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

1 Purpose and Structure

1.1 Purpose

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

1.2 Structure

This document is organized into the following major parts:

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

2 General Notes on This Document

As of version SDMX 2.1, the term "Key family" has been replaced by Data Structure Definition (also known and referred to as DSD) both in the XML schemas and the Information Model.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

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

3 Guide for SDMX Format Standards

3.1 Introduction

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

3.2 SDMX Information Model for Format Implementers

3.2.1 Introduction

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

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

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

3.3 SDMX Formats: Expressive Capabilities and Function

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

3.3.1 Format Optimizations and Differences

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

Version 2.0 was characterised by 4 data messages, each with a distinct format: Generic, Compact, Cross-Sectional and Utility. Because of the design, data in some formats could not always be related to another format. In version 2.1, this issue has been addressed by merging some formats and eliminating others. As a result, in SDMX 2.1 there were just two types of data formats: *GenericData* and *StructureSpecificData* (i.e., specific to one Data Structure Definition). As of SDMX 3.0, based also on the real-life usage of 2.1 XML formats but also the new formats introduced (JSON and CSV), only one XML format remains, i.e., *StructureSpecificData*. Furthermore, the time specific sub-formats have also been deprecated due to the lack of usage.

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

Structure Definition

- The SDMX-ML Structure Message is currently the main way of modelling a DSD. The SDMX-JSON version follows the same principles, while the SDMX-CSV does not support structures, yet.
- The SDMX-ML Structure Message allows for the structures on which a Data Structure Definition depends – that is, codelists and concepts – to be either included in the message or to be referenced by the message containing the data structure definition. XML syntax is designed to leverage URIs and other Internet-based referencing mechanisms, and these are used in the SDMX-ML message. This option is also available in SDMX-JSON. The latter, though, further supports conveying data with some structural metadata within a single message.
- All structures can be inserted, replaced or deleted, unless structural dependencies are not respected.

Validation

- The SDMX-ML structure specific messages will allow validation of XML syntax and data typing to be performed with a generic XML parser and enforce agreement between the structural definition and the data to a moderate degree with the same tool.
- Similarly, the SDMX-JSON message can be validated using JSON Schema and hence may also be generically parsed and validated.
- The SDMX-CSV format cannot be validated by generic tools.

Update and Delete Messages

- All data messages allow for both append/replace/delete messages.
- These messages allow also transmitting only data or only documentation (i.e., Attribute values without Observation values).

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

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

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

3.4 SDMX Best Practices

3.4.1 Reporting and Dissemination Guidelines

3.4.1.1 Central Institutions and Their Role in Statistical Data Exchanges

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

Central institutions can play a double role:

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

3.4.1.2 Defining Data Structure Definitions (DSDs)

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

Dimensions, Attributes and Codelists

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

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

- **Avoid composite dimensions**. Each dimension should correspond to a single characteristic of the data, not to a combination of characteristics.
- Consider the inclusion of the following attributes. Once the key structure of a data structure definition has been decided, then the set of (preferably mandatory) attributes of this data structure definition has to be defined. In general, some statistical concepts are deemed necessary across all Data Structure Definitions to qualify the contained information. Examples of these are:
    - A descriptive title for the series (this is most useful for dissemination of data for viewing e.g., on the web).
    - Collection (e.g., end of period, averaged or summed over period).
    - Unit (e.g., currency of denomination).
    - Unit multiplier (e.g., expressed in millions).
    - Availability (which institutions can a series become available to).
    - Decimals (i.e., number of decimal digits used in numerical observations).
    - Observation Status (e.g., estimate, provisional, normal).

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

- **Avoid creating a new code list where one already exists**. It is highly recommended that structural definitions and code lists be consistent with internationally agreed standard methodologies, wherever they exist, e.g., System of National Accounts 1993; Balance of Payments Manual, Fifth Edition; Monetary and Financial Statistics Manual; Government Finance Statistics Manual, etc. When setting-up a new data exchange, the following order of priority is suggested when considering the use of code lists:
    - international standard code lists;
    - international code lists supplemented by other international and/or regional institutions;
    - standardised lists used already by international institutions;
    - new code lists agreed between two international or regional institutions;
    - new code lists which extend existing code lists, by adding only missing codes;
    - new specific code lists.

