.
- 2434 • If the `agency-id` is not specified, it is assumed by default equal to the `agency-`  
 2435 `id` of the `TransformationScheme`, `UserDefinedOperatorScheme` or  
 2436 `RulesetScheme` from which the artefact is invoked. For example, the `agency-`  
 2437 `id` can be omitted if it is the same as the invoking `TransformationScheme`  
 2438 and cannot be omitted if the artefact comes from another agency<sup>17</sup>. Take also  
 2439 into account that, according to the VTL consistency rules, the agency of the  
 2440 result of a `Transformation` must be the same as its  
 2441 `TransformationScheme`, therefore the `agency-id` can be omitted for all the  
 2442 results (left part of `Transformation` statements).
- 2443 • As for the `maintainedobject-id`, this is essential in some cases while in other  
 2444 cases it can be omitted:
- 2445 ○ if the referenced artefact is a `Dataflow`, which is a maintainable class,  
 2446 the `maintainedobject-id` is the `dataflow-id` and obviously cannot be  
 2447 omitted;
  - 2448 ○ if the referenced artefact is a `Dimension`, `TimeDimension`, `Measure`,  
 2449 `DataAttribute`, which are not maintainable and belong to the  
 2450 `DataStructure` maintainable class, the `maintainedobject-id` is the  
 2451 `dataStructure-id` and can be omitted, given that these components are  
 2452 always invoked within the invocation of a `Dataflow`, whose  
 2453 `dataStructure-id` can be deduced from the SDMX structural definitions;
  - 2454 ○ if the referenced artefact is a `Concept`, which is not maintainable and  
 2455 belong to the `ConceptScheme` maintainable class, the `maintained`  
 2456 `object` is the `conceptScheme-id` and cannot be omitted;
  - 2457 ○ if the referenced artefact is a `Codelist`, which is a maintainable  
 2458 class, the `maintainedobject-id` is the `codelist-id` and obviously  
 2459 cannot be omitted.
- 2460 • When the `maintainedobject-id` is omitted, the `maintainedobject-version` is  
 2461 omitted too. When the `maintainedobject-id` is not omitted and the  
 2462 `maintainedobject-version` is omitted, the version 1.0 is assumed by default.
- 2463 • As said, the `container-object-id` does not apply to the classes that can be  
 2464 referenced in VTL Transformations, therefore is not present in their URN
  - 2465 • The `object-id` does not exist for the artefacts belonging to the `Dataflow`,  
 2466 and `Codelist` classes, while it exists and cannot be omitted for the  
 2467 artefacts belonging to the classes `Dimension`, `TimeDimension`,  
 2468 `Measure`, `DataAttribute` and `Concept`, as for them the `object-id` is  
 2469 the main identifier of the artefact

---

<sup>16</sup> In case the invoked artefact is a VTL component, which can be invoked only within the invocation of a VTL data set (SDMX `Dataflow`), the specific SDMX class-name (e.g. `Dimension`, `TimeDimension`, `Measure` or `DataAttribute`) can be deduced from the data structure of the SDMX `Dataflow`, which the component belongs to.

<sup>17</sup> If the Agency is composite (for example `AgencyA.Dept1.Unit2`), the agency is considered different even if only part of the composite name is different (for example `AgencyA.Dept1.Unit3` is a different Agency than the previous one). Moreover the `agency-id` cannot be omitted in part (i.e., if a `TransformationScheme` owned by `AgencyA.Dept1.Unit2` references an artefact coming from `AgencyA.Dept1.Unit3`, the specification of the `agency-id` becomes mandatory and must be complete, without omitting the possibly equal parts like `AgencyA.Dept1`)

2470 The simplified object identifier is obtained by omitting all the first part of the URN,  
2471 including the special characters, till the first part not omitted.

2472

2473 For example, the full formulation that uses the complete URN shown at the end of the  
2474 previous paragraph:

2475

2476 'urn:sdmx:org.sdmx.infomodel.datastructure.Dataflow=AG:DFR(1.0.0)' :=

2477 'urn:sdmx:org.sdmx.infomodel.datastructure.Dataflow=AG:DF1(1.0.0)' +

2478 'urn:sdmx:org.sdmx.infomodel.datastructure.Dataflow=AG:DF2(1.0.0)'

2479

2480 by omitting all the non-essential parts would become simply:

2481

2482 DFR := DF1 + DF2

2483

2484 The references to the `Codelists` can be simplified similarly. For example, given the  
2485 non-abbreviated reference to the `Codelist` `AG:CL_FREQ(1.0.0)`, which is<sup>18</sup>:

2486

2487 'urn:sdmx:org.sdmx.infomodel.codelist.Codelist=AG:CL\_FREQ(1.0.0)'

2488

2489 if the `Codelist` is referenced from a `RulesetScheme` belonging to the agency `AG`,  
2490 omitting all the optional parts, the abbreviated reference would become simply<sup>19</sup>:

2491

2492 CL\_FREQ

2493

2494 As for the references to the components, it can be enough to specify the component-  
2495 id, given that the `dataStructure-id` can be omitted. An example of non-abbreviated  
2496 reference, if the data structure is `DST1` and the component is `SECTOR`, is the  
2497 following:

2498

2499 'urn:sdmx:org.sdmx.infomodel.datastructure.DataStructure=AG:DST1(1.0.0).S  
2500 ECTOR'

2501

2502 The corresponding fully abbreviated reference, if made from a  
2503 `TransformationScheme` belonging to `AG`, would become simply:

2504

2505 SECTOR

2506

2507 For example, the `Transformation` for renaming the component `SECTOR` of the  
2508 `Dataflow` `DF1` into `SEC` can be written as<sup>20</sup>:

2509

2510 'DFR(1.0.0)' := 'DF1(1.0.0)' [rename SECTOR to SEC]

2511

2512 In the references to the `Concepts`, which can exist for example in the definition of the  
2513 `VTL Rulesets`, at least the `conceptScheme-id` and the `concept-id` must be  
2514 specified.

2515

<sup>18</sup> Single quotes are needed because this reference is not a VTL regular name.

<sup>19</sup> Single quotes are not needed in this case because `CL_FREQ` is a VTL regular name.

<sup>20</sup> The result `DFR(1.0.0)` is be equal to `DF1(1.0.0)` save that the component `SECTOR` is called `SEC`

2516 An example of non-abbreviated reference, if the `conceptScheme-id` is CS1 and the  
2517 `concept-id` is SECTOR, is the following:

2518  
2519 `'urn:sdmx:org.sdmx.infomodel.conceptscheme.Concept=AG:CS1(1.0.0).SECTOR'`

2520  
2521 The corresponding fully abbreviated reference, if made from a `RulesetScheme`  
2522 belonging to AG, would become simply:

2523  
2524 `CS1(1.0.0).SECTOR`

2525  
2526 The Codes and in general all the Values can be written without any other specification,  
2527 for example, the transformation to check if the values of the measures of the  
2528 `Dataflow` DF1 are between 0 and 25000 can be written like follows:

2529  
2530 `'DFR(1.0.0)' := between ( 'DF1(1.0.0)', 0, 25000 )`

2531  
2532 The artefact (`Component`, `Concept`, `Codelist` ...) which the Values are referred to  
2533 can be deduced from the context in which the reference is made, taking also into  
2534 account the VTL syntax. In the Transformation above, for example, the values 0 and  
2535 2500 are compared to the values of the measures of DF1(1.0.0).

#### 2536 **12.2.4 User-defined alias**

2537 The third possibility for referencing SDMX artefacts from VTL statements is to use  
2538 user-defined aliases not related to the SDMX URN of the artefact.

2539  
2540 This approach gives preference to the use of symbolic names for the SDMX artefacts.  
2541 As a consequence, in the VTL code the referenced artefacts may become not directly  
2542 intelligible by a human reader. In any case, the VTL aliases are associated to the  
2543 SDMX URN through the `VtlMappingScheme` and `VtlMapping` classes. These  
2544 classes provide for structured references to SDMX artefacts whatever kind of reference  
2545 is used in VTL statements (URN, abbreviated URN or user-defined aliases).

#### 2546 **12.2.5 References to SDMX artefacts from VTL Rulesets**

2547 The VTL Rulesets allow defining sets of reusable Rules that can be applied by some  
2548 VTL operators, like the ones for validation and hierarchical roll-up. A "Rule" consists in  
2549 a relationship between Values belonging to some Value Domains or taken by some  
2550 Variables, for example: (i) when the Country is USA then the Currency is USD; (ii) the  
2551 Benelux is composed by Belgium, Luxembourg, Netherlands.

2552  
2553 The VTL Rulesets have a signature, in which the Value Domains or the Variables on  
2554 which the Ruleset is defined are declared, and a body, which contains the Rules.

2555  
2556 In the signature, given the mapping between VTL and SDMX better described in the  
2557 following paragraphs, a reference to a VTL Value Domain becomes a reference to a  
2558 SDMX `Codelist`, while a reference to a VTL Represented Variable becomes a  
2559 reference to a SDMX `Concept`, assuming for it a definite representation<sup>21</sup>.

2560

---

<sup>21</sup> Rulesets of this kind cannot be reused when the referenced Concept has a different representation.

2561 In general, for referencing SDMX `Codelists` and `Concepts`, the conventions  
2562 described in the previous paragraphs apply. In the Ruleset syntax, the elements that  
2563 reference SDMX artefacts are called "valueDomain" and "variable" for the Datapoint  
2564 Rulesets and "ruleValueDomain", "ruleVariable", "condValueDomain" "condVariable"  
2565 for the Hierarchical Rulesets). The syntax of the Ruleset signature allows also to define  
2566 aliases of the elements above, these aliases are valid only within the specific Ruleset  
2567 definition statement and cannot be mapped to SDMX.<sup>22</sup>

2568  
2569 In the body of the Rulesets, the Codes and in general all the Values can be written  
2570 without any other specification, because the artefact, which the Values are referred  
2571 (`Codelist`, `Concept`) to can be deduced from the Ruleset signature.

## 2572 **12.3 Mapping between SDMX and VTL artefacts**

### 2573 **12.3.1 When the mapping occurs**

2574 The mapping methods between the VTL and SDMX object classes allow transforming  
2575 a SDMX definition in a VTL one and vice-versa for the artefacts to be manipulated.  
2576 It should be remembered that VTL programs (i.e. Transformation Schemes) are  
2577 represented in SDMX through the `TransformationScheme` maintainable class  
2578 which is composed of `Transformations` (nameable artefacts). Each  
2579 `Transformation` assigns the outcome of the evaluation of a VTL expression to a  
2580 result: the input operands of the expression and the result can be SDMX artefacts.  
2581 Every time a SDMX object is referenced in a VTL Transformation as an input operand,  
2582 there is the need to generate a VTL definition of the object, so that the VTL operations  
2583 can take place. This can be made starting from the SDMX definition and applying a  
2584 SDMX-VTL mapping method in the direction from SDMX to VTL. The possible mapping  
2585 methods from SDMX to VTL are described in the following paragraphs and are  
2586 conceived to allow the automatic deduction of the VTL definition of the object from the  
2587 knowledge of the SDMX definition.

2588 In the opposite direction, every time an object calculated by means of VTL must be  
2589 treated as a SDMX object (for example for exchanging it through SDMX), there is the  
2590 need of a SDMX definition of the object, so that the SDMX operations can take place.  
2591 The SDMX definition is needed for the VTL objects for which a SDMX use is  
2592 envisaged<sup>23</sup>.

2593  
2594 The mapping methods from VTL to SDMX are described in the following paragraphs  
2595 as well, however they do not allow the complete SDMX definition to be automatically  
2596 deduced from the VTL definition, more than all because the former typically contains  
2597 additional information in respect to the latter. For example, the definition of a SDMX  
2598 DSD includes also some mandatory information not available in VTL (like the concept  
2599 scheme to which the SDMX components refer, the 'usage' and 'attributeRelationship'  
2600 for the `DataAttributes` and so on). Therefore the mapping methods from VTL to SDMX  
2601 provide only a general guidance for generating SDMX definitions properly starting from  
2602 the information available in VTL, independently of how the SDMX definition it is actually  
2603 generated (manually, automatically or part and part).

---

<sup>22</sup> See also the section "VTL-DL Rulesets" in the VTL Reference Manual.

<sup>23</sup> If a calculated artefact is persistent, it needs a persistent definition, i.e. a SDMX definition in a SDMX environment. In addition, possible calculated artefact that are not persistent may require a SDMX definition, for example when the result of a non-persistent calculation is disseminated through SDMX tools (like an inquiry tool).

2604 **12.3.2 General mapping of VTL and SDMX data structures**

2605 This section makes reference to the VTL "Model for data and their structure"<sup>24</sup> and the  
2606 correspondent SDMX "Data Structure Definition"<sup>25</sup>.

2607 The main type of artefact that the VTL can manipulate is the VTL Data Set, which in  
2608 general is mapped to the SDMX `Dataflow`. This means that a VTL Transformation,  
2609 in the SDMX context, expresses the algorithm for calculating a derived `Dataflow`  
2610 starting from some already existing `Dataflows` (either collected or derived).<sup>26</sup>

2611 While the VTL Transformations are defined in term of `Dataflow` definitions, they are  
2612 assumed to be executed on instances of such `Dataflows`, provided at runtime to the  
2613 VTL engine (the mechanism for identifying the instances to be processed are not part  
2614 of the VTL specifications and depend on the implementation of the VTL-based  
2615 systems). As already said, the SDMX `Datasets` are instances of SDMX `Dataflows`,  
2616 therefore a VTL Transformation defined on some SDMX `Dataflows` can be applied  
2617 on some corresponding SDMX `Datasets`.

2618  
2619 A VTL Data Set is structured by one and just one Data Structure and a VTL Data  
2620 Structure can structure any number of Data Sets. Correspondingly, in the SDMX  
2621 context a SDMX `Dataflow` is structured by one and just one  
2622 `DataStructureDefinition` and one `DataStructureDefinition` can structure  
2623 any number of `Dataflows`.

2624  
2625 A VTL Data Set has a Data Structure made of Components, which in turn can be  
2626 Identifiers, Measures and Attributes. Similarly, a SDMX `DataflowDefinition` has  
2627 a `DataStructureDefinition` made of components that can be  
2628 `DimensionComponents`, `Measure` and `DataAttributes`. In turn, a SDMX  
2629 `DimensionComponent` can be a `Dimension` or a `TimeDimension`.  
2630 Correspondingly, in the SDMX implementation of the VTL, the VTL Identifiers can be  
2631 (optionally) distinguished in similar sub-classes (Simple Identifier, Time Identifier) even  
2632 if such a distinction is not evidenced in the VTL IM.

2633  
2634 The possible mapping options are described in more detail in the following sections.

2635 **12.3.3 Mapping from SDMX to VTL data structures**

2636 **12.3.3.1 Basic Mapping**

2637 The main mapping method from SDMX to VTL is called **Basic** mapping. This is  
2638 considered as the default mapping method and is applied unless a different method is  
2639 specified through the `VtlMappingScheme` and `VtlDataflowMapping` classes.

2640 When transforming **from SDMX to VTL**, this method consists in leaving the  
2641 components unchanged and maintaining their names and roles, according to the  
2642 following table:

SDMX	VTL
Dimension	(Simple) Identifier
TimeDimension	(Time) Identifier

<sup>24</sup> See the VTL 2.0 User Manual

<sup>25</sup> See the SDMX Standards Section 2 – Information Model

<sup>26</sup> Besides the mapping between one SDMX `Dataflow` and one VTL Data Set, it is also possible to map distinct parts of a SDMX `Dataflow` to different VTL Data Set, as explained in a following paragraph.

Measure	Measure
DataAttribute	Attribute

2643

2644

2645

2646

2647

2648

2649

2650

The SDMX `DataAttributes`, in VTL they are all considered "at data point / observation level" (i.e. dependent on all the VTL Identifiers), because VTL does not have the SDMX `AttributeRelationships`, which defines the construct to which the `DataAttribute` is related (e.g. observation, dimension or set or group of dimensions, whole data set).