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

Data Structure Definition Structure

- The following items have to be specified by a structural definitions maintenance agency when defining a new data structure definition:
- Data structure definition (DSD) identification:
    - DSD identifier
    - DSD name
- A list of metadata concepts assigned as dimensions of the data structure definition. For each:
    - (statistical) concept identifier
    - code list identifier (id, version, maintenance agency) if the representation is coded
- A list of (statistical) concepts assigned as attributes for the data structure definition. For each:
    - (statistical) concept identifier
    - code list identifier if the concept is coded
    - usage: mandatory, optional
    - relationship to dimensions and measures
    - maximum text length for the uncoded concepts
    - maximum code length for the coded concepts
- A list of the code lists used in the data structure definition. For each:
    - code list identifier
    - code list name
    - code values and descriptions
- Definition of Dataflow. Two (or more) partners performing data exchanges in a certain context need to agree on:
    - the list of dataset identifiers they will be using;
    - for each Dataflow:
        - its content (e.g., by Constraints) and description
        - the relevant DSD that defines the structure of the data reported or disseminated according the Dataflow

3.4.1.3 Exchanging Attributes

3.4.1.3.1 *Attributes on series and group levels*

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

expected that partner institutions will be in a position to provide this information in the available formats over time.

- A centre may agree with its data exchange partners special procedures for authorising the setting of attributes' initial values.
- Communication of changes to the centre.
- Following the creation of a series, the attribute values do not have to be reported again by senders, as long as they do not change.
- Whenever changes in attribute values for a series (or group) occur, the reporting institutions should report either all attribute values again (this is the recommended option) or only the attribute values which have changed. This applies both to the mandatory and the conditional attributes. For example, if a previously reported value for a conditional attribute is no longer valid, this has to be reported to the centre.
- A centre may agree with its data exchange partners special procedures for authorising modifications in the attribute values.
- Communication of observation level attributes "observation status", "observation confidentiality", "observation pre-break" is recommended.
- Whenever an observation is exchanged, the corresponding observation status is recommended to also be exchanged attached to the observation, regardless of whether it has changed or not since the previous data exchange.
- If the "observation status" changes and the observation remains unchanged, both components would have to be reported (unless the observation is deleted).

    For Data Structure Definitions having also the observation level attributes "observation confidentiality" and "observation pre-break" defined, this rule applies to these attributes as well: if an institution receives from another institution an observation with an observation status attribute only attached, this means that the associated observation confidentiality and pre-break observation attributes either never existed or from now they do not have a value for this observation.

3.4.2 Best Practices for Batch Data Exchange

3.4.2.1 Introduction

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

3.4.2.2 Positioning of the Dimension "Frequency"

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

3.4.2.3 Identification of Data Structure Definitions (DSDs)

In order to facilitate the easy and immediate recognition of the structural definition maintenance agency that defined a data structure definition, some central institutions devising structural definitions use the first characters of the data structure definition identifiers to identify their institution: e.g., BIS_EER, EUROSTAT_BOP_01, ECB_BOP1, etc. Nevertheless, using the AgencyId may disambiguate any Artefact in a more efficient and machine readable way.

3.4.2.4 Identification of the Dataflows

In order to facilitate the easy and immediate recognition of the institution administrating a Dataflow, some central institutions prefer to use the first characters of the Dataflow identifiers to identify their institution: e.g. BIS_EER, ECB_BOP1, ECB_BOP1, etc. Nevertheless, using the AgencyId may disambiguate any Artefact in a more efficient and machine readable way.

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

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

3.4.2.5 Special Issues

3.4.2.5.1 *"Frequency" related issues*

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

4 General Notes for Implementers

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

4.1 Representations

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

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

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`

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

- `AlphaNumeric` (`common:AlphaNumericType`, string which only allows A-z and 0-9)
- `Alpha` (`common:AlphaType`, string which only allows A-z)
- `Numeric` (`common:NumericType`, string which only allows 0-9, but is not numeric so that is can having leading zeros)
- `Count` (`xs:integer`, a sequence with an interval of "1")
- `InclusiveValueRange` (`xs:decimal` with the `minValue` and `maxValue` facets supplying the bounds)
- `ExclusiveValueRange` (`xs:decimal` with the `minValue` and `maxValue` facets supplying the bounds)
- `Incremental` (`xs:decimal` with a specified `interval`; the interval is typically enforced outside of the XML validation)
- `TimeRange` (`common:TimeRangeType`, `startDateTime` + `Duration`)
- `ObservationalTimePeriod` (`common:ObservationalTimePeriodType`, a union of `StandardTimePeriod` and `TimeRange`).
- `StandardTimePeriod` (`common:StandardTimePeriodType`, a union of `BasicTimePeriod` and `ReportingTimePeriod`).
- `BasicTimePeriod` (`common:BasicTimePeriodType`, a union of `GregorianTimePeriod` and `DateTime`)
- `GregorianTimePeriod` (`common:GregorianTimePeriodType`, a union of `GregorianYear`, `GregorianMonth`, and `GregorianDay`)
- `ReportingTimePeriod` (`common:ReportingTimePeriodType`, a union of `ReportingYear`, `ReportingSemester`, `ReportingTrimester`, `ReportingQuarter`, `ReportingMonth`, `ReportingWeek`, and `ReportingDay`).
- `ReportingYear` (`common:ReportingYearType`)
- `ReportingSemester` (`common:ReportingSemesterType`)
- `ReportingTrimester` (`common:ReportingTrimesterType`)
- `ReportingQuarter` (`common:ReportingQuarterType`)
- `ReportingMonth` (`common:ReportingMonthType`)
- `ReportingWeek` (`common:ReportingWeekType`)
- `ReportingDay` (`common:ReportingDayType`)
- `XHTML` (`common:StructuredText`, allows for multi-lingual text content that has `XHTML` markup)
- `KeyValues` (`common:DataKeyType`)
- `IdentifiableReference` (types for each `IdentifiableObject`)
- `GeospatialInformation` (a geo feature set, according to the pattern in section 7.2)

Data types also have a set of facets:

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

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

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

4.1.1 Data Types

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

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

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

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

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

The above looks like this, in JSON schema:

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

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

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

SDMX Facet	**XML Schema**	**JSON schema "pattern"[1] 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))?$"`
`GregorianMonth`	`xsd:gYearMonth`	`"^-?([1-9][0-9]{3,}|0[0-9]{3})-(0[1-9]|1[0-2])(Z|(\+|-)((0[0-9]|1[0-3]):[0-5][0-9]|14:00))?$"`

[1] 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)?)?$"

4.2 Time and Time Format

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

4.2.1 Introduction

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

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

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

- **Observational Time Period**
    - **Standard Time Period**
        - **Basic Time Period**
            - **Gregorian Time Period**
            - *Date Time*
        - **Reporting Time Period**
    - *Time Range*

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

4.2.2 Observational Time Period

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

4.2.3 Standard Time Period

This is the superset of any predefined time period or a distinct point in time. A time period consists of a distinct start and end point. If the start and end of a period are expressed as date instead of a complete date time, then it is implied that the start of the period is the beginning of the start day (i.e. 00:00:00) and the end of the period is the end of the end day (i.e. 23:59:59).

4.2.4 Gregorian Time Period

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

Gregorian Year:
Representation: xs:gYear (YYYY)
Period: the start of January 1 to the end of December 31
Gregorian Year Month:
Representation: xs:gYearMonth (YYYY-MM)
Period: the start of the first day of the month to end of the last day of the month
Gregorian Day:
Representation: xs:date (YYYY-MM-DD)
Period: the start of the day (00:00:00) to the end of the day (23:59:59)

4.2.5 Date Time

This is used to unambiguously state that a date-time represents an observation at a single point in time. Therefore, if one wants to use SDMX for data which is measured at a distinct point in time rather than being reported over a period, the date-time representation can be used.
Representation: xs:dateTime (YYYY-MM-DDThh:mm:ss)[2]

4.2.6 Standard Reporting Period

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

[REPORTING_YEAR]-[PERIOD_INDICATOR][PERIOD_VALUE]

Where:
REPORTING_YEAR represents the reporting year as four digits (YYYY)
PERIOD_INDICATOR identifies the type of period which determines the duration of the period
PERIOD_VALUE indicates the actual period within the year

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

Reporting Year:
Period Indicator: A
Period Duration: P1Y (one year)
Limit per year: 1
Representation: common:ReportingYearType (YYYY-A1, e.g. 2000-A1)
Reporting Semester:
Period Indicator: S
Period Duration: P6M (six months)
Limit per year: 2
Representation: common:ReportingSemesterType (YYYY-Ss, e.g. 2000-S2)

[2] The seconds can be reported fractionally

Reporting Trimester:
Period Indicator: T
Period Duration: P4M (four months)
Limit per year: 3
Representation: common:ReportingTrimesterType (YYYY-Tt, e.g. 2000-T3)

Reporting Quarter:
Period Indicator: Q
Period Duration: P3M (three months)
Limit per year: 4
Representation: common:ReportingQuarterType (YYYY-Qq, e.g. 2000-Q4)

Reporting Month:
Period Indicator: M
Period Duration: P1M (one month)
Limit per year: 1
Representation: common:ReportingMonthType (YYYY-Mmm, e.g. 2000-M12)
Notes: The reporting month is always represented as two digits, therefore 1-9 are 0 padded (e.g. 01). This allows the values to be sorted chronologically using textual sorting methods.