With the Basic mapping, one SDMX observation<sup>27</sup> generates one VTL data point.

2651

### 12.3.3.2 Pivot Mapping

2652

2653

2654

2655

2656

2657

2658

2659

2660

An alternative mapping method from SDMX to VTL is the **Pivot** mapping, which makes sense and is different from the Basic method only for the SDMX data structures that contain a `Dimension` that plays the role of measure dimension (like in SDMX 2.1) and just one `Measure`. Through this method, these structures can be mapped to multi-measure VTL data structures. Besides that, a user may choose to use any `Dimension` acting as a list of `Measures` (e.g., a `Dimension` with indicators), either by considering the "Measure" role of a `Dimension`, or at will using any coded `Dimension`. Of course, in SDMX 3.0, this can only work when only one `Measure` is defined in the DSD.

2661

2662

2663

2664

2665

2666

2667

2668

2669

2670

2671

2672

2673

2674

2675

2676

2677

In SDMX 2.1 the `MeasureDimension` was a subclass of `DimensionComponent` like `Dimension` and `TimeDimension`. In the current SDMX version, this subclass does not exist anymore, however a `Dimension` can have the role of measure dimension (i.e. a `Dimension` that contributes to the identification of the measures). In SDMX 2.1 a `DataStructure` could have zero or one `MeasureDimensions`, in the current version of the standard, from zero to many `Dimension` may have the role of measure dimension. Hereinafter a `Dimension` that plays the role of measure dimension is referenced for simplicity as "MeasureDimension", i.e. maintaining the capital letters and the courier font even if the `MeasureDimension` is not anymore a class in the SDMX Information Model of the current SDMX version. For the sake of simplicity, the description below considers just one `Dimension` having the role of `MeasureDimension` (i.e., the more simple and common case). Nevertheless, it maintains its validity also if in the `DataStructure` there are more dimension with the role of `MeasureDimensions`: in this case what is said about the `MeasureDimension` must be applied to the combination of all the `MeasureDimensions` considered as a joint variable<sup>28</sup>.

2678

2679

2680

2681

2682

Among other things, the Pivot method provides also backward compatibility with the SDMX 2.1 data structures that contained a `MeasureDimension`.

If applied to SDMX structures that do not contain any `MeasureDimension`, this method behaves like the Basic mapping (see the previous paragraph).

<sup>27</sup> Here an SDMX observation is meant to correspond to one combination of values of the `DimensionComponents`.

<sup>28</sup> E.g., if in the data structure there exist 3 `Dimensions` C,D,E having the role of `MeasureDimension`, they should be considered as a joint `MeasureDimension`  $Z=(C,D,E)$ ; therefore when the description says "each possible value  $C_j$  of the `MeasureDimension` ..." it means "each possible combination of values  $(C_j, D_k, E_w)$  of the joint `MeasureDimension`  $Z=(C,D,E)$ ".

2683

2684

The SDMX structures that contain a `MeasureDimension` are mapped as described below (this mapping is equivalent to a pivoting operation):

2685

2686

2687

2688

2689

2690

2691

2692

2693

2694

2695

2696

2697

2698

2699

2700

2701

2702

2703

2704

2705

2706

2707

2708

2709

2710

2711

2712

2713

2714

2715

2716

2717

2718

2719

2720

2721

2722

2723

2724

2725

- A SDMX simple dimension becomes a VTL (simple) identifier and a SDMX `TimeDimension` becomes a VTL (time) identifier;
- Each possible `Code Cj` of the SDMX `MeasureDimension` is mapped to a VTL Measure, having the same name as the SDMX `Code` (i.e. `Cj`); the VTL Measure `Cj` is a new VTL component even if the SDMX data structure has not such a `Component`;
- The SDMX `MeasureDimension` is not mapped to VTL (it disappears in the VTL Data Structure);
- The SDMX `Measure` is not mapped to VTL as well (it disappears in the VTL Data Structure);
- An SDMX `DataAttribute` is mapped in different ways according to its `AttributeRelationship`:
  - If, according to the SDMX `AttributeRelationship`, the values of the `DataAttribute` do not depend on the values of the `MeasureDimension`, the SDMX `DataAttribute` becomes a VTL `Attribute` having the same name. This happens if the `AttributeRelationship` is not specified (i.e. the `DataAttribute` does not depend on any `DimensionComponent` and therefore is at data set level), or if it refers to a set (or a group) of dimensions which does not include the `MeasureDimension`;
  - Otherwise, if, according to the SDMX `AttributeRelationship`, the values of the `DataAttribute` depend on the `MeasureDimension`, the SDMX `DataAttribute` is mapped to one VTL `Attribute` for each possible `Code` of the SDMX `MeasureDimension`. By default, the names of the VTL `Attributes` are obtained by concatenating the name of the SDMX `DataAttribute` and the names of the correspondent `Code` of the `MeasureDimension` separated by underscore. For example, if the SDMX `DataAttribute` is named `DA` and the possible `Codes` of the SDMX `MeasureDimension` are named `C1`, `C2`, ..., `Cn`, then the corresponding VTL `Attributes` will be named `DA_C1`, `DA_C2`, ..., `DA_Cn` (if different names are desired, they can be achieved afterwards by renaming the `Attributes` through VTL operators).
  - Like in the Basic mapping, the resulting VTL `Attributes` are considered as dependent on all the VTL identifiers (i.e. "at data point / observation level"), because VTL does not have the SDMX notion of `Attribute Relationship`.

The summary mapping table of the "pivot" mapping from SDMX to VTL for the SDMX data structures that contain a `MeasureDimension` is the following:

<b>SDMX</b>	<b>VTL</b>
Dimension	(Simple) Identifier
TimeDimension	(Time) Identifier
MeasureDimension & one Measure	One Measure for each Code of the SDMX MeasureDimension

DataAttribute not depending on the MeasureDimension	Attribute
DataAttribute depending on the MeasureDimension	One Attribute for each Code of the SDMX MeasureDimension

2726

2727

Using this mapping method, the components of the data structure can change in the conversion from SDMX to VTL and it must be taken into account that the VTL statements can reference only the components of the resulting VTL data structure.

2728

2729

2730

2731

At observation / data point level, calling  $C_j$  ( $j=1, \dots, n$ ) the  $j^{\text{th}}$  Code of the MeasureDimension:

2732

2733

2734

- The set of SDMX observations having the same values for all the Dimensions except than the MeasureDimension become one multi-measure VTL Data Point, having one Measure for each Code  $C_j$  of the SDMX MeasureDimension;

2735

2736

2737

2738

- The values of the SDMX simple Dimensions, TimeDimension and DataAttributes not depending on the MeasureDimension (these components by definition have always the same values for all the observations of the set above) become the values of the corresponding VTL (simple) Identifiers, (time) Identifier and Attributes.

2739

2740

2741

2742

2743

- The value of the Measure of the SDMX observation belonging to the set above and having MeasureDimension= $C_j$  becomes the value of the VTL Measure  $C_j$

2744

2745

2746

- For the SDMX DataAttributes depending on the MeasureDimension, the value of the DataAttribute DA of the SDMX observation belonging to the set above and having MeasureDimension= $C_j$  becomes the value of the VTL Attribute DA\_  $C_j$

2747

2748

2749

2750

### 12.3.3.3 From SDMX DataAttributes to VTL Measures

2751

- In some cases, it may happen that the DataAttributes of the SDMX DataStructure need to be managed as Measures in VTL. Therefore, a variant of both the methods above consists in transforming all the SDMX DataAttributes in VTL Measures. When DataAttributes are converted to Measures, the two methods above are called Basic\_A2M and Pivot\_A2M (the suffix "A2M" stands for Attributes to Measures). Obviously, the resulting VTL data structure is, in general, multi-measure and does not contain Attributes.

2752

2753

2754

2755

2756

2757

2758

2759

The Basic\_A2M and Pivot\_A2M behaves respectively like the Basic and Pivot methods, except that the final VTL components, which according to the Basic and Pivot methods would have had the role of Attribute, assume instead the role of Measure.

2760

2761

2762

2763

Proper VTL features allow changing the role of specific attributes even after the SDMX to VTL mapping: they can be useful when only some of the DataAttributes need to be managed as VTL Measures.

2764

2765

2766

## 12.3.4 Mapping from VTL to SDMX data structures

2767

### 12.3.4.1 Basic Mapping

2768

The main mapping method from VTL to SDMX is called **Basic** mapping as well.

2769 This is considered as the default mapping method and is applied unless a different  
 2770 method is specified through the `VtlMappingScheme` and `VtlDataflowMapping`  
 2771 classes.

2772  
 2773 The method consists in leaving the components unchanged and maintaining their  
 2774 names and roles in SDMX, according to the following mapping table, which is the same  
 2775 as the basic mapping from SDMX to VTL, only seen in the opposite direction.

2776  
 2777 Mapping table:  
 2778

VTL	SDMX
(Simple) Identifier	Dimension
(Time) Identifier	TimeDimension
Measure	Measure
Attribute	DataAttribute

2779  
 2780 If the distinction between simple identifier and time identifier is not maintained in the  
 2781 VTL environment, the classification between `Dimension` and `TimeDimension` exists  
 2782 only in SDMX, as declared in the relevant `DataStructureDefinition`.

2783  
 2784 Regarding the Attributes, because VTL considers all of them "at observation level", the  
 2785 corresponding SDMX `DataAttributes` should be put "at observation level" as well,  
 2786 unless some different information about their `AttributeRelationship` is otherwise  
 2787 available.

2788  
 2789 Note that the basic mappings in the two directions (from SDMX to VTL and vice-versa)  
 2790 are (almost completely) reversible. In fact, if a SDMX structure is mapped to a VTL  
 2791 structure and then the latter is mapped back to SDMX, the resulting data structure is  
 2792 like the original one (apart for the `AttributeRelationship`, that can be different if  
 2793 the original SDMX structure contains `DataAttributes` that are not at observation  
 2794 level). In reverse order, if a VTL structure is mapped to SDMX and then the latter is  
 2795 mapped back to VTL, the original data structure is obtained.

2796  
 2797 As said, the resulting SDMX definitions must be compliant with the SDMX consistency  
 2798 rules. For example, the SDMX DSD must have the `AttributeRelationship` for  
 2799 the `DataAttributes`, which does not exist in VTL.

#### 2800 12.3.4.2 Unpivot Mapping

2801 An alternative mapping method from VTL to SDMX is the **Unpivot** mapping.

2802  
 2803 Although this mapping method can be used in any case, it makes major sense in case  
 2804 the VTL data structure has more than one measure component (multi-measures VTL  
 2805 structure). This is used to support the SDMX 2.1 case of a `MeasureDimension` or  
 2806 any other `Dimension` acting as a list of `Measures`, under the assumptions explained  
 2807 in section "Pivot Mapping".

2808  
 2809 The multi-measures VTL structures are converted to SDMX `Dataflows` having an  
 2810 added `MeasureDimension`, which disambiguates the VTL multiple `Measures`, and a  
 2811 new `Measure` in place of the VTL ones, containing the values of the VTL `Measures`.

2812  
 2813 The **unpivot** mapping behaves like follows:

- 2814 • like in the basic mapping, a VTL (simple) identifier becomes a SDMX
- 2815 Dimension and a VTL (time) identifier becomes a SDMX TimeDimension
- 2816 (as said, a measure identifier cannot exist in multi-measure VTL structures);
- 2817 • a MeasureDimension component called "measure\_name" is added to the
- 2818 SDMX DataStructure;
- 2819 • a Measure component called "obs\_value" is added to the SDMX
- 2820 DataStructure;
- 2821 • each VTL Measure is mapped to a Code of the SDMX MeasureDimension
- 2822 having the same name as the VTL Measure (therefore all the VTL Measure
- 2823 Components do not originate Components in the SDMX DataStructure);
- 2824 • a VTL Attribute becomes a SDMX DataAttribute having
- 2825 AttributeRelationship referred to all the SDMX
- 2826 DimensionComponents including the TimeDimension and except the
- 2827 MeasureDimension.

2828  
2829  
2830

The summary mapping table of the **unpivot** mapping method is the following:

VTL	SDMX
(Simple) Identifier	Dimension
(Time) Identifier	TimeDimension
All Measure Components	MeasureDimension (having one Code for each VTL measure component) & one Measure
Attribute	DataAttribute depending on all SDMX Dimensions including the TimeDimension and except the MeasureDimension

2831  
2832  
2833  
2834  
2835  
2836  
2837  
2838  
2839  
2840  
2841  
2842  
2843

At observation / data point level:

- a multi-measure VTL Data Point becomes a set of SDMX observations, one for each VTL Measure;
- the values of the VTL Identifiers become the values of the corresponding SDMX DimensionComponents, for all the observations of the set above;
- the name of the  $j^{\text{th}}$  VTL Measure (e.g. "Cj") becomes the Code of the SDMX MeasureDimension of the  $j^{\text{th}}$  observation of the set;
- the value of the  $j^{\text{th}}$  VTL Measure becomes the value of the SDMX Measure of the  $j^{\text{th}}$  observation of the set;
- the values of the VTL Attributes become the values of the corresponding SDMX DataAttributes (in principle for all the observations of the set above).

2844 If desired, this method can be applied also to mono-measure VTL structures, provided  
2845 that none of the VTL Components has already the role of Measure Identifier. Like in  
2846 the general case, a MeasureDimension component called "measure\_name" is  
2847 added to the SDMX DataStructure, in this case it has just one possible Code,  
2848 corresponding to the name of the unique VTL Measure. The original VTL Measure  
2849 would not become a Component in the SDMX data structure. The value of the VTL  
2850 Measure would be assigned to the unique SDMX Measure called "obs\_value".

2851 In any case, the resulting SDMX definitions must be compliant with the SDMX  
 2852 consistency rules. For example, the possible Codes of the SDMX  
 2853 MeasureDimension need to be listed in a SDMX Codelist, with proper id, agency  
 2854 and version; moreover, the SDMX DSD must have the AttributeRelationship  
 2855 for the DataAttributes, which does not exist in VTL.

#### 2856 12.3.4.3 From VTL Measures to SDMX Data Attributes

2857 More than all for the multi-measure VTL structures (having more than one Measure  
 2858 Component), it may happen that the Measures of the VTL Data Structure need to be  
 2859 managed as DataAttributes in SDMX. Therefore, a third mapping method  
 2860 consists in transforming some VTL measures in a corresponding SDMX Measures  
 2861 and all the other VTL Measures in SDMX DataAttributes. This method is called  
 2862 M2A (“M2A” stands for “Measures to DataAttributes”).

2863  
 2864 All VTL Measures maintain their names in SDMX. The VTL Measure Components that  
 2865 become SDMX DataAttributes are the ones declared as DataAttributes in the  
 2866 target SDMX data structure definition.

2867  
 2868 The mapping table is the following:  
 2869

VTL	SDMX
(Simple) Identifier	Dimension
(Time) Identifier	TimeDimension
Some Measures	Measure
Other Measures	DataAttribute
Attribute	DataAttribute

2870  
 2871 Even in this case, the resulting SDMX definitions must be compliant with the SDMX  
 2872 consistency rules. For example, the SDMX DSD must have the  
 2873 attributeRelationship for the DataAttributes, which does not exist in VTL.

#### 2874 12.3.5 Declaration of the mapping methods between data structures

2875 In order to define and understand properly VTL Transformations, the applied mapping  
 2876 methods must be specified in the SDMX structural metadata. If the default mapping  
 2877 method (Basic) is applied, no specification is needed.

2878  
 2879 A customized mapping can be defined through the VtlMappingScheme and  
 2880 VtlDataflowMapping classes (see the section of the SDMX IM relevant to the VTL).  
 2881 A VtlDataflowMapping allows specifying the mapping methods to be used for a  
 2882 specific dataflow, both in the direction from SDMX to VTL (toVtlMappingMethod)  
 2883 and from VTL to SDMX (fromVtlMappingMethod); in fact a  
 2884 VtlDataflowMapping associates the structured URN that identifies a SDMX  
 2885 Dataflow to its VTL alias and its mapping methods.