Reporting Week:
Period Indicator: W
Period Duration: P7D (seven days)
Limit per year: 53
Representation: common:ReportingWeekType (YYYY-Www, e.g. 2000-W53)
Notes: There are either 52 or 53 weeks in a reporting year. This is based on the ISO 8601 definition of a week (Monday - Saturday), where the first week of a reporting year is defined as the week with the first Thursday on or after the reporting year start day.[3] The reporting week is always represented as two digits, therefore 1-9 are 0 padded (e.g. 01). This allows the values to be sorted chronologically using textual sorting methods.

Reporting Day:
Period Indicator: D
Period Duration: P1D (one day)
Limit per year: 366
Representation: common:ReportingDayType (YYYY-Dddd, e.g. 2000-D366)
Notes: There are either 365 or 366 days in a reporting year, depending on whether the reporting year includes leap day (February 29). The reporting day is always represented as three digits, therefore 1-99 are 0 padded (e.g. 001). This allows the values to be sorted chronologically using textual sorting methods.

The meaning of a reporting year is always based on the start day of the year and requires that the reporting year is expressed as the year at the start of the period. This start day is always the same for a reporting year, and is expressed as a day and a month (e.g. July 1). Therefore, the reporting year 2000 with a start day of July 1 begins on July 1, 2000.

A specialized attribute (reporting year start day) exists for the purpose of communicating the reporting year start day. This attribute has a fixed identifier (REPORTING_YEAR_START_DAY) and a fixed representation (xs:gMonthDay) so

[3] 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.

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

Since the duration and the reporting year start day are known for any reporting period, it is possible to relate any reporting period to a distinct calendar period. The actual Gregorian calendar period covered by the reporting period can be computed as follows (based on the standard format of [REPROTING_YEAR]-[PERIOD_INDICATOR][PERIOD_VALUE] and the reporting year start day as [REPORTING_YEAR_START_DAY]):

1. **Determine [REPORTING_YEAR_BASE]:**
   Combine [REPORTING_YEAR] of the reporting period value (YYYY) with [REPORTING_YEAR_START_DAY] (MM-DD) to get a date (YYYY-MM-DD). This is the [REPORTING_YEAR_START_DATE]
   a) **If the [PERIOD_INDICATOR] is W:**
      1. **If [REPORTING_YEAR_START_DATE] is a Friday, Saturday, or Sunday:**
         Add[4] (P3D, P2D, or P1D respectively) to the [REPORTING_YEAR_START_DATE]. The result is the [REPORTING_YEAR_BASE].
      2. **If [REPORTING_YEAR_START_DATE] is a Monday, Tuesday, Wednesday, or Thursday:**
         Add[4] (P0D, -P1D, -P2D, or -P3D respectively) to the [REPORTING_YEAR_START_DATE]. The result is the [REPORTING_YEAR_BASE].
   b) **Else:**
      The [REPORTING_YEAR_START_DATE] is the [REPORTING_YEAR_BASE].
2. **Determine [PERIOD_DURATION]:**
   a) If the [PERIOD_INDICATOR] is A, the [PERIOD_DURATION] is P1Y.
   b) If the [PERIOD_INDICATOR] is S, the [PERIOD_DURATION] is P6M.
   c) If the [PERIOD_INDICATOR] is T, the [PERIOD_DURATION] is P4M.
   d) If the [PERIOD_INDICATOR] is Q, the [PERIOD_DURATION] is P3M.
   e) If the [PERIOD_INDICATOR] is M, the [PERIOD_DURATION] is P1M.
   f) If the [PERIOD_INDICATOR] is W, the [PERIOD_DURATION] is P7D.
   g) If the [PERIOD_INDICATOR] is D, the [PERIOD_DURATION] is P1D.
3. **Determine [PERIOD_START]:**
   Subtract one from the [PERIOD_VALUE] and multiply this by the [PERIOD_DURATION]. Add[4] this to the [REPORTING_YEAR_BASE]. The result is the [PERIOD_START].
4. **Determine the [PERIOD_END]:**

[4] 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.

Multiply the [PERIOD_VALUE] by the [PERIOD_DURATION]. Add[4] this to the [REPORTING_YEAR_BASE] add[4] -P1D. The result is the [PERIOD_END].

For all of these ranges, the bounds include the beginning of the [PERIOD_START] (i.e. 00:00:00) and the end of the [PERIOD_END] (i.e. 23:59:59).

Examples:

2010-Q2, REPORTING_YEAR_START_DAY = --07-01 (July 1)

1. [REPORTING_YEAR_START_DATE] = 2010-07-01
    b) [REPORTING_YEAR_BASE] = 2010-07-01
2. [PERIOD_DURATION] = P3M
3. (2-1) * P3M = P3M
   2010-07-01 + P3M = 2010-10-01
   [PERIOD_START] = 2010-10-01
4. 2 * P3M = P6M
   2010-07-01 + P6M = 2010-13-01 = 2011-01-01
   2011-01-01 + -P1D = 2010-12-31
   [PERIOD_END] = 2010-12-31

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

2011-W36, REPORTING_YEAR_START_DAY = --07-01 (July 1)

1. [REPORTING_YEAR_START_DATE] = 2010-07-01
    a) 2011-07-01 = Friday
       2011-07-01 + P3D = 2011-07-04
       [REPORTING_YEAR_BASE] = 2011-07-04
2. [PERIOD_DURATION] = P7D
3. (36-1) * P7D = P245D
   2011-07-04 + P245D = 2012-03-05
   [PERIOD_START] = 2012-03-05
4. 36 * P7D = P252D
   2011-07-04 + P252D =2012-03-12
   2012-03-12 + -P1D = 2012-03-11
   [PERIOD_END] = 2012-03-11

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

4.2.7 Distinct Range

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

4.2.8 Time Format

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

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)

Table 1: SDMX-ML Time Format Codes

4.2.9 Time Zones

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

- Time Range (start date): 2006-06-05**-05:00**/P5D
- Time Range (start date-time): 2006-06-05T00:00:00**-05:00**/P5D
- Gregorian Year: 2006**-05:00**
- Gregorian Month: 2006-06**-05:00**
- Gregorian Day: 2006-06-05**-05:00**
- Distinct Point: 2006-06-05T00:00:00**-05:00**
- Reporting Year: 2006-A1**-05:00**
- Reporting Semester: 2006-S2**-05:00**

- Reporting Trimester: 2006-T2**-05:00**
- Reporting Quarter: 2006-Q3**-05:00**
- Reporting Month: 2006-M06**-05:00**
- Reporting Week: 2006-W23**-05:00**
- Reporting Day: 2006-D156**-05:00**
- Reporting Year Start Day: --07-01**-05:00**

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

4.2.10 Representing Time Spans Elsewhere

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

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

can now be represented with this:

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

4.2.11 Notes on Formats

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

4.2.12 Effect on Time Ranges

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

4.2.13 Time in Query Messages

When querying for time values, the value of a time parameter can be provided as any of the Observational Time Period formats and must be paired with an operator. This section will detail how systems processing query messages should interpret these parameters.

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

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

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

Examples:

Gregorian Period

Query Parameter: Greater than 2010

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

Example Matches:

- 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)

Reporting Period

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

- 2011 or later
- 2010-07 or later
- 2010-07-01 or later
- 2010-07-01/P[Any Duration] or any later start date
- 2011-[Any reporting period] (any reporting year start day)
- 2010-S2 (any reporting year start day)
- 2010-T3 (any reporting year start day)
- 2010-Q3 or later (any reporting year start day)
- 2010-M07 or later (any reporting year start day)
- 2010-W27 or later (reporting year start day --01-01)[5]
- 2010-D182 or later (reporting year start day --01-01)
- 2010-W28 or later (reporting year start day --07-01)[6]
- 2010-D185 or later (reporting year start day --07-01)

4.3 Versioning

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

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

4.3.1 Non-versioned artefacts

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

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

[5] 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.
[6] 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.

4.3.2 Semantically versioned artefacts

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

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

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

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

1. `MAJOR` version when backwards incompatible artefact changes are made,
2. `MINOR` version when artefact elements are added in a backwards compatible manner, or
3. `PATCH` version when backwards compatible artefact property changes are made.

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

Extensions can be added, changed or dropped.

Given an extended version number `MAJOR.MINOR.PATCH-EXTENSION`, when making changes to that versioned artefact, then one is not required to increment the version if those changes are within the allowed scope of the version increment from the previous version (if that existed); otherwise, the above version increment rules apply. `EXTENSION`s can be used e.g., for drafting or a pre-release.
Semantically versioned SDMX artefacts will thus be safe to use. Specific version patterns allow them to become either immutable, i.e., the maintainer commits to never change their content, or changeable only within a well-defined scope. If any further change is required, a new version must be created first. Furthermore, the impact of the further change is communicated using a clear version increment. The built-in version extension facility allows for eased drafting of new SDMX artefact versions.