2886  
 2887 It is possible to specify the toVtlMappingMethod and fromVtlMappingMethod  
 2888 also for the conventional dataflow called "generic\_dataflow": in this case the  
 2889 specified mapping methods are intended to become the default ones, overriding the  
 2890 "Basic" methods. In turn, the toVtlMappingMethod and fromVtlMappingMethod  
 2891 declared for a specific Dataflow are intended to override the default ones for such a  
 2892 Dataflow.

2893 The `VtlMappingScheme` is a container for zero or more `VtlDataflowMapping` (it  
2894 may contain also mappings towards artefacts other than dataflows).

### 2895 **12.3.6 Mapping dataflow subsets to distinct VTL Data Sets**

2896 Until now it has been assumed to map one SMDX `Dataflow` to one VTL Data Set and  
2897 vice-versa. This mapping one-to-one is not mandatory according to VTL because a  
2898 VTL Data Set is meant to be a set of observations (data points) on a logical plane,  
2899 having the same logical data structure and the same general meaning, independently  
2900 of the possible physical representation or storage (see VTL 2.0 User Manual page 24),  
2901 therefore a SDMX `Dataflow` can be seen either as a unique set of data observations  
2902 (corresponding to one VTL Data Set) or as the union of many sets of data observations  
2903 (each one corresponding to a distinct VTL Data Set).

2904 As a matter of fact, in some cases it can be useful to define VTL operations involving  
2905 definite parts of a SDMX `Dataflow` instead than the whole.<sup>29</sup>

2906 Therefore, in order to make the coding of VTL operations simpler when applied on  
2907 parts of SDMX `Dataflows`, it is allowed to map distinct parts of a SDMX `Dataflow`  
2908 to distinct VTL Data Sets according to the following rules and conventions. This kind  
2909 of mapping is possible both from SDMX to VTL and from VTL to SDMX, as better  
2910 explained below.<sup>30</sup>

2911 Given a SDMX `Dataflow` and some predefined `Dimensions` of its `DataStructure`,  
2912 it is allowed to map the subsets of observations that have the same combination of  
2913 values for such `Dimensions` to correspondent VTL datasets.

2914 For example, assuming that the SDMX `Dataflow` DF1(1.0.0) has the `Dimensions`  
2915 INDICATOR, TIME\_PERIOD and COUNTRY, and that the user declares the  
2916 `Dimensions` INDICATOR and COUNTRY as basis for the mapping (i.e. the mapping  
2917 dimensions): the observations that have the same values for INDICATOR and  
2918 COUNTRY would be mapped to the same VTL dataset (and vice-versa).

2919 In practice, this kind mapping is obtained like follows:

- 2920
- 2921 • For a given SDMX `Dataflow`, the user (VTL definer) declares the  
2922 `DimensionComponents` on which the mapping will be based, in a given  
2923 order.<sup>31</sup> Following the example above, imagine that the user declares the  
2924 `Dimensions` INDICATOR and COUNTRY.

---

<sup>29</sup> A typical example of this kind is the validation, and more in general the manipulation, of individual time series belonging to the same `Dataflow`, identifiable through the `DimensionComponents` of the `Dataflow` except the `TimeDimension`. The coding of these kind of operations might be simplified by mapping distinct time series (i.e. different parts of a SDMX `Dataflow`) to distinct VTL Data Sets.

<sup>30</sup> Please note that this kind of mapping is only an option at disposal of the definer of VTL Transformations; in fact it remains always possible to manipulate the needed parts of SDMX `Dataflows` by means of VTL operators (e.g. “sub”, “filter”, “calc”, “union” ...), maintaining a mapping one-to-one between SDMX `Dataflows` and VTL Data Sets.

<sup>31</sup> This definition is made through the `ToVtlSubspace` and `ToVtlSpaceKey` classes and/or the `FromVtlSuperspace` and `FromVtlSpaceKey` classes, depending on the direction of the mapping (“key” means “dimension”). The mapping of `Dataflow` subsets can be applied independently in the two directions, also according to different `Dimensions`. When no `Dimension` is declared for a given direction, it is assumed that the option of mapping different parts of a SDMX `Dataflow` to different VTL Data Sets is not used.

- 2925       • The VTL Data Set is given a name using a special notation also called “ordered  
2926 concatenation” and composed of the following parts:  
2927       ○ The reference to the `SDMX Dataflow` (expressed according to the  
2928 rules described in the previous paragraphs, i.e. URN, abbreviated URN  
2929 or another alias); for example `DF(1.0.0)`;  
2930       ○ a slash (“/”) as a separator;<sup>32</sup>  
2931       ○ The reference to a specific part of the `SDMX Dataflow` above,  
2932 expressed as the concatenation of the values that the `SDMX`  
2933 `DimensionComponents` declared above must have, separated by  
2934 dots (“.”) and written in the order in which these  
2935 `DimensionComponents` are defined<sup>33</sup>. For example  
2936 `POPULATION.USA` would mean that such a VTL Data Set is mapped  
2937 to the `SDMX` observations for which the dimension `INDICATOR` is  
2938 equal to `POPULATION` and the dimension `COUNTRY` is equal to `USA`.

2939 In the VTL Transformations, this kind of dataset name must be referenced between  
2940 single quotes because the slash (“/”) is not a regular character according to the VTL  
2941 rules.

2942 Therefore, the generic name of this kind of VTL datasets would be:

2943       '`DF(1.0.0)/INDICATORvalue.COUNTRYvalue`'  
2944  
2945

2946 Where `DF(1.0.0)` is the `Dataflow` and `INDICATORvalue` and `COUNTRYvalue` are  
2947 placeholders for one value of the `INDICATOR` and `COUNTRY` dimensions.

2948 Instead the specific name of one of these VTL datasets would be:

2949       '`DF(1.0.0)/POPULATION.USA`'  
2950  
2951

2952 In particular, this is the VTL dataset that contains all the observations of the `Dataflow`  
2953 `DF(1.0.0)` for which `INDICATOR = POPULATION` and `COUNTRY = USA`.

2954 Let us now analyse the different meaning of this kind of mapping in the two mapping  
2955 directions, i.e. from `SDMX` to VTL and from VTL to `SDMX`.  
2956

2957 As already said, the mapping from `SDMX` to VTL happens when the `SDMX`  
2958 `dataflows` are operand of VTL Transformations, instead the mapping from VTL to  
2959 `SDMX` happens when the VTL Data Sets that is result of Transformations<sup>34</sup> need to be  
2960 treated as `SDMX` objects. This kind of mapping can be applied independently in the  
2961 two directions and the `Dimensions` on which the mapping is based can be different  
2962 in the two directions: these `Dimensions` are defined in the `ToVtlSpaceKey` and in  
2963 the `FromVtlSpaceKey` classes respectively.

---

<sup>32</sup> As a consequence of this formalism, a slash in the name of the VTL Data Set assumes the specific meaning of separator between the name of the `Dataflow` and the values of some of its `Dimensions`.

<sup>33</sup> This is the order in which the dimensions are defined in the `ToVtlSpaceKey` class or in the `FromVtlSpaceKey` class, depending on the direction of the mapping.

<sup>34</sup> It should be remembered that, according to the VTL consistency rules, a given VTL dataset cannot be the result of more than one VTL Transformation.

2964 First, let us see what happens in the mapping direction from SDMX to VTL, i.e. when  
 2965 parts of a SDMX *Dataflow* (e.g. DF1(1.0.0)) need to be mapped to distinct VTL Data  
 2966 Sets that are operand of some VTL Transformations.

2967 As already said, each VTL Data Set is assumed to contain all the observations of the  
 2968 SDMX *Dataflow* having *INDICATOR=INDICATORvalue* and *COUNTRY=COUNTRYvalue*. For example, the VTL dataset 'DF1(1.0.0)/POPULATION.USA'  
 2969 would contain all the observations of DF1(1.0.0) having *INDICATOR = POPULATION*  
 2970 and *COUNTRY = USA*.  
 2971

2972 In order to obtain the data structure of these VTL Data Sets from the SDMX one, it is  
 2973 assumed that the SDMX *DimensionComponents* on which the mapping is based are  
 2974 dropped, i.e. not maintained in the VTL data structure; this is possible because their  
 2975 values are fixed for each one of the invoked VTL Data Sets<sup>35</sup>. After that, the mapping  
 2976 method from SDMX to VTL specified for the *Dataflow* DF1(1.0.0) is applied (i.e.  
 2977 basic, pivot ...).  
 2978

2979 In the example above, for all the datasets of the kind  
 2980 'DF1(1.0.0)/*INDICATORvalue.COUNTRYvalue*', the dimensions *INDICATOR* and  
 2981 *COUNTRY* would be dropped so that the data structure of all the resulting VTL Data  
 2982 Sets would have the identifier *TIME\_PERIOD* only.

2983 It should be noted that the desired VTL Data Sets (i.e. of the kind 'DF1(1.0.0)/  
 2984 *INDICATORvalue.COUNTRYvalue*') can be obtained also by applying the VTL  
 2985 operator "**sub**" (subspace) to the *Dataflow* DF1(1.0.0), like in the following VTL  
 2986 expression:

```
2987 'DF1(1.0.0)/POPULATION.USA' :=
2988 DF1(1.0.0) [ sub INDICATOR="POPULATION", COUNTRY="USA" ];
2989
2990 'DF1(1.0.0)/POPULATION.CANADA' :=
2991 DF1(1.0.0) [ sub INDICATOR="POPULATION", COUNTRY="CANADA" ];
2992
2993 ... ..
```

2994 In fact the VTL operator "sub" has exactly the same behaviour. Therefore, mapping  
 2995 different parts of a SDMX *Dataflow* to different VTL Data Sets in the direction from  
 2996 SDMX to VTL through the ordered concatenation notation is equivalent to a proper use  
 2997 of the operator "**sub**" on such a *Dataflow*.<sup>36</sup>

2998 In the direction from SDMX to VTL it is allowed to omit the value of one or more  
 2999 *DimensionComponents* on which the mapping is based, but maintaining all the  
 3000 separating dots (therefore it may happen to find two or more consecutive dots and dots

---

<sup>35</sup> If these *DimensionComponents* would not be dropped, the various VTL Data Sets resulting from this kind of mapping would have non-matching values for the Identifiers corresponding to the mapping Dimensions (e.g. *POPULATION* and *COUNTRY*). As a consequence, taking into account that the typical binary VTL operations at dataset level (+, -, \*, / and so on) are executed on the observations having matching values for the identifiers, it would not be possible to compose the resulting VTL datasets one another (e.g. it would not be possible to calculate the population ratio between USA and CANADA).

<sup>36</sup> In case the ordered concatenation notation is used, the VTL Transformation described above, e.g. 'DF1(1.0)/POPULATION.USA' := DF1(1.0) [ sub INDICATOR="POPULATION", COUNTRY="USA"], is implicitly executed. In order to test the overall compliance of the VTL program to the VTL consistency rules, it has to be considered as part of the VTL program even if it is not explicitly coded.

3001 in the beginning or in the end). The absence of value means that for the corresponding  
 3002 Dimension all the values are kept and the Dimension is not dropped.  
 3003 For example, 'DF(1.0.0)/POPULATION.' (note the dot in the end of the name) is the  
 3004 VTL dataset that contains all the observations of the `Dataflow` DF(1.0.0) for which  
 3005 `INDICATOR = POPULATION` and `COUNTRY = any value`.  
 3006

3007 This is equivalent to the application of the VTL "sub" operator only to the identifier  
 3008 `INDICATOR`:

```
3009     'DF1(1.0.0)/POPULATION.' :=
3010     DF1(1.0.0) [ sub INDICATOR="POPULATION" ];
```

3012 Therefore the VTL Data Set 'DF1(1.0.0)/POPULATION.' would have the identifiers  
 3013 `COUNTRY` and `TIME_PERIOD`.

3015 Heterogeneous invocations of the same `Dataflow` are allowed, i.e. omitting different  
 3016 `Dimensions` in different invocations.

3017 Let us now analyse the mapping direction from VTL to SDMX.

3018 In this situation, distinct parts of a SDMX `Dataflow` are calculated as distinct VTL  
 3019 datasets, under the constraint that they must have the same VTL data structure.

3020 For example, let us assume that the VTL programmer wants to calculate the SDMX  
 3021 `Dataflow` DF2(1.0.0) having the `Dimensions` `TIME_PERIOD`, `INDICATOR`, and  
 3022 `COUNTRY` and that such a programmer finds it convenient to calculate separately the  
 3023 parts of DF2(1.0.0) that have different combinations of values for `INDICATOR` and  
 3024 `COUNTRY`:

- 3025 • each part is calculated as a VTL derived Data Set, result of a dedicated VTL  
 3026 Transformation;<sup>37</sup>
- 3027 • the data structure of all these VTL Data Sets has the `TIME_PERIOD` identifier  
 3028 and does not have the `INDICATOR` and `COUNTRY` identifiers.<sup>38</sup>

3029 Under these hypothesis, such derived VTL Data Sets can be mapped to DF2(1.0.0) by  
 3030 declaring the `DimensionComponents` `INDICATOR` and `COUNTRY` as mapping  
 3031 dimensions<sup>39</sup>.  
 3032

3033 The corresponding VTL Transformations, assuming that the result needs to be  
 3034 persistent, would be of this kind:<sup>40</sup>

```
3035     'DF2(1.0.0)/INDICATORvalue.COUNTRYvalue' <- expression
```

3036  
 3037 Some examples follow, for some specific values of `INDICATOR` and `COUNTRY`:

```
3038     'DF2(1.0.0)/GDPPERCAPITA.USA' <- expression11;
3039     'DF2(1.0.0)/GDPPERCAPITA.CANADA' <- expression12;
```

---

<sup>37</sup> If the whole DF2(1.0) is calculated by means of just one VTL Transformation, then the mapping between the SDMX `Dataflow` and the corresponding VTL dataset is one-to-one and this kind of mapping (one SDMX `Dataflow` to many VTL datasets) does not apply.

<sup>38</sup> This is possible as each VTL dataset corresponds to one particular combination of values of `INDICATOR` and `COUNTRY`.

<sup>39</sup> The mapping dimensions are defined as `FromVtlSpaceKeys` of the `FromVtlSuperSpace` of the `VtlDataflowMapping` relevant to DF2(1.0).

<sup>40</sup> the symbol of the VTL persistent assignment is used (<-)

```

3041      ... ..
3042      'DF2(1.0.0)/POPGROWTH.USA'          <- expression21;
3043      'DF2(1.0.0)/POPGROWTH.CANADA'      <- expression22;
3044      ... ..