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

4.3.3 Legacy-versioned artefacts

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

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

4.3.4 Dependency management and references

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

- a fixed version of another artefact
- the **latest available** version of another artefact
- the **latest backward compatible** version of another artefact, or

the **latest backward and forward compatible** version of another artefact.

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

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

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

One area which is much impacted by this versioning scheme is the ability to reference external objects. With the many dependencies within the various structural objects in SDMX, it is useful to have a scheme for external referencing. This is done at the level of maintainable objects (DSDs, Codelists, Concept Schemes, etc.) In an SDMX Structure Message, whenever an "`isExternalReference`" attribute is set to true, then the application must resolve the address provided in the associated "`uri`" attribute and use the SDMX Structure Message stored at that location for the full definition of the object in question. Alternately, if a registry "`urn`" attribute has been provided, the registry can be used to supply the full details of the object.

The detailed rules for dependency management and references are listed in chapter 14 in the annex for “Semantic Versioning”.

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

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

The combination of wildcarded queries with a specific version extension is not permitted.
Full details can be found in the SDMX RESTful web services specification.

4.4 Structural Metadata Querying Best Practices

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

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

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

5 Reference Metadata

5.1 Scope of the Metadata Structure Definition (MSD)

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

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

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

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

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

The schematic structure of an MSD is shown below.

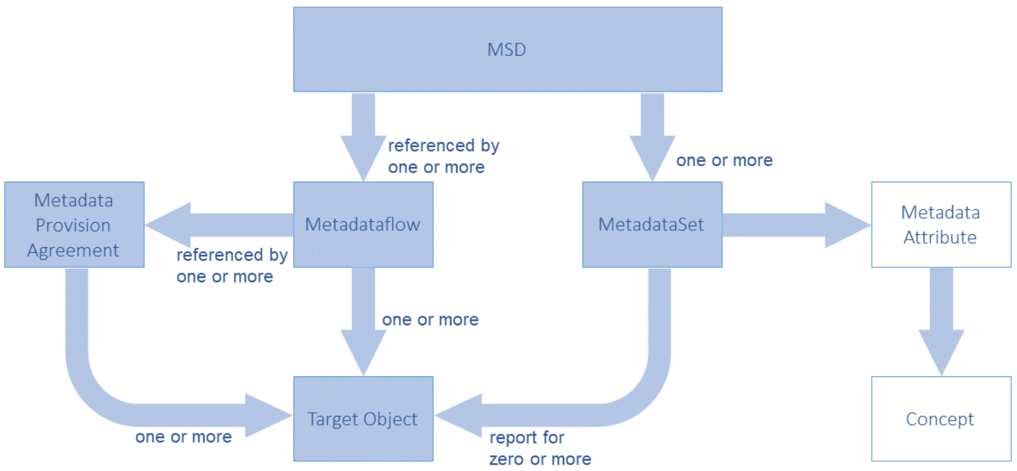


Figure 1: Schematic of the Metadata Structure Definition

The MSD contains one Metadata Attribute Descriptor comprising the Metadata Attributes that identify the Concepts for which metadata may be reported in the Metadata Set. The Metadataflow and Metadata Provision Agreement comprise the specification of the objects to which metadata can be reported in a Metadata Set (Metadata Target(s)).

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

```
<str:MetadataStructure agencyID="SDMX" id="MSD" version="1.0.0-draft">
  <com:Name>MSD 3.0 sample</com:Name>
  <str:MetadataAttributeDescriptor id="MetadataAttributeDescriptor">
    ...
  </str: MetadataAttributeDescriptor>
</str:MetadataStructure>
```

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

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

Figure 3: Wildcarded Target Objects as specified in a Metadataflow

```
<str:MetadataProvisionAgreement agencyID="OECD" id="ABS_INDICATORS"
version="1.0.0-draft">
  <com:Name xml:lang="en">Metadata Provision Agreement 3.0 sample</com:Name>
  <str:StructureUsage>urn:sdmx:org.sdmx.infomodel.metadatastructure.
    Metadataflow=OECD:GENERAL_METADATA(1.0.0-draft)</str:StructureUsage>
  <str:MetadataProvider>urn:sdmx:org.sdmx.infomodel.base.
    MetadataProvider=OECD:METADATA_PROVIDERS(1.0).ABS</str:MetadataProvider>
  <!-- Attach to specific Dataflows maintained by the OECD -->
  <str:Target>urn:sdmx:org.sdmx.infomodel.datastructure.Dataflow=
    OECD:GDP(*)</str:Target>
  <str:Target>urn:sdmx:org.sdmx.infomodel.datastructure.Dataflow=
    OECD:EXR(*)</str:Target>
  <str:Target>urn:sdmx:org.sdmx.infomodel.datastructure.Dataflow=
    OECD:ABC(*)</str:Target>
</str:MetadataProvisionAgreement>
```

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

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

5.3 Metadata Structure Definition

An example is shown below.

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

```
          Concept=SDMX:CONCEPTS(1.0.0).CONTACT</str:ConceptIdentity>
        <str:MetadataAttribute id="CONTACT_NAME" minOccurs="1" maxOccurs="1">
          <str:ConceptIdentity>urn:sdmx:org.sdmx.infomodel.conceptscheme.
            Concept=SDMX:CONCEPTS(1.0.0).CONTACT_NAME</str:ConceptIdentity>
          <str:LocalRepresentation>
            <str:TextFormat textType="String"/>
          </str:LocalRepresentation>
        </str:MetadataAttribute>
        <str:MetadataAttribute id="ADDRESS" minOccurs="1" maxOccurs="3"
isPresentational="true">
          <str:ConceptIdentity>urn:sdmx:org.sdmx.infomodel.conceptscheme.
            Concept=SDMX:CONCEPTS(1.0.0).ADDRESS</str:ConceptIdentity>
          <str:MetadataAttribute id="HOUSE_NUMBER" minOccurs="1"
maxOccurs="1">
            <str:ConceptIdentity>urn:sdmx:org.sdmx.infomodel.conceptscheme.
              Concept=SDMX:CONCEPTS(1.0.0).HOUSE_NUMBER</str:ConceptIdentity>
            <str:LocalRepresentation>
              <str:TextFormat textType="Integer"/>
            </str:LocalRepresentation>
          </str:MetadataAttribute>
        </str:MetadataAttribute>
      </str:MetadataAttribute>
    </str:MetadataAttributeDescriptor>
</str:MetadataStructure>
```

Figure 5: Example MSD showing specification of some Metadata Attributes

This example shows the following hierarchy of Metadata Attributes:

- Contact – this is presentational; no metadata is expected to be reported at this level
    - Contact Name
    - Address – this is also presentational; up to 3 addresses are allowed
        - House Number

5.4 Metadata Set

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

```
<msg:MetadataSet id="ALB" metadataProviderID="OECD" version="1.0.0">
   <str:MetadataProvision>urn:sdmx:org.sdmx.infomodel.registry.MetadataProvis
ionAgreement=OECD:ABS_INDICATORS(1.0.0-draft)</str:MetadataProvision>
   <str:Target>urn:sdmx:org.sdmx.infomodel.datastructure.Dataflow=OECD:GDP(1.
0.0)</str:Target>
   <md:AttributeSet>
      <md:ReportedAttribute id="CONTACT">
         <md:AttributeSet>
            <md:ReportedAttribute id="CONTACT_NAME">John Doe
            </md:ReportedAttribute>
            <md:ReportedAttribute id="ADDRESS">
               <md:AttributeSet>
                  <md:ReportedAttribute id="STREET_NAME">
                     <com:Text xml:lang="en">5th Avenue</com:Text>
                  </md:ReportedAttribute>
                  <md:ReportedAttribute id="HOUSE_NUMBER">12
                  </md:ReportedAttribute>
               </md:AttributeSet>
            </md:ReportedAttribute>
            <md:ReportedAttribute id="HTML_ATTR">
               <com:StructuredText xml:lang="en">
                  <div xmlns="http://www.w3.org/1999/xhtml">
```

```
                <p>Lorem Ipsum</p>
              </div>
            </com:StructuredText>
          </md:ReportedAttribute>
        </md:AttributeSet>
      </md:ReportedAttribute>
    </md:AttributeSet>
</msg:MetadataSet>
```

Figure 6: Example Metadata Set

This example shows:

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

5.5 Reference Metadata in Data Structure Definition and Dataset

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

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

6 Codelist

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

6.1 Codelist extension and discriminated unions

A `Codelist` can extend one or more `Codelist`s. `Codelist` extensions are defined as a list of references to parent `Codelist`s. The order of the references is important when it comes to conflict resolution on `Code Id`s. When two `Codelist`s have the same `Code Id`, the `Codelist` referenced later takes priority. In the example below, the code '`A`', exists in both `CL_INDICATOR` and `CL_SERIES`. The `Codelist` `CL_INDICATOR_EX` will contain the code '`A`' from `CL_SERIES` as this was the second `Codelist` to be referenced in the sequence of references.