```

3045  
3046 As said, it is assumed that these VTL derived Data Sets have the TIME\_PERIOD as  
3047 the only identifier. In the mapping from VTL to SMDX, the Dimensions INDICATOR  
3048 and COUNTRY are added to the VTL data structure on order to obtain the SDMX one,  
3049 with the following values respectively:

3050	VTL dataset	INDICATOR value	COUNTRY value
3051			
3052			
3053	'DF2(1.0.0)/GDPPERCAPITA.USA'	GDPPERCAPITA	USA
3054	'DF2(1.0.0)/GDPPERCAPITA.CANADA'	GDPPERCAPITA	CANADA
3055	... ..		
3056	'DF2(1.0.0)/POPGROWTH.USA'	POPGROWTH	USA
3057	'DF2(1.0.0)/POPGROWTH.CANADA'	POPGROWTH	CANADA
3058	... ..		
3059			

3060 It should be noted that the application of this many-to-one mapping from VTL to SDMX  
3061 is equivalent to an appropriate sequence of VTL Transformations. These use the VTL  
3062 operator "calc" to add the proper VTL identifiers (in the example, INDICATOR and  
3063 COUNTRY) and to assign to them the proper values and the operator "union" in order  
3064 to obtain the final VTL dataset (in the example DF2(1.0.0)), that can be mapped one-  
3065 to-one to the homonymous SDMX Dataflow. Following the same example, these  
3066 VTL Transformations would be:

```

3067 DF2bis_GDPPERCAPITA_USA      := 'DF2(1.0.0)/GDPPERCAPITA.USA'
3068                               [calc identifier INDICATOR := "GDPPERCAPITA",
3069                               identifier COUNTRY := "USA"];
3070
3071 DF2bis_GDPPERCAPITA_CANADA := 'DF2(1.0.0)/GDPPERCAPITA.CANADA'
3072                               [calc identifier INDICATOR:= "GDPPERCAPITA",
3073                               identifier COUNTRY:= "CANADA"];
3074 ... ..
3075
3076 DF2bis_POPGROWTH_USA        := 'DF2(1.0.0)/POPGROWTH.USA'
3077                               [calc identifier INDICATOR := "POPGROWTH",
3078                               identifier COUNTRY := "USA"];
3079
3080 DF2bis_POPGROWTH_CANADA'    := 'DF2(1.0.0)/POPGROWTH.CANADA'
3081                               [calc identifier INDICATOR := "POPGROWTH",
3082                               identifier COUNTRY := "CANADA"];
3083
3084 ... ..
3085
3086 DF2(1.0) <- UNION            (DF2bis_GDPPERCAPITA_USA',
3087                               DF2bis_GDPPERCAPITA_CANADA',
3088                               ... ,
3089                               DF2bis_POPGROWTH_USA',
3090                               DF2bis_POPGROWTH_CANADA'
3091                               ...);

```

3090 In other words, starting from the datasets explicitly calculated through VTL (in the  
3091 example 'DF2(1.0)/GDPPERCAPITA.USA' and so on), the first step consists in  
3092 calculating other (non-persistent) VTL datasets (in the example  
3093 DF2bis\_GDPPERCAPITA\_USA and so on) by adding the identifiers INDICATOR and

3094 COUNTRY with the desired values (*INDICATORvalue* and *COUNTRYvalue*). Finally,  
 3095 all these non-persistent Data Sets are united and give the final result DF2(1.0)<sup>41</sup>, which  
 3096 can be mapped one-to-one to the homonymous SDMX *Dataflow* having the  
 3097 dimension components TIME\_PERIOD, INDICATOR and COUNTRY.

3098 Therefore, mapping different VTL datasets having the same data structure to different  
 3099 parts of a SDMX *Dataflow*, i.e. in the direction from VTL to SDMX, through the  
 3100 ordered concatenation notation is equivalent to a proper use of the operators “calc”  
 3101 and “union” on such datasets.<sup>42</sup>

3102 It is worth noting that in the direction from VTL to SDMX it is mandatory to specify the  
 3103 value for every Dimension on which the mapping is based (in other word, in the name  
 3104 of the calculated VTL dataset is not possible to omit the value of some of the  
 3105 Dimensions).  
 3106

### 3107 12.3.7 Mapping variables and value domains between VTL and SDMX

3108 With reference to the VTL “model for Variables and Value domains”, the following  
 3109 additional mappings have to be considered:

VTL	SDMX
<b>Data Set Component</b>	Although this abstraction exists in SDMX, it does not have an explicit definition and correspond to a <b>Component</b> (either a <b>DimensionComponent</b> or a <b>Measure</b> or a <b>DataAttribute</b> ) belonging to one specific <b>Dataflow</b> <sup>43</sup>
<b>Represented Variable</b>	<b>Concept</b> with a definite <b>Representation</b>
<b>Value Domain</b>	<b>Representation</b> (see the Structure Pattern in the Base Package)
<b>Enumerated Value Domain / Code List</b>	<b>Codelist</b>
<b>Code</b>	<b>Code</b> (for enumerated <b>DimensionComponent</b> , <b>Measure</b> , <b>DataAttribute</b> )
<b>Described Value Domain</b>	non-enumerated <b>Representation</b> (having <b>Facets</b> / <b>ExtendedFacets</b> , see the Structure Pattern in the Base Package)
<b>Value</b>	Although this abstraction exists in SDMX, it does not have an explicit definition and correspond to a <b>Code</b> of a <b>Codelist</b> (for enumerated <b>Representations</b> ) or

<sup>41</sup> The result is persistent in this example but it can be also non persistent if needed.

<sup>42</sup> In case the ordered concatenation notation from VTL to SDMX is used, the set of Transformations described above is implicitly performed; therefore, in order to test the overall compliance of the VTL program to the VTL consistency rules, these implicit Transformations have to be considered as part of the VTL program even if they are not explicitly coded.

<sup>43</sup> Through SDMX **Constraints**, it is possible to specify the values that a **Component** of a **Dataflow** can assume.

	to a valid <b>value</b> (for non-enumerated Representations)
<b>Value Domain Subset / Set</b>	This abstraction does not exist in SDMX
<b>Enumerated Value Domain Subset / Enumerated Set</b>	This abstraction does not exist in SDMX
<b>Described Value Domain Subset / Described Set</b>	This abstraction does not exist in SDMX
<b>Set list</b>	This abstraction does not exist in SDMX

3110

3111

3112

3113

3114

3115

3116

3117

3118

3119

3120

3121

3122

3123

3124

3125

3126

3127

3128

3129

3130

3131

3132

3133

3134

3135

3136

3137

3138

3139

3140

3141

3142

3143

3144

3145

The main difference between VTL and SDMX relies on the fact that the VTL artefacts for defining subsets of Value Domains do not exist in SDMX, therefore the VTL features for referring to predefined subsets are not available in SDMX. These artefacts are the Value Domain Subset (or Set), either enumerated or described, the Set List (list of values belonging to enumerated subsets) and the Data Set Component (aimed at defining the set of values that the Component of a Data Set can take, possibly a subset of the codes of Value Domain).

Another difference consists in the fact that all Value Domains are considered as identifiable objects in VTL either if enumerated or not, while in SDMX the `CodeList` (corresponding to a VTL enumerated Value Domain) is identifiable, while the SDMX non-enumerated `Representation` (corresponding to a VTL non-enumerated Value Domain) is not identifiable. As a consequence, the definition of the VTL Rulesets, which in VTL can refer either to enumerated or non-enumerated value domains, in SDMX can refer only to enumerated Value Domains (i.e. to SDMX `CodeLists`).

As for the mapping between VTL variables and SDMX `Concepts`, it should be noted that these artefacts do not coincide perfectly. In fact, the VTL variables are represented variables, defined always on the same Value Domain ("Representation" in SDMX) independently of the data set / data structure in which they appear<sup>44</sup>, while the SDMX `Concepts` can have different `Representations` in different `DataStructures`.<sup>45</sup> This means that one SDMX `Concept` can correspond to many VTL Variables, one for each representation the `Concept` has.

Therefore, it is important to be aware that some VTL operations (for example the binary operations at data set level) are consistent only if the components having the same names in the operated VTL Data Sets have also the same representation (i.e. the same Value Domain as for VTL). For example, it is possible to obtain correct results from the VTL expression

$DS\_c := DS\_a + DS\_b$  (where `DS_a`, `DS_b`, `DS_c` are VTL Data Sets) if the matching components in `DS_a` and `DS_b` (e.g. `ref_date`, `geo_area`, `sector ...`) refer to the same general representation. In simpler words, `DS_a` and `DS_b` must use the same values/codes (for `ref_date`, `geo_area`, `sector ...`), otherwise the relevant values would not match and the result of the operation would be wrong.

As mentioned, the property above is not enforced by construction in SDMX, and different representations of the same `Concept` can be not compatible one another (for example, it may happen that `geo_area` is represented by ISO-alpha-3 codes in `DS_a` and by ISO alpha-2 codes in `DS_b`). Therefore, it will be up to the definer of VTL

<sup>44</sup> By using represented variables, VTL can assume that data structures having the same variables as identifiers can be composed one another because the correspondent values can match.

<sup>45</sup> A `Concept` becomes a `Component` in a `DataStructureDefinition`, and `Components` can have different `LocalRepresentations` in different `DataStructureDefinitions`, also overriding the (possible) base representation of the `Concept`.

3146 Transformations to ensure that the VTL expressions are consistent with the actual  
3147 representations of the correspondent SDMX *Concepts*.

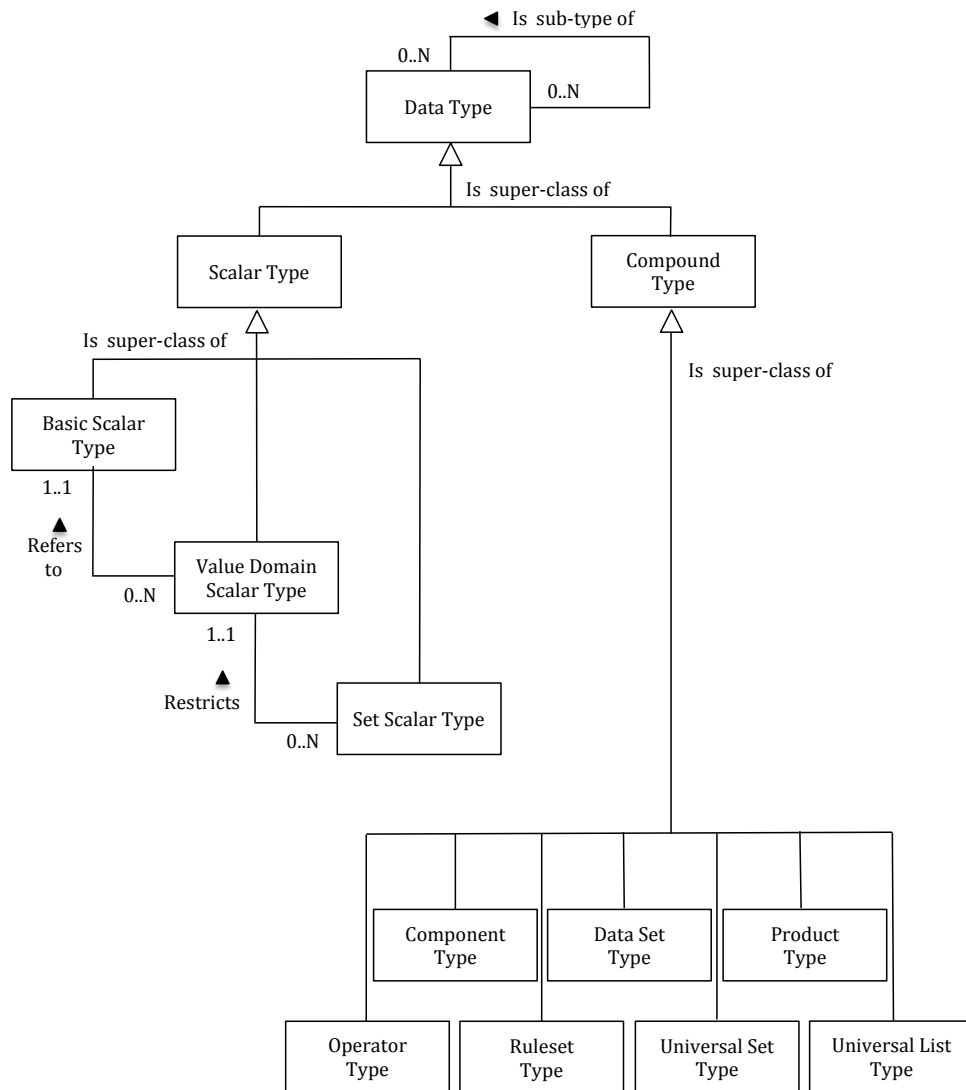
3148 It remains up to the SDMX-VTL definer also the assurance of the consistency between  
3149 a VTL Ruleset defined on *Variables* and the SDMX *Components* on which the Ruleset  
3150 is applied. In fact, a VTL Ruleset is expressed by means of the values of the *Variables*  
3151 (i.e. SDMX *Concepts*), i.e. assuming definite representations for them (e.g. ISO-  
3152 alpha-3 for country). If the Ruleset is applied to SDMX *Components* that have the same  
3153 name of the *Concept* they refer to but different representations (e.g. ISO-alpha-2 for  
3154 country), the Ruleset cannot work properly.  
3155

## 3156 **12.4 Mapping between SDMX and VTL Data Types**

### 3157 **12.4.1 VTL Data types**

3158 According to the VTL User Guide the possible operations in VTL depend on the data  
3159 types of the artefacts. For example, numbers can be multiplied but text strings cannot.  
3160 In the VTL Transformations, the compliance between the operators and the data types  
3161 of their operands is statically checked, i.e., violations result in compile-time errors.  
3162

3163 The VTL data types are sub-divided in scalar types (like integers, strings, etc.), which  
3164 are the types of the scalar values, and compound types (like Data Sets, Components,  
3165 Rulesets, etc.), which are the types of the compound structures. See below the  
3166 diagram of the VTL data types, taken from the VTL User Manual:



**Figure 22 – VTL Data Types**

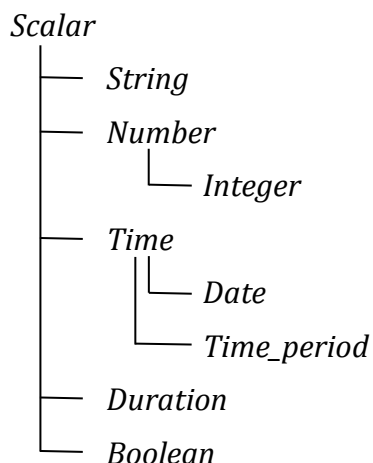
3167

3168

3169

3170 The VTL scalar types are in turn subdivided in basic scalar types, which are elementary  
 3171 (not defined in term of other data types) and Value Domain and Set scalar types, which  
 3172 are defined in terms of the basic scalar types.

3173 The VTL basic scalar types are listed below and follow a hierarchical structure in terms  
 3174 of supersets/subsets (e.g. "scalar" is the superset of all the basic scalar types):



3175  
3176

**Figure 23 – VTL Basic Scalar Types**

3177 **12.4.2 VTL basic scalar types and SDMX data types**

3178 The VTL assumes that a basic scalar type has a unique internal representation and  
3179 can have more external representations.

3180 The internal representation is the format used within a VTL system to represent (and  
3181 process) all the scalar values of a certain type. In principle, this format is hidden and  
3182 not necessarily known by users. The external representations are instead the external  
3183 formats of the values of a certain basic scalar type, i.e. the formats known by the users.  
3184 For example, the internal representation of the dates can be an integer counting the  
3185 days since a predefined date (e.g. from 01/01/4713 BC up to 31/12/5874897 AD like  
3186 in Postgres) while two possible external representations are the formats YYYY-MM-  
3187 GG and MM-GG-YYYY (e.g. respectively 2010-12-31 and 12-31-2010).

3188 The internal representation is the reference format that allows VTL to operate on more  
3189 values of the same type (for example on more dates) even if such values have different  
3190 external formats: these values are all converted to the unique internal representation  
3191 so that they can be composed together (e.g. to find the more recent date, to find the  
3192 time span between these dates and so on).

3193 The VTL assumes that a unique internal representation exists for each basic scalar  
3194 type but does not prescribe any particular format for it, leaving the VTL systems free  
3195 to using they preferred or already existing internal format. By consequence, in VTL the  
3196 basic scalar types are abstractions not associated to a specific format.

3197 SDMX data types are conceived instead to support the data exchange, therefore they  
3198 do have a format, which is known by the users and correspond, in VTL terms, to  
3199 external representations. Therefore, for each VTL basic scalar type there can be more  
3200 SDMX data types (the latter are explained in the section "General Notes for  
3201 Implementers" of this document and are actually much more numerous than the  
3202 former).

3203  
3204 The following paragraphs describe the mapping between the SDMX data types and  
3205 the VTL basic scalar types. This mapping shall be presented in the two directions of  
3206 possible conversion, i.e. from SDMX to VTL and vice-versa.

3207  
3208 The conversion from SDMX to VTL happens when an SDMX artefact acts as inputs of  
3209 a VTL Transformation. As already said, in fact, at compile time the VTL needs to know  
3210 the VTL type of the operands in order to check their compliance with the VTL operators

3211 and at runtime it must convert the values from their external (SDMX) representations  
3212 to the corresponding internal (VTL) ones.

3213

3214 The opposite conversion, i.e. from VTL to SDMX, happens when a VTL result, i.e. a  
3215 VTL Data Set output of a Transformation, must become a SDMX artefact (or part of it).

3216 The values of the VTL result must be converted into the desired (SDMX) external  
3217 representations (data types) of the SDMX artefact.

### 3218 **12.4.3 Mapping SDMX data types to VTL basic scalar types**

3219 The following table describes the default mapping for converting from the SDMX data  
3220 types to the VTL basic scalar types.

SDMX data type (BasicComponentDataType)	Default VTL basic scalar type
String (string allowing any character)	string
Alpha (string which only allows A-z)	string
AlphaNumeric (string which only allows A-z and 0-9)	string
Numeric (string which only allows 0-9, but is not numeric so that is can having leading zeros)	string
BigInteger (corresponds to XML Schema xs:integer datatype; infinite set of integer values)	integer
Integer (corresponds to XML Schema xs:int datatype; between -2147483648 and +2147483647 (inclusive))	integer
Long (corresponds to XML Schema xs:long datatype; between -9223372036854775808 and +9223372036854775807 (inclusive))	integer
Short (corresponds to XML Schema xs:short datatype; between -32768 and -32767 (inclusive))	integer
Decimal (corresponds to XML Schema xs:decimal datatype; subset of real numbers that can be represented as decimals)	number
Float (corresponds to XML Schema xs:float datatype; patterned after the IEEE single-precision 32-bit floating point type)	number
Double (corresponds to XML Schema xs:double datatype; patterned after the IEEE double-precision 64-bit floating point type)	number
Boolean (corresponds to the XML Schema xs:boolean datatype; support the mathematical concept of binary-valued logic: {true, false})	boolean

URI (corresponds to the XML Schema xs:anyURI; absolute or relative Uniform Resource Identifier Reference)	string
Count (an integer following a sequential pattern, increasing by 1 for each occurrence)	integer
InclusiveValueRange (decimal number within a closed interval, whose bounds are specified in the SDMX representation by the facets minValue and maxValue)	number
ExclusiveValueRange (decimal number within an open interval, whose bounds are specified in the SDMX representation by the facets minValue and maxValue)	number
Incremental (decimal number the increased by a specific interval (defined by the interval facet), which is typically enforced outside of the XML validation)	number
ObservationalTimePeriod (superset of StandardTimePeriod and TimeRange)	time
StandardTimePeriod (superset of BasicTimePeriod and ReportingTimePeriod)	time
BasicTimePeriod (superset of GregorianTimePeriod and DateTime)	date
GregorianTimePeriod (superset of GregorianYear, GregorianYearMonth, and GregorianDay)	date
GregorianYear (YYYY)	date
GregorianYearMonth / GregorianMonth (YYYY-MM)	date
GregorianDay (YYYY-MM-DD)	date
ReportingTimePeriod (superset of RepostingYear, ReportingSemester, ReportingTrimester, ReportingQuarter, ReportingMonth, ReportingWeek, ReportingDay)	time_period
ReportingYear (YYYY-A1 – 1 year period)	time_period
ReportingSemester (YYYY-Ss – 6 month period)	time_period
ReportingTrimester (YYYY-Tt – 4 month period)	time_period
ReportingQuarter (YYYY-Qq – 3 month period)	time_period
ReportingMonth (YYYY-Mmm – 1 month period)	time_period
ReportingWeek	time_period

(YYYY-Www – 7 day period; following ISO 8601 definition of a week in a year)	
ReportingDay (YYYY-Dddd – 1 day period)	time_period
DateTime (YYYY-MM-DDThh:mm:ss)	date
TimeRange (YYYY-MM-DD(Thh:mm:ss)?/<duration>)	time
Month (--MM; specifies a month independent of a year; e.g. February is black history month in the United States)	string
MonthDay (--MM-DD; specifies a day within a month independent of a year; e.g. Christmas is December 25 <sup>th</sup> ; used to specify reporting year start day)	string
Day (---DD; specifies a day independent of a month or year; e.g. the 15 <sup>th</sup> is payday)	string
Time (hh:mm:ss; time independent of a date; e.g. coffee break is at 10:00 AM)	string
Duration (corresponds to XML Schema xs:duration datatype)	duration
XHTML	Metadata type – not applicable
KeyValues	Metadata type – not applicable
IdentifiableReference	Metadata type – not applicable
DataSetReference	Metadata type – not applicable

3221 **Figure 14 – Mappings from SDMX data types to VTL Basic Scalar Types**

3222 When VTL takes in input SDMX artefacts, it is assumed that a type conversion  
 3223 according to the table above always happens. In case a different VTL basic scalar type  
 3224 is desired, it can be achieved in the VTL program taking in input the default VTL basic  
 3225 scalar type above and applying to it the VTL type conversion features (see the implicit  
 3226 and explicit type conversion and the "cast" operator in the VTL Reference Manual).

3227 **12.4.4 Mapping VTL basic scalar types to SDMX data types**

3228 The following table describes the default conversion from the VTL basic scalar types  
 3229 to the SDMX data types .

VTL basic scalar type	Default SDMX data type (BasicComponentDataType)	Default output format
String	String	Like XML (xs:string)
Number	Float	Like XML (xs:float)
Integer	Integer	Like XML (xs:int)
Date	DateTime	YYYY-MM-DDT00:00:00Z
Time	StandardTimePeriod	<date>/<date> (as defined above)

time_period	ReportingTimePeriod (StandardReportingPeriod)	YYYY-Pppp (according to SDMX )
Duration	Duration	Like XML (xs:duration) PnYnMnDTnHnMnS
Boolean	Boolean	Like XML (xs:boolean) with the values "true" or "false"

**Figure 14 – Mappings from SDMX data types to VTL Basic Scalar Types**

3230

3231 In case a different default conversion is desired, it can be achieved through the  
3232 CustomTypeScheme and CustomType artefacts (see also the section  
3233 Transformations and Expressions of the SDMX information model).

3234

3235 The custom output formats can be specified by means of the VTL formatting mask  
3236 described in the section "Type Conversion and Formatting Mask" of the VTL Reference  
3237 Manual. Such a section describes the masks for the VTL basic scalar types "number",  
3238 "integer", "date", "time", "time\_period" and "duration" and gives examples. As for the  
3239 types "string" and "boolean" the VTL conventions are extended with some other special  
3240 characters as described in the following table.

VTL special characters for the formatting masks	
Number	
D	one numeric digit (if the scientific notation is adopted, D is only for the mantissa)
E	one numeric digit (for the exponent of the scientific notation)
. (dot)	possible separator between the integer and the decimal parts.
, (comma)	possible separator between the integer and the decimal parts.
Time and duration	
C	century
Y	year
S	semester
Q	quarter
M	month
W	week
D	day
h	hour digit (by default on 24 hours)
M	minute
S	second
D	decimal of second
P	period indicator (representation in one digit for the duration)
P	number of the periods specified in the period indicator
AM/PM	indicator of AM / PM (e.g. am/pm for "am" or "pm")
MONTH	uppercase textual representation of the month (e.g., JANUARY for January)
DAY	uppercase textual representation of the day (e.g., MONDAY for Monday)
Month	lowercase textual representation of the month (e.g., january)
Day	lowercase textual representation of the month (e.g., monday)
Month	First character uppercase, then lowercase textual representation of the month (e.g., January)

Day	First character uppercase, then lowercase textual representation of the day using (e.g. Monday)
String	
X	any string character
Z	any string character from "A" to "z"
9	any string character from "0" to "9"
Boolean	
B	Boolean using "true" for True and "false" for False
1	Boolean using "1" for True and "0" for False
0	Boolean using "0" for True and "1" for False
Other qualifiers	
*	an arbitrary number of digits (of the preceding type)
+	at least one digit (of the preceding type)
( )	optional digits (specified within the brackets)
\	prefix for the special characters that must appear in the mask
N	fixed number of digits used in the preceding textual representation of the month or the day

3241

3242

3243

3244

3245

3246

3247

The default conversion, either standard or customized, can be used to deduce automatically the representation of the components of the result of a VTL Transformation. In alternative, the representation of the resulting SDMX Dataflow can be given explicitly by providing its `DataStructureDefinition`. In other words, the representation specified in the DSD, if available, overrides any default conversion<sup>46</sup>.

3248

#### 12.4.5 Null Values

3249

3250

3251

3252

3253

3254

3255

3256

3257

3258

3259

3260

3261

3262

In the conversions from SDMX to VTL it is assumed by default that a missing value in SDMX becomes a NULL in VTL. After the conversion, the NULLs can be manipulated through the proper VTL operators.

On the other side, the VTL programs can produce in output NULL values for Measures and Attributes (Null values are not allowed in the Identifiers). In the conversion from VTL to SDMX, it is assumed that a NULL in VTL becomes a missing value in SDMX. In the conversion from VTL to SDMX, the default assumption can be overridden, separately for each VTL basic scalar type, by specifying which the value that represents the NULL in SDMX is. This can be specified in the attribute "nullValue" of the `CustomType` artefact (see also the section Transformations and Expressions of the SDMX information model). A `CustomType` belongs to a `CustomTypeScheme`, which can be referenced by one or more `TransformationScheme` (i.e. VTL programs). The overriding assumption is applied for all the SDMX Dataflows calculated in the `TransformationScheme`.

<sup>46</sup> The representation given in the DSD should obviously be compatible with the VTL data type.

3263 **12.4.6 Format of the literals used in VTL Transformations**

3264 The VTL programs can contain literals, i.e. specific values of certain data types written  
3265 directly in the VTL definitions or expressions. The VTL does not prescribe a specific  
3266 format for the literals and leave the specific VTL systems and the definers of VTL  
3267 Transformations free of using their preferred formats.

3268 Given this discretion, it is essential to know which are the external representations  
3269 adopted for the literals in a VTL program, in order to interpret them correctly. For  
3270 example, if the external format for the dates is YYYY-MM-DD the date literal 2010-01-  
3271 02 has the meaning of 2<sup>nd</sup> January 2010, instead if the external format for the dates is  
3272 YYYY-DD-MM the same literal has the meaning of 1<sup>st</sup> February 2010.

3273 Hereinafter, i.e. in the SDMX implementation of the VTL, it is assumed that the literals  
3274 are expressed according to the "default output format" of the table of the previous  
3275 paragraph ("Mapping VTL basic scalar types to SDMX data types") unless otherwise  
3276 specified.

3277 A different format can be specified in the attribute "vtlLiteralFormat" of the  
3278 CustomType artefact (see also the section Transformations and Expressions of the  
3279 SDMX information model).

3280 Like in the case of the conversion of NULLs described in the previous paragraph, the  
3281 overriding assumption is applied, for a certain VTL basic scalar type, if a value is found  
3282 for the vtlLiteralFormat attribute of the CustomType of such VTL basic scalar  
3283 type. The overriding assumption is applied for all the literals of a related VTL  
3284 TransformationScheme.

3285 In case a literal is operand of a VTL Cast operation, the format specified in the Cast  
3286 overrides all the possible otherwise specified formats.

3287 **13 Structure Mapping**

3288 **13.1 Introduction**

3289 The purpose of SDMX structure mapping is to transform datasets from one  
 3290 dimensionality to another. In practice, this means that the input and output datasets  
 3291 conform to different Data Structure Definition.

3292 Structure mapping does not alter the observation values and is not intended to perform  
 3293 any aggregations or calculations.

3294 An input series maps to:

- 3295 a. Exactly one output series; or
- 3296 b. Multiple output series with different Series Keys, but the same observation  
 3297 values; or
- 3298 c. Zero output series where no source rule matches the input Component values.

3299  
 3300 Typical use cases include:

- 3301 • Transforming received data into a common internal structure;
- 3302 • Transforming reported data into the data collector's preferred structure;
- 3303 • Transforming unidimensional datasets<sup>47</sup> to multi-dimensional; and
- 3304 • Transforming internal datasets with a complex structure to a simpler structure  
 3305 with fewer dimensions suitable for dissemination.

3306 **13.21-1 structure maps**

3307 1-1 (pronounced 'one to one') mappings support the simple use case where the value  
 3308 of a Component in the source structure is translated to a different value in the target,  
 3309 usually where different classification schemes are used for the same Concept.

3310 In the example below, ISO 2-character country codes are mapped to their ISO 3-  
 3311 character equivalent.  
 3312  
 3313

Country	Alpha-2 code	Alpha-3 code
Afghanistan	AF	AFG
Albania	AL	ALB
Algeria	DZ	DZA
American Samoa	AS	ASM
Andorra	AD	AND
etc...		

3314 Different source values can also map to the same target value, for example when  
 3315 deriving regions from country codes.  
 3316  
 3317

---

<sup>47</sup> Unidimensional datasets are those with a single 'indicator' or 'series code' dimension.

Source Component: REF_AREA	Target Component: REGION
FR	EUR
DE	EUR
IT	EUR
ES	EUR
BE	EUR

3318

3319 **13.3 N-n structure maps**

3320 N-n (pronounced 'N to N') mappings describe rules where a specified combination of  
 3321 values in multiple source Components map to specified values in one or more target  
 3322 Components. For example, when mapping a partial Series Key from a highly  
 3323 multidimensional cube (like Balance of Payments) to a single 'Indicator' Dimension in  
 3324 a target Data Structure.

3325

3326

Example:

Rule	Source	Target
1	If FREQUENCY=A; and ADJUSTMENT=N; and MATURITY=L.	Set INDICATOR=A_N_L
2	If FREQUENCY=M; and ADJUSTMENT=S_A1; and MATURITY=TY12.	Set INDICATOR=MON_SAX_12

3327

3328

N-n rules can also set values for multiple source Components.

Rule	Source	Target
1	If FREQUENCY=A; and ADJUSTMENT=N; and MATURITY=L.	Set INDICATOR=A_N_L, STATUS=QXR15, NOTE="Unadjusted".
2	If FREQUENCY=M; and ADJUSTMENT=S_A1; and MATURITY=TY12.	Set INDICATOR=MON_SAX_12, STATUS=MPM12, NOTE="Seasonally Adjusted"

3329

3330 **13.4 Ambiguous mapping rules**

3331 A structure map is ambiguous if the rules result in a dataset containing multiple series  
3332 with the same Series Key.

3333  
3334 A simple example mapping a source dataset with a single dimension to one with  
3335 multiple dimensions is shown below:  
3336

Source	Target	Output Series Key
SERIES_CODE=XMAN_Z_21	Dimensions INDICATOR=XM FREQ=A ADJUSTMENT=N  Attributes UNIT_MEASURE=_Z COMP_ORG=21	XM:A:N
SERIES_CODE=XMAN_Z_34	Dimensions INDICATOR=XM FREQ=A ADJUSTMENT=N  Attributes UNIT_MEASURE=_Z COMP_ORG=34	XM:A:N

3337  
3338 The above behaviour can be okay if the series XMAN\_Z\_21 contains observations for  
3339 different periods of time then the series XMAN\_Z\_34. If however both series contain  
3340 observations for the same point in time, the output for this mapping will be two  
3341 observations with the same series key, for the same period in time.

3342 **13.5 Representation maps**

3343 Representation Maps replace the SDMX 2.1 Codelist Maps and are used describe  
3344 explicit mappings between source and target Component values.

3345  
3346 The source and target of a Representation Map can reference any of the following:

- 3347 a. Codelist
- 3348 b. Free Text (restricted by type, e.g String, Integer, Boolean)
- 3349 c. Valuelist

3350  
3351 A Representation Map mapping ISO 2-character to ISO 3-character Codelists would  
3352 take the following form:

CL_ISO_ALPHA2	CL_ISO_ALPHA3
AF	AFG
AL	ALB
DZ	DZA
AS	ASM
AD	AND
etc...	

3353  
3354 A Representation Map mapping free text country names to an ISO 2-character Codelist  
3355 could be similarly described:

Text	CL_ISO_ALPHA2
"Germany"	DE
"France"	FR
"United Kingdom"	GB
"Great Britain"	GB
"Ireland"	IE
"Eire"	IE
etc...	

3356  
3357 Valuelists, introduced in SDMX 3.0, are equivalent to Codelists but allow the  
3358 maintenance of non-SDMX identifiers. Importantly, their IDs do not need to conform to  
3359 IDType, but as a consequence are not Identifiable.

3360 When used in Representation Maps, Valuelists allow Non-SDMX identifiers containing  
3361 characters like £, \$, % to be mapped to Code IDs, or Codes mapped to non-SDMX  
3362 identifiers.

3363 In common with Codelists, each item in a Valuelist has a multilingual name giving it a  
3364 human-readable label and an optional description. For example:

Value	Locale	Name
\$	en	United States Dollar
%	En	Percentage
	fr	Pourcentage

3365

3366 Other characteristics of Representation Maps:

3367 • Support the mapping of multiple source Component values to multiple Target  
3368 Component values as described in section 13.3 on n-to-n mappings; this covers  
3369 also the case of mapping an Attribute with an array representation to map  
3370 combinations of values to a single target value;

3371 • Allow source or target mappings for an Item to be optional allowing rules such  
3372 as 'A maps to nothing' or 'nothing maps to A'; and

3373 • Support for mapping rules where regular expressions or substrings are used to  
3374 match source Component values. Refer to section 13.6 for more on this topic.

### 3375 **13.6 Regular expression and substring rules**

3376 It is common for classifications to contain meanings within the identifier, for example  
3377 the code Id 'XULADS' may refer to a particular seasonality because it starts with the  
3378 letters XU.

3379 With SDMX 2.1 each code that starts with XU had to be individually mapped to the  
3380 same seasonality, and additional mappings added when new Codes were added to  
3381 the Codelists. This led to many hundreds or thousands of mappings which can be  
3382 more efficiently summarised in a single conceptual rule:

3383 *If starts with 'XU' map to 'Y'*

3384 These rules are described using either regular expressions, or substrings for simpler  
3385 use cases.

3386 **13.6.1 Regular expressions**

3387 Regular expression mapping rules are defined in the Representation Map.

3388 Below is an example set of regular expression rules for a particular component.

Regex	Description	Output
A	Rule match if input = 'A'	OUT_A
^[A-G]	Rule match if the input starts with letters A to G	OUT_B
A B	Rule match if input is either 'A' or 'B'	OUT_C

3389

3390 Like all mapping rules, the output is either a Code, a Value or free text depending on  
3391 the representation of the Component in the target Data Structure Definition.

3392 If the regular expression contains capture groups, these can be used in the definition  
3393 of the output value, by specifying \n as an output value where n is the number of the  
3394 capture group starting from 1. For example

3395

Regex	Target output	Example Input	Example Output
([0-9]{4})[0-9]([0-9]{1})	\1-Q\2	200933	2009-Q3

3396

3397 As regular expression rules can be used as a general catch-all if nothing else matches,  
3398 the ordering of the rules is important. Rules should be tested starting with the highest  
3399 priority, moving down the list until a match is found.

3400 The following example shows this:

Priority	Regex	Description	Output
1	A	Rule match if input = 'A'	OUT_A
2	B	Rule match if input = 'B'	OUT_B
3	[A-Z]	Any character A-Z	OUT_C

3401

3402 The input 'A' matches both the first and the last rule, but the first takes precedence  
3403 having the higher priority. The output is OUT\_A.

3404 The input 'G' matches on the last rule which is used as a catch-all or default in this  
3405 example.

3406 **13.6.2 Substrings**

3407 Substrings provide an alternative to regular expressions where the required section of  
3408 an input value can be described using the number of the starting character, and the  
3409 length of the substring in characters. The first character is at position 1.

3410 For instance:

Input String	Start	Length	Output
ABC_DEF_XYZ	5	3	DEF
XULADS	1	2	XU

3411

3412 Sub-strings can therefore be used for the conceptual rule *If starts with 'XU' map to Y*  
 3413 as shown in the following example:

Start	Length	Source	Target
1	2	XU	Y

3414 **13.7 Mapping non-SDMX time formats to SDMX formats**

3415 Structure mapping allows non-SDMX compliant time values in source datasets to be  
 3416 mapped to an SDMX compliant time format.

3417 Two types of time input are defined:

3418 a. **Pattern based dates** – a string which can be described using a notation like  
 3419 dd/mm/yyyy or is represented as the number of periods since a point in time, for  
 3420 example: 2010M001 (first month in 2010), or 2014D123 (123<sup>rd</sup> day in 2014); and

3421 b. **Numerical based datetime** – a number specifying the elapsed periods since a  
 3422 fixed point in time, for example Unix Time is measured by the number of  
 3423 milliseconds since 1970.

3424 The output of a time-based mapping is derived from the output Frequency, which is  
 3425 either explicitly stated in the mapping or defined as the value output by a specific  
 3426 Dimension or Attribute in the output mapping. If the output frequency is unknown or if  
 3427 the SDMX format is not desired, then additional rules can be provided to specify the  
 3428 output date format for the given frequency Id. The default rules are:

Frequency	Format	Example
A	YYYY	2010
D	YYYY-MM-DD	2010-01-01
I	YYYY-MM-DD- Thh:mm:ss	2010-01T20:22:00
M	YYYY-MM	2010-01
Q	YYYY-Qn	2010-Q1
S	YYYY-Sn	2010-S1
T	YYYY-Tn	2010-T1

W	YYYY-Wn	YYYY-W53
---	---------	----------

3429

3430 In the case where the input frequency is lower than the output frequency, the mapping  
 3431 defaults to end of period, but can be explicitly set to start, end or mid-period.

3432

3433

There are two important points to note:

3434

3435

1. The output frequency determines the output date format, but the default output can be redefined using a Frequency Format mapping to force explicit rules on how the output time period is formatted.

3436

3437

3438

2. To support the use case of changing frequency the structure map can optionally provide a start of year attribute, which defines the year start date in MM-DD format. For example: YearStart=04-01.

3439

3440

3441

### 13.7.1 Pattern based dates

3442

Date and time formats are specified by date and time pattern strings based on Java's Simple Date Format. Within date and time pattern strings, unquoted letters from 'A' to 'Z' and from 'a' to 'z' are interpreted as pattern letters representing the components of a date or time string. Text can be quoted using single quotes (') to avoid interpretation. "" represents a single quote. All other characters are not interpreted; they're simply copied into the output string during formatting or matched against the input string during parsing.

3443

3444

3445

3446

3447

3448

3449

Due to the fact that dates may differ per locale, an optional property, defining the locale of the pattern, is provided. This would assist processing of source dates, according to the given locale<sup>48</sup>. An indicative list of examples is presented in the following table:

3450

3451

English (en)	Australia (AU)	en-AU
English (en)	Canada (CA)	en-CA
English (en)	United Kingdom (GB)	en-GB
English (en)	United States (US)	en-US
Estonian (et)	Estonia (EE)	et-EE
Finnish (fi)	Finland (FI)	fi-FI
French (fr)	Belgium (BE)	fr-BE
French (fr)	Canada (CA)	fr-CA
French (fr)	France (FR)	fr-FR
French (fr)	Luxembourg (LU)	fr-LU
French (fr)	Switzerland (CH)	fr-CH
German (de)	Austria (AT)	de-AT
German (de)	Germany (DE)	de-DE

<sup>48</sup> A list of commonly used locales can be found in the Java supported locales: <https://www.oracle.com/java/technologies/javase/jdk8-jre8-suported-locales.html>

German (de)	Luxembourg (LU)	de-LU
German (de)	Switzerland (CH)	de-CH
Greek (el)	Cyprus (CY)	el-CY(*)
Greek (el)	Greece (GR)	el-GR
Hebrew (iw)	Israel (IL)	iw-IL
Hindi (hi)	India (IN)	hi-IN
Hungarian (hu)	Hungary (HU)	hu-HU
Icelandic (is)	Iceland (IS)	is-IS
Indonesian (in)	Indonesia (ID)	in-ID(*)
Irish (ga)	Ireland (IE)	ga-IE(*)
Italian (it)	Italy (IT)	it-IT

3452

### 3453 Examples

3454 22/06/1981 would be described as dd/MM/YYYY, with locale en-GB

3455 2008-mars-12 would be described as YYYY-MMM-DD, with locale fr-FR

3456 22 July 1981 would be described as dd MMM YYYY, with locale en-US

3457 22 Jul 1981 would be described as dd MMM YYYY

3458 2010 D62 would be described as YYYYDnn (day 62 of the year 2010)

3459 The following pattern letters are defined (all other characters from 'A' to 'Z' and from 'a'  
3460 to 'z' are reserved):

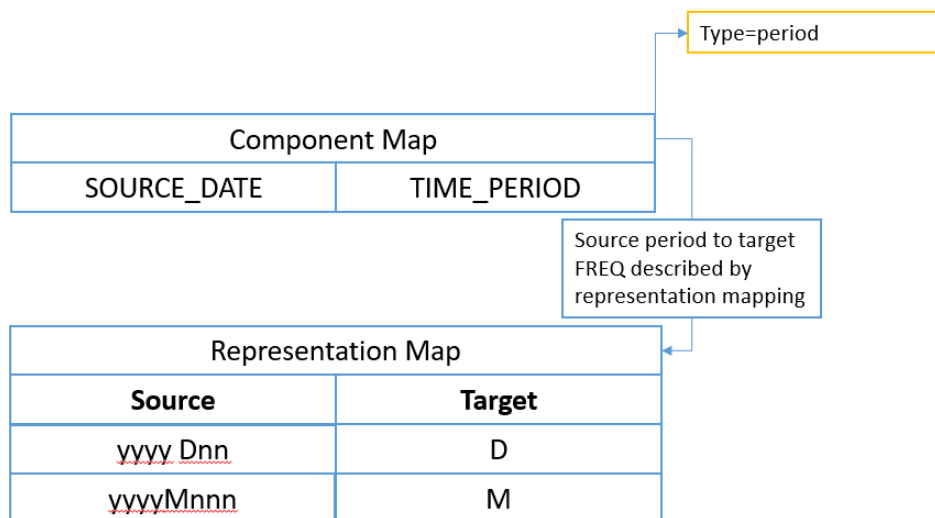
Letter	Date or Time Component	Presentation	Examples
G	Era designator	Text	AD
yy	Year short (upper case is Year of Week <sup>49</sup> )	Year	96
yyyy	Year Full (upper case is Year of Week)	Year	1996
MM	Month number in year starting with 1	Month	07
MMM	Month name short	Month	Jul
MMMM	Month name full	Month	July
ww	Week in year	Number	27
W	Week in month	Number	2
DD	Day in year	Number	189
dd	Day in month	Number	10
F	Day of week in month	Number	2
E	Day name in week	Text	Tuesday; Tue

<sup>49</sup> yyyy represents the calendar year while YYYY represents the year of the week, which is only relevant for 53 week years

U	Day number of week (1 = Monday, ..., 7 = Sunday)	Number	1
HH	Hour in day (0-23)	Number	0
kk	Hour in day (1-24)	Number	24
KK	Hour in am/pm (0-11)	Number	0
hh	Hour in am/pm (1-12)	Number	12
mm	Minute in hour	Number	30
ss	Second in minute	Number	55
S	Millisecond	Number	978
n	Number of periods, used after a SDMX Frequency Identifier such as M, Q, D (month, quarter, day)	Number	12

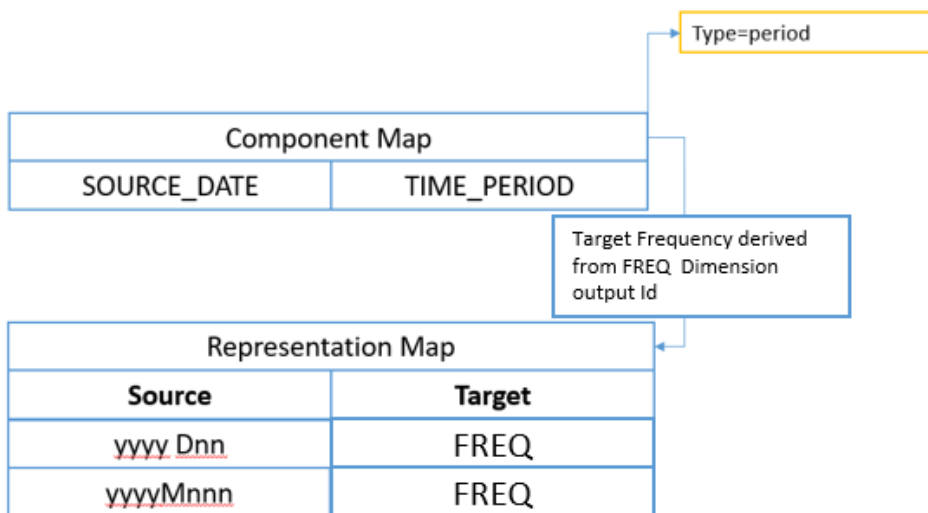
3461

3462 The model is illustrated below:



3463

3464 **Figure 24 showing the component map mapping the SOURCE\_DATE Dimension to the**  
 3465 **TIME\_PERIOD dimension with the additional information on the component map to**  
 3466 **describe the time format**



3467  
3468  
3469

Figure 25 showing an input date format, whose output frequency is derived from the output value of the FREQ Dimension

3470 **13.7.2 Numerical based datetime**

3471 Where the source datetime input is purely numerical, the mapping rules are defined by  
3472 the **Base** as a valid SDMX Time Period, and the **Period** which must take one of the  
3473 following enumerated values:

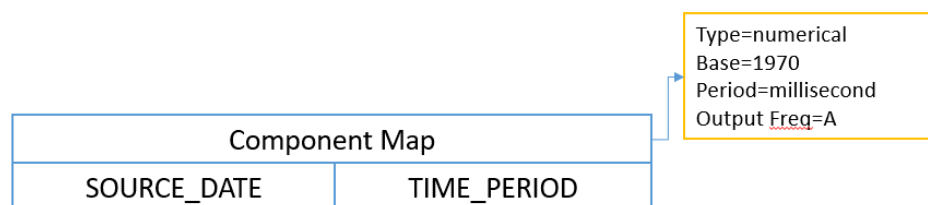
- 3474
- day
  - 3475 • second
  - 3476 • millisecond
  - 3477 • microsecond
  - 3478 • nanosecond

Numerical datetime systems	Base	Period
Epoch Time (UNIX) Milliseconds since 01 Jan 1970	1970	millisecond
Windows System Time Milliseconds since 01 Jan 1601	1601	millisecond

3479  
3480  
3481

The example above illustrates numerical based datetime mapping rules for two commonly used time standards.

3482 The model is illustrated below:



3483

3484 **Figure 26 showing the component map mapping the SOURCE\_DATE Dimension to the**  
3485 **TIME\_PERIOD Dimension with the additional information on the component map to**  
3486 **describe the numerical datetime system in use**

3487 **13.7.3 Mapping more complex time inputs**

3488 VTL should be used for more complex time inputs that cannot be interpreted using the  
 3489 pattern based on numerical methods.

3490 **13.8 Using TIME\_PERIOD in mapping rules**

3491 The source TIME\_PERIOD Dimension can be used in conjunction with other input  
 3492 Dimensions to create discrete mapping rules where the output is conditional on the  
 3493 time period value.

3494 The main use case is setting the value of Observation Attributes in the target dataset.

Rule	Source	Target
1	If INDICATOR=XULADS; and TIME_PERIOD=2007.	Set OBS_CONF=F
2	If INDICATOR=XULADS; and TIME_PERIOD=2008.	Set OBS_CONF=F
3	If INDICATOR=XULADS; and TIME_PERIOD=2009.	Set OBS_CONF=F
4	If INDICATOR=XULADS; and TIME_PERIOD=2010.	Set OBS_CONF=C

3495 In the example above, OBS\_CONF is an Observation Attribute.

3496 **13.9 Time span mapping rules using validity periods**

3497 Creating discrete mapping rules for each TIME\_PERIOD is impractical where rules  
 3498 need to cover a specific span of time regardless of frequency, and for high-frequency  
 3499 data.

3500 Instead, an optional validity period can be set for each mapping.

3501 By specifying validity periods, the example from Section 13.8 can be re-written using  
 3502 two rules as follows:

Rule	Source	Target
1	If INDICATOR=XULADS.  Validity Period start period=2007 end period=2009	Set OBS_CONF=F
2	If INDICATOR=XULADS.  Validity Period start period=2010	Set OBS_CONF=F

3503

3504 In Rule 1, start period resolves to the start of the 2007 period (2007-01-01T00:00:00),  
 3505 and the end period resolves to the very end of 2009 (2009-12-31T23:59:59). The rule

3506 will hold true regardless of the input data frequency. Any observations reporting data  
 3507 for the Indicator XULADS that fall into that time range will have an OBS\_CONF value  
 3508 of F.

3509 In Rule 2, no end period is specified so remains in effect from the start of the period  
 3510 (2010-01-01T00:00:00) until the end of time. Any observations reporting data for the  
 3511 Indicator XULADS that fall into that time range will have an OBS\_CONF value of C.

## 3512 13.10 Mapping examples

### 3513 13.10.1 Many to one mapping (N-1)

Source	Map To
<b>FREQ="A"</b> ADJUSTMENT="N" <b>REF_AREA="PL"</b> <b>COUNTERPART_AREA="W0"</b> REF_SECTOR="S1" COUNTERPART_SECTOR="S1" ACCOUNTING_ENTRY="B" STO="B5G"	FREQ="A" REF_AREA="PL" COUNTERPART_AREA="W0" INDICATOR="IND_ABC"

3514 The bold Dimensions map from source to target verbatim. The mapping simply  
 3515 specifies:

3516 FREQ => FREQ  
 3517 REF\_AREA=> REF\_AREA  
 3518 COUNTERPART\_AREA=> COUNTERPART\_AREA

3519 No Representation Mapping is required. The source value simply copies across  
 3520 unmodified.

3521 The remaining Dimensions all map to the Indicator Dimension. This is an example of  
 3522 many Dimensions mapping to one Dimension. In this case a Representation  
 3523 Mapping is required, and the mapping first describes the input 'partial key' and how  
 3524 this maps to the target indicator:

3525 N:S1:S1:B:B5G => IND\_ABC

3526 Where the key sequence is based on the order specified in the mapping (i.e  
 3527 ADJUSTMENT, REF\_SECTOR, etc will result in the first value N being taken from  
 3528 ADJUSTMENT as this was the first item in the source Dimension list.

3529 **Note:** The key order is NOT based on the Dimension order of the DSD, as the mapping  
 3530 needs to be resilient to the DSD changing.

3531

### 3532 13.10.2 Mapping other data types to Code Id

3533 In the case where the incoming data type is not a string and not a code identifier i.e.  
 3534 the source Dimension is of type Integer and the target is Codelist. This is supported by  
 3535 the RepresentationMap. The RepresentationMap source can reference a Codelist,  
 3536 Valuelist, or be free text, the free text can include regular expressions.

3537 The following representation mapping can be used to explicitly map each age to an  
 3538 output code.

Source Input Free Text	Desired Output Code Id
0	A
1	A
2	A
3	B
4	B

3545

3546

If this mapping takes advantage of regular expressions it can be expressed in two rules:

3547

Regular Expression	Desired Output
[0-2]	A
[3-4]	B

3548

3549

### 13.10.3 Observation Attributes for Time Period

3550

This use case is where a specific observation for a specific time period has an attribute value.

3551

Input INDICATOR	Input TIME_PERIOD	Output OBS_CONF
XULADS	2008	C
XULADS	2009	C
XULADS	2010	C

3552

3553

Or using a validity period on the Representation Mapping:

Input INDICATOR	Valid From/ Valid To	Output OBS_CONF
XULADS	2008/2010	C

3554

3555

### 13.10.4 Time mapping

3556

This use case is to create a time period from an input that does not respect SDMX Time Formats.

3557

3558

The Component Mapping from SYS\_TIME to TIME\_PERIOD specifies itself as a time mapping with the following details:

3559

Source Value	Source Mapping	Target Frequency	Output
18/07/1981	dd/MM/yyyy	A	1981

3560

3561

When the target frequency is based on another target Dimension value, in this example the value of the FREQ Dimension in the target DSD.

3562

Source Value	Source Mapping	Target Frequency Dimension	Output
18/07/1981	dd/MM/yyyy	FREQ	1981-07-18 (when FREQ=D)

3563

3564

When the source is a numerical format

Source Value	Start Period	Interval	Target FREQ	Output
1589808220	1970	millisecond	M	2020-05

3565

3566

3567 When the source frequency is lower than the target frequency additional information  
 3568 can be provided for resolve to start of period, end of period, or mid period, as shown  
 3569 in the following example:

Source Value	Source Mapping	Target Frequency Dimension	Output
1981	yyyy	D – End of Period	1981-12-31

3570  
 3571 When the start of year is April 1<sup>st</sup> the Structure Map has YearStart=04-01:

Source Value	Source Mapping	Target Frequency Dimension	Output
1981	yyyy	D – End of Period	1982-03-31

3572

## 3573 14 ANNEX Semantic Versioning

### 3574 14.1 Introduction to Semantic Versioning

3575 In the world of versioned data modelling exists a dreaded place called "dependency  
3576 hell." The bigger your data model through organisational, national or international  
3577 harmonisation grows and the more artefacts you integrate into your modelling, the  
3578 more likely you are to find yourself, one day, in this pit of despair.

3579  
3580 In systems with many dependencies, releasing new artefact versions can quickly  
3581 become a nightmare. If the dependency specifications are too tight, you are in danger  
3582 of version lock (the inability to upgrade an artefact without having to release new  
3583 versions of every dependent artefact). If dependencies are specified too loosely, you  
3584 will inevitably be bitten by version promiscuity (assuming compatibility with more future  
3585 versions than is reasonable). Dependency hell is where you are when version lock  
3586 and/or version promiscuity prevent you from easily and safely moving your data  
3587 modelling forward.

3588  
3589 As a very successful solution to the similar problem in software development,  
3590 "Semantic Versioning" [semver.org](http://semver.org) proposes a simple set of rules and requirements  
3591 that dictate how version numbers are assigned and incremented. These rules make  
3592 also perfect sense in the world of versioned data modelling and help to solve the  
3593 "dependency hell" encountered with previous versions of SDMX. SDMX 3.0 applies  
3594 thus the Semantic Versioning rules on all versioned SDMX artefacts. Once you release  
3595 a versioned SDMX artefact, you communicate changes to it with specific increments  
3596 to your version number.

3597  
3598 **This SDMX 3.0(.0) specification inherits the original [semver.org](http://semver.org) 2.0.0 wording**  
3599 **(license: [Creative Commons - CC BY 3.0](https://creativecommons.org/licenses/by/3.0/)) and applies it to versioned SDMX**  
3600 **structural artefacts.** Under this scheme, version numbers and the way they change  
3601 convey meaning about the underlying data structures and what has been modified from  
3602 one version to the next.  
3603

### 3604 14.2 Semantic Versioning Specification for SDMX 3.0(.0)

3605 The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",  
3606 "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this  
3607 document are to be interpreted as described in RFC 2119.  
3608

3609 In the following, "versioned" artefacts are understood to be semantically versioned  
3610 SDMX structural artefacts, and X, Y, Z and EXT are understood as placeholders for  
3611 the version parts MAJOR, MINOR, PATCH, and EXTENSION, as defined in chapter 4.3.

3612 The following rules apply to versioned artefacts:

- 3613
- All versioned SDMX artefacts MUST specify a version number.
  - The version number of immutable versioned SDMX artefacts MUST take the form X.Y.Z where X, Y, and Z are non-negative integers and MUST NOT contain leading zeroes. X is the MAJOR version, Y is the MINOR version, and Z is the PATCH version. Each element MUST increase numerically. For instance: 1.9.0 -> 1.10.0 -> 1.11.0.
- 3614  
3615  
3616  
3617  
3618

- 3619       • Once an SDMX artefact with an X.Y.Z version has been shared externally or  
3620 publicly released, the contents of that version **MUST NOT** be modified. That  
3621 artefact version is considered stable. Any modifications **MUST** be released as  
3622 a new version.
- 3623       • **MAJOR** version zero (0.y.z) is for initial modelling. Anything **MAY** change at any  
3624 time. The externally released or public artefact **SHOULD NOT** be considered  
3625 stable.
- 3626       • Version 1.0.0 defines the first stable artefact. The way in which the version  
3627 number is incremented after this release is dependent on how this externally  
3628 released or public artefact changes.
- 3629       • **PATCH** version Z (x.y.Z | x > 0) **MUST** be incremented if only backwards  
3630 compatible property changes are introduced. A property change is defined as  
3631 an internal change that does not affect the relationship to other artefacts. These  
3632 are changes in the artefact's or artefact element's names, descriptions and  
3633 annotations that **MUST NOT** alter their meaning.
- 3634       • **MINOR** version Y (x.Y.z | x > 0) **MUST** be incremented if a new, backwards  
3635 compatible element is introduced to a stable artefact. These are additional  
3636 items in ItemScheme artefacts. It **MAY** be incremented if substantial new  
3637 information is introduced within the artefact's properties. It **MAY** include **PATCH**  
3638 level changes. **PATCH** version **MUST** be reset to 0 when **MINOR** version is  
3639 incremented.
- 3640       • **MAJOR** version X (X.y.z | X > 0) **MUST** be incremented if any backwards  
3641 incompatible changes are introduced to a stable artefact. These often relate to  
3642 deletions of items in ItemSchemes or to backwards incompatibility introduced  
3643 due to changes in references to other artefacts. A **MAJOR** version change **MAY**  
3644 also include **MINOR** and **PATCH** level changes. **PATCH** and **MINOR** version  
3645 **MUST** be reset to 0 when **MAJOR** version is incremented.
- 3646       • A mutable version, e.g. used for externally released or public drafts or as pre-  
3647 release, **MUST** be denoted by appending an **EXTENSION** that consists of a  
3648 hyphen and a series of dot separated identifiers immediately following the  
3649 **PATCH** version (x.y.z-EXT). Identifiers **MUST** comprise only ASCII  
3650 alphanumerics and hyphen [0-9A-Za-z-]. Identifiers **MUST NOT** be empty.  
3651 Numeric identifiers **MUST NOT** include leading zeroes. However, to foster  
3652 harmonisation and general comprehension it is generally recommended to use  
3653 the standard **EXTENSION** "**-draft**". Extended versions have a lower precedence  
3654 than the associated stable version. An extended version indicates that the  
3655 version is unstable and it might not satisfy the intended compatibility  
3656 requirements as denoted by its associated stable version. When making  
3657 changes to an SDMX artefact with an extended version number then one is not  
3658 required to increment the version if those changes are kept within the allowed  
3659 scope of the version increment from the previous version (if that existed),  
3660 otherwise also here the before mentioned version increment rules for X.Y.Z  
3661 apply. Concretely, a version X.0.0-EXT will allow for any changes, a version  
3662 X.Y.0-EXT will allow only for **MINOR** changes and a version X.Y.Z-EXT will  
3663 allow only for any **PATCH** changes, as defined above. **EXTENSION** examples:  
3664 1.0.0-draft, 1.0.0-draft.1, 1.0.0-0.3.7, 1.0.0-x.7.z.92.

- 3665
- 3666
- 3667
- 3668
- 3669
- 3670
- 3671
- 3672
- 3673
- 3674
- 3675
- 3676
- 3677
- 3678
- 3679
- 3680
- 3681
- Precedence refers to how versions are compared to each other when ordered. Precedence MUST be calculated by separating the version into MAJOR, MINOR, PATCH and EXTENSION identifiers in that order. Precedence is determined by the first difference when comparing each of these identifiers from left to right as follows: MAJOR, MINOR, and PATCH versions are always compared numerically. Example: 1.0.0 < 2.0.0 < 2.1.0 < 2.1.1. When MAJOR, MINOR, and PATCH are equal, an extended version has lower precedence than a stable version. Example: 1.0.0-draft < 1.0.0. Precedence for two extended versions with the same MAJOR, MINOR, and PATCH version MUST be determined by comparing each dot separated identifier from left to right until a difference is found as follows: identifiers consisting of only digits are compared numerically and identifiers with letters or hyphens are compared lexically in ASCII sort order. Numeric identifiers always have lower precedence than non-numeric identifiers. A larger set of EXTENSION fields has a higher precedence than a smaller set, if all of the preceding identifiers are equal. Example: 1.0.0-draft < 1.0.0-draft.1 < 1.0.0-draft.prerelease < 1.0.0-prerelease < 1.0.0-prerelease.2 < 1.0.0-prerelease.11 < 1.0.0-rc.1 < 1.0.0.
- 3682
- The reasons for version changes MAY be documented in brief form in an artefact's annotation of type "CHANGELOG".
- 3683
- 3684

3685 **14.3 Backus–Naur Form Grammar for Valid SDMX 3.0(0)**

3686 **Semantic Versions**

```

3687 <valid semver> ::= <version core>
3688                 | <version core> "-" <extension>
3689
3690 <version core> ::= <major> "." <minor> "." <patch>
3691
3692 <major> ::= <numeric identifier>
3693
3694 <minor> ::= <numeric identifier>
3695
3696 <patch> ::= <numeric identifier>
3697
3698 <extension> ::= <dot-separated extension identifiers>
3699
3700 <dot-separated extension identifiers> ::= <extension identifier>
3701                                         | <extension identifier> "." <dot-
3702 separated extension identifiers>
3703
3704 <extension identifier> ::= <alphanumeric identifier>
3705                             | <numeric identifier>
3706
3707 <alphanumeric identifier> ::= <non-digit>
3708                             | <non-digit> <identifier characters>
3709                             | <identifier characters> <non-digit>
3710                             | <identifier characters> <non-digit> <identifier
3711 characters>
3712
3713 <numeric identifier> ::= "0"
3714                             | <positive digit>
3715                             | <positive digit> <digits>
3716
3717 <identifier characters> ::= <identifier character>
3718                             | <identifier character> <identifier characters>
3719

```

```

3720 <identifier character> ::= <digit>
3721 | <non-digit>
3722
3723 <non-digit> ::= <letter>
3724 | "-"
3725
3726 <digits> ::= <digit>
3727 | <digit> <digits>
3728
3729 <digit> ::= "0"
3730 | <positive digit>
3731
3732 <positive digit> ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
3733
3734 <letter> ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J"
3735 | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T"
3736 | "U" | "V" | "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "d"
3737 | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n"
3738 | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x"
3739 | "y" | "z"
3740

```

#### 3741 **14.4 Dependency Management in SDMX 3.0(.0):**

3742 MAJOR, MINOR or PATCH version parts in SDMX 3.0 artefact references CAN be  
3743 wildcarded using "+" as extension:

- 3744 • X+.Y.Z means the currently latest available version  $\geq$  X.Y.Z
  - 3745 ○ Example: "2+.3.1" means the currently latest available version  $\geq$   
3746 "2.3.1" (even if not backwards compatible)
  - 3747 ○ Typical use case: references in SDMX Categorisations
- 3748 • X.Y+.Z means the currently latest available backwards compatible version  $\geq$   
3749 X.Y.Z
  - 3750 ○ Example: "2.3+.1" means the currently latest available version  $\geq$   
3751 "2.3.1" and  $<$  "3.0.0" (all backwards compatible versions  $\geq$   
3752 "2.3.1")
  - 3753 ○ Typical use case: references in SDMX DSD
- 3754 • X.Y.Z+ means the currently latest available forwards and backwards  
3755 compatible version  $\geq$  X.Y.Z
  - 3756 ○ Example: "2.3.1+" means the currently latest available version  $\geq$   
3757 "2.3.1" and  $<$  "2.4.0" (all forwards and backwards compatible  
3758 versions  $\geq$  "2.3.1")
- 3759 • Non-versioned and 2-digit version SDMX structural artefacts CAN reference  
3760 any other non-versioned or versioned (whether SemVer or not) SDMX  
3761 structural artefacts.
- 3762
- 3763 • Semantically versioned artefacts MUST only reference other semantically  
3764 versioned artefacts.
- 3765 • Wildcarded references in a stable artefact implicitly target only future stable  
3766 versions of the referenced artefacts within the defined wildcard scope.

3767                   ○ Example: The reference to "AGENCY\_ID:CODELIST\_ID(2.3+.1) "  
3768                   in an artefact "AGENCY\_ID:DSD\_ID(2.2.1) " resolves to artefact  
3769                   "AGENCY\_ID:CODELIST\_ID(2.4.3) " if that was currently the latest  
3770                   available stable version.

3771                   • Wildcarded references in a version-extended artefact implicitly target future  
3772                   stable and version-extended versions of the referenced artefacts within the  
3773                   defined wildcard scope.

3774                   ○ Example: The reference to "AGENCY\_ID:CODELIST\_ID(2.3+.1) "  
3775                   in an artefact "AGENCY\_ID:DSD\_ID(2.2.1-draft) " resolves to  
3776                   artefact "AGENCY\_ID:CODELIST\_ID(2.5.0-draft) " if that was  
3777                   currently the latest available version.

3778                   • References to specific version-extended artefacts MAY be used, but those  
3779                   cannot be combined with a wildcard.

3780                   ○ Example: The reference to "AGENCY\_ID:CODELIST\_ID(2.5.0-  
3781                   draft) " in an artefact "AGENCY\_ID:DSD\_ID(2.2.1) " resolves to  
3782                   artefact "AGENCY\_ID:CODELIST\_ID(2.5.0-draft) ", which might  
3783                   be subject to continued backwards compatible changes.

3784                   Because both, wildcarded references and references to version-extended artefacts,  
3785                   allow for changes in the referenced artefacts, care needs to be taken when choosing  
3786                   the appropriate references in order to achieve the required limitation in the allowed  
3787                   scope of changes.

3788  
3789                   For references to non-dependent artefacts, MAJOR, MINOR or PATCH version parts in  
3790                   SDMX 3.0 artefact references CAN alternatively be wildcarded using "\*" as  
3791                   replacement:

3792                   \* means all available versions

3793                   ***14.5 Upgrade and conversions of artefacts defined with***  
3794                   ***previous SDMX standard versions to Semantic Versioning***

3795                   Because SDMX standardises the interactions between statistical systems, which  
3796                   cannot all be upgraded at the same time, the new versioning rules cannot be applied  
3797                   to existing artefacts in EDIFACT, SDMX 1.0, 2.0 or 2.1. SemVer can only be applied  
3798                   to structural artefacts that are newly modelled with the SDMX 3.0 Information Model.  
3799                   Migrating to SemVer means migrating to the SDMX 3.0 Information Model, to its new  
3800                   API version and new versions of its exchange message formats.

3801  
3802                   To migrate SDMX structural artefacts created previously to SDMX 3.0.0:

3803  
3804                   If the artefacts do not need versioning, then the new artefacts based on the SDMX 3.0  
3805                   Information Model SHOULD remain as-is, e.g., a previous artefact with the non-final  
3806                   version 1.0 and that doesn't need versioning becomes non-versioned, i.e., keeps  
3807                   version 1.0. This will be the case for all AgencyScheme artefacts.

3808  
3809                   If artefact versioning is required and SDMX 3.0.0 Semantic Versioning is available  
3810                   within the tools and processes used, then it is recommended to switch to Semantic  
3811                   Versioning with the following steps:

3812 1. Complement the missing version parts with 0s to make the version number  
3813 SemVer-compliant using the MAJOR.MINOR.PATCH-EXTENSION syntax:

3814 Example: Version 2 becomes version 2.0.0 and version 3.1 becomes  
3815 version 3.1.0.

3816 2. Replace the "isFinal=false" property by the version extensions "-draft" (or  
3817 alternatively "-unstable" or "-nonfinal" depending on the use case).

3818 Example: Version 1.3 with isFinal=true becomes version 1.3.0 and  
3819 version 1.3 with isFinal=false becomes version 1.3.0-draft.

3820 If artefact versioning is required but semantic versioning cannot be applied, then  
3821 version strings used in previous versions of the Standard (e.g., version=1.2) may  
3822 continue to be used.

3823 Note: Like for other not fully backwards compatible SDMX 3.0 features, also some  
3824 cases of semantically versioned SDMX 3.0 artefacts cannot be converted back to  
3825 earlier SDMX versions. This is the case when one or more extensions have been  
3826 created in parallel to the corresponding stable version. In this case, only the stable  
3827 version SHOULD be converted to a final version (e.g., 3.2.1 becomes 3.2.1 final, and  
3828 3.2.1-draft cannot be converted back).

## 3831 **14.6 FAQ for Semantic Versioning**

3832 **My organisation is new to SDMX and starts to implement 3.0 or starts to**  
3833 **implement a new process fully based on SDMX 3.0. Which versioning scheme**  
3834 **should be used?**

3836 If all counterparts involved in the process and all tools used for its implementation are  
3837 SDMX 3.0-ready, then it is recommended to:

- 3838 • in general, use semantic versioning;
- 3839 • exceptionally, do not use versioning for artefacts that do not require it, e.g.  
3840 artefacts that never change, that are only used internally or for which  
3841 communication on changes with external parties or systems is not required.

3842 **How should I deal with revisions in the 0.y.z initial modelling phase?**

3845 The simplest thing to do is start your initial modelling release at 0.1.0 and then  
3846 increment the minor version for each subsequent release.

3848 **How do I know when to release 1.0.0?**

3850 If your data model is being used in production, it should probably already be 1.0.0. If  
3851 you have a stable artefact on which users have come to depend, you should be 1.0.0.  
3852 If you're worrying a lot about backwards compatibility, you should probably already be  
3853 1.0.0.

3855 **Doesn't this discourage rapid modelling and fast iteration?**

3856

3857 Major version zero is all about rapid modelling. If you're changing the artefact every  
3858 day you should either still be in version 0.y.z or on the next (minor or) major version  
3859 for a separate modelling.

3860

3861 **If even the tiniest backwards incompatible changes to the public artefact require**  
3862 **a major version bump, won't I end up at version 42.0.0 very rapidly?**

3863

3864 This is a question of responsible modelling and foresight. Incompatible changes should  
3865 not be introduced lightly to a data model that has a lot of dependencies. The cost that  
3866 must be incurred to upgrade can be significant. Having to bump major versions to  
3867 release incompatible changes means you will think through the impact of your  
3868 changes, and evaluate the cost/benefit ratio involved.

3869

3870 **Documenting the version changes in an artefact's annotation of type**  
3871 **"CHANGELOG" is too much work!**

3872

3873 It is your responsibility as a professional modeller to properly document the artefacts  
3874 that are intended for use by others. Managing data model complexity is a hugely  
3875 important part of keeping a project efficient, and that's hard to do if nobody knows how  
3876 to use your data model, or what artefacts are safe to reuse. In the long run, SDMX 3.0  
3877 Semantic Versioning can keep everyone and everything running smoothly.

3878 However, refrain from overdoing. Nobody can and will read too long lists of changes.

3879 Thus, keep it to the absolute essence, and mainly use it for short announcements. You

3880 can even skip the changelog if the change is impact-less. For all complete reports, a

3881 new API feature could be more useful to automatically generate a log of differences

3882 between two versions.

3883

3884 **What do I do if I accidentally release a backwards incompatible change as a**  
3885 **minor version?**

3886

3887 As soon as you realise that you've broken the SDMX 3.0 Semantic Versioning  
3888 specification, fix the problem and release a new minor version that corrects the  
3889 problem and restores backwards compatibility. Even under this circumstance, it is  
3890 unacceptable to modify versioned releases. If it's appropriate, document the offending  
3891 version and inform your users of the problem so that they are aware of the offending  
3892 version.

3893

3894 **What if I inadvertently alter the public artefact in a way that is not compliant with**  
3895 **the version number change (i.e. the modification incorrectly introduces a major**  
3896 **breaking change in a patch release)?**

3897

3898 Use your best judgement. If you have a huge audience that will be drastically impacted  
3899 by changing the behaviour back to what the public artefact intended, then it may be  
3900 best to perform a major version release, even though the property change could strictly  
3901 be considered a patch release. Remember, SDMX 3.0.0 Semantic Versioning is all  
3902 about conveying meaning by how the version number changes. If these changes are  
3903 important to your users, use the version number to inform them.

3904

3905 **How should I handle deprecating elements?**

3906

3907 Deprecating existing elements is a normal part of data modelling and is often required  
3908 to make forward progress or follow history (changing classifications, evolving reference

3909 areas). When you deprecate part of your stable artefact, you should issue a new minor  
 3910 version with the deprecation in place (e.g. add the new country code but still keep the  
 3911 old country code) and with a "CHANGELOG" annotation announcing the deprecation  
 3912 (e.g. the intention to remove the old country code in a future version) . Before you  
 3913 completely remove the functionality in a new major release there should be at least  
 3914 one minor release that contains the deprecation so that users can smoothly transition  
 3915 to the new artefact.

3916  
 3917 **Does SDMX 3.0.0 Semantic Versioning have a size limit on the version string?**  
 3918

3919 No, but use good judgement. A 255 character version string is probably overkill, for  
 3920 example. In addition, specific SDMX implementations may impose their own limits on  
 3921 the size of the string. Remember, it is generally recommended to use the standard  
 3922 extension "-draft".  
 3923

3924 **Is "v1.2.3" a semantic version?**  
 3925

3926 No, "v1.2.3" is not a semantic version. The semantic version is "1.2.3".  
 3927

3928 **Is there a suggested regular expression (RegEx) to check an SDMX 3.0.0**  
 3929 **Semantic Versioning string?**  
 3930

3931 There are two:  
 3932

3933 One with named groups for those systems that support them (PCRE [Perl Compatible  
 3934 Regular Expressions, i.e. Perl, PHP and R], Python and Go).  
 3935

3936 Reduced version (without original SemVer "build metadata") from:  
 3937 <https://regex101.com/r/Ly7O1x/3/>  
 3938

3939 `^(?P<major>0|[1-9]\d*)\.(?P<minor>0|[1-9]\d*)\.(?P<patch>0|[1-9]\d*)(?:-(?P<extension>(?:0|[1-9]\d*|\d*[a-zA-Z-][0-9a-zA-Z-]*))?(?:\.(?:0|[1-9]\d*|\d*[a-zA-Z-][0-9a-zA-Z-]*)*)?)?$`  
 3940  
 3941

3942 And one with numbered capture groups instead (so cg1 = major, cg2 = minor, cg3 =  
 3943 patch and cg4 = extension) that is compatible with ECMA Script (JavaScript), PCRE  
 3944 (Perl Compatible Regular Expressions, i.e. Perl, PHP and R), Python and Go.  
 3945  
 3946

3947 Reduced version (without original SemVer "build metadata") from:  
 3948 <https://regex101.com/r/vkijKf/1/>  
 3949

3950 `^(0|[1-9]\d*)\.(0|[1-9]\d*)\.(0|[1-9]\d*)(?:-((?:0|[1-9]\d*|\d*[a-zA-Z-][0-9a-zA-Z-]*)|(?:\.(?:0|[1-9]\d*|\d*[a-zA-Z-][0-9a-zA-Z-]*)*)?)?)?$`  
 3951  
 3952  
 3953

3954 **Must I adopt semantic versioning rules when switching to SDMX 3.0?**  
 3955

3956 No. If backwards compatibility with pre-existing tools and processes is required, then  
 3957 it is possible to continue using the previous versioning scheme (with up to two version  
 3958 parts MAJOR.MINOR). Semantic versioning is indicated only for those use cases  
 3959 where a proper artefact versioning is required. If versioning does not apply to some or  
 3960 all of your artefacts, then rather migrate to non-versioned SDMX 3.0 artefacts.

3961

3962

**May I mix artefacts that follow semantic versioning with artefacts that don't?**

3963

3964

Artefacts that are not (semantically) versioned may reference artefacts that are semantically versioned, but those are fully safe to use only when not extended. However, the reverse is not true: non-semantically-versioned artefacts do not offer change guarantees, and, therefore, should not be referenced by semantically versioned artefacts.

3965

3966

3967

3968

3969

3970

**I have plenty of artefacts. I'm happy with my current versioning policy and I don't want to use SemVer! Can I still migrate to SDMX 3.0, and if so, what do I need to do?**

3971

3972

3973

3974

Yes, of course, you can. The introduction of semantic versioning is done in a way which is largely backward compatible with previous versions of the standard, so you can keep your existing 2-digit version numbers (1.0, 1.1, 2.0, etc.) if that is required by your current tools and processes. However, if not using SemVer then pre-SDMX 3.0 final artefacts will be migrated as non-final and mutable in SDMX 3.0. There are also many good reasons to move to SemVer, and the migration is encouraged. Be assured that there will be tools out there that will assist you doing this in an efficient and convenient way.

3975

3976

3977

3978

3979

3980

3981

3982

3983

**I have plenty of artefacts versioned 'X.Y'. I want to make some of them immutable, and enjoy the benefits provided by semantic versioning. Some other artefacts however must remain mutable (i.e. non final). However, in both cases, I'd like adopt the semantic versioning. What do I need to do?**

3984

3985

3986

3987

3988

For artefacts that will be made immutable and are therefore safe to use, simply append a '.0' to the current version (use X.Y.0) when migrating to Semantic Versioning. E.g., if the version of your artefact is currently 1.10, then migrate to 1.10.0.

3989

3990

3991

3992

For artefacts that remain mutable, and therefore do not bring the guarantees of semantic versioning, if you want to benefit from the advantages of semantic versioning, then simply append '.0-notfinal' to the version string. So, if the version of your artefact is currently 1.10, use 1.10.0-notfinal instead. Indeed, other extensions can be used depending on your use case.

3993

3994

3995

3996

3997

3998

**I have adopted SDMX 3.0 with the semantic versioning conventions for the version strings of all my artefacts, regardless of whether these are stable (e.g. 1.0.0) or unstable (e.g. 1.0.0-notfinal, 1.0.0-draft, etc.). However, I still receive artefacts from organizations that have not yet adopted SemVer conventions for the version strings. How should I treat these?**

3999

4000

4001

4002

4003

4004

The only artefacts that are safe to use, are those that are semantically versioned. Starting with SDMX 3.0, these artefacts MUST use the SEMVER version string to indicate this fact and the version string of these artefacts MUST be expressed as X.Y.Z (e.g. 2.1.0). Extended versions bring some limited guarantees for changes.

4005

4006

4007

4008

4009

All other artefacts are in principle unsafe. They might be safe in practice but the SDMX standard does not bring any guarantees in that respect, and these artefacts may change in unpredictable ways.

4010

4011

4012

4013 In practice, the migration approach will often mirror the way in which organisations  
4014 have migrated between earlier SDMX versions. Rarely, the new data models used  
4015 mixed SDMX standard versions in their dependencies, and if they did then standard  
4016 conversions were put in place. A typical method is to first migrate the re-used artefacts  
4017 from the previous SDMX version to SDMX 3.0 and while doing so to apply the  
4018 appropriate new semantic version string. From that point onwards, you can enjoy the  
4019 advantages of the new SDMX versioning features for all those artefacts that require  
4020 appropriate versioning.