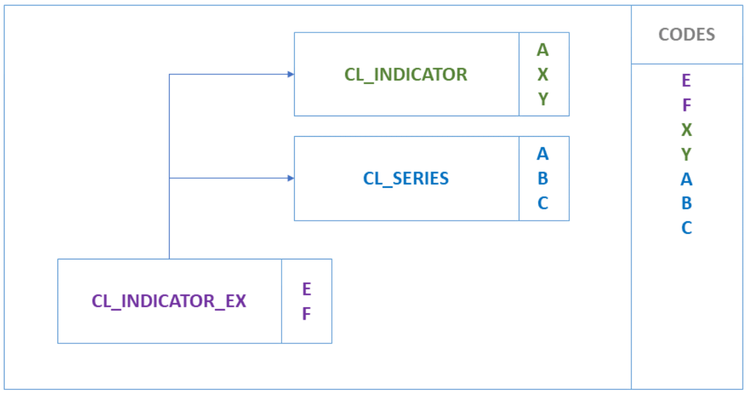


Figure 7: Codelist extension

As the extended `Codelist`, `CL_INDICATOR_EX` in this example, may also define its own `Code`s, these take the ultimate priority over any referenced `Codelist`s. If `CL_INDICATOR_EX` defines Code '`A`', then this will be used instead of `Code` '`A`' from `CL_INDICATOR` and `CL_SERIES`, as shown below:

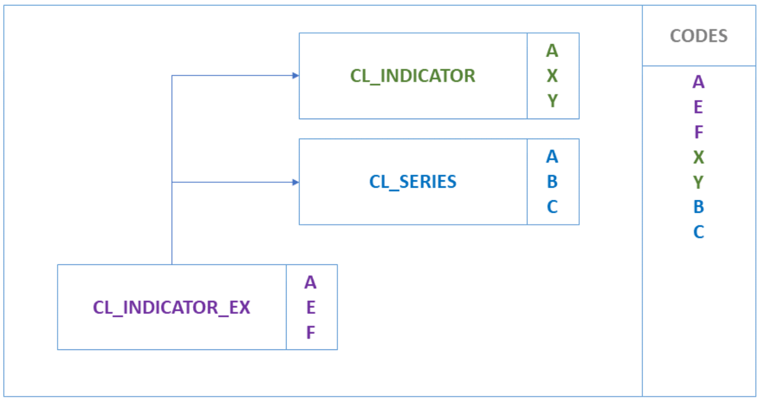


Figure 8: Codelist extension with new Codes

6.1.1 Prefixing Code Ids

A reference to a `Codelist` may contain a prefix. If a prefix is provided, this prefix will be applied to all the codes in the `Codelist` before they are imported into the extended `Codelist`. Following the above example if the `CL_SERIES` reference includes a prefix of '`SER_`' then the resulting `Codelist` would contain 7 codes, `A`, `E`, `F`, `X`, `Y`, `SER_A`, `SER_B`, `SER_C`.

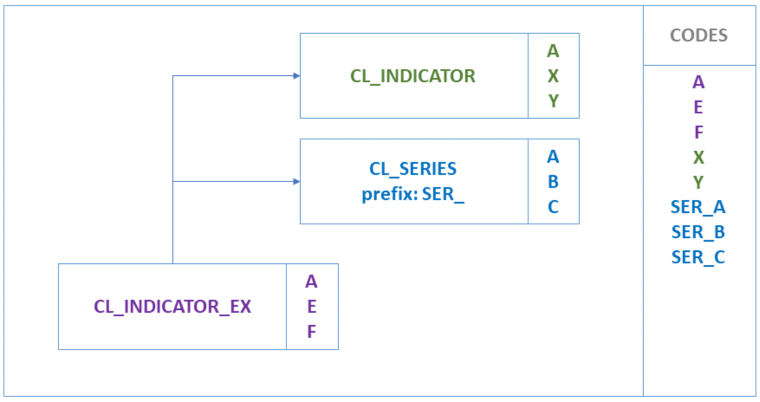


Figure 9: Extended Codelist with prefix

6.1.2 Including / Excluding Specific Codes

The default behaviour of extending another `Codelist` is to import all `Code`s. However, an explicit list of `Code Id`s may be provided for explicit inclusion or exclusion. This list of `Id`s may contain wildcards using the same notation as `Constraint`s (`%`). Cascading values is also supported using the same syntax as the `Constraint`s. The list of `Id`s is either a list of excluded items, or included items, exclusion and inclusion is not supported against a single `Codelist`.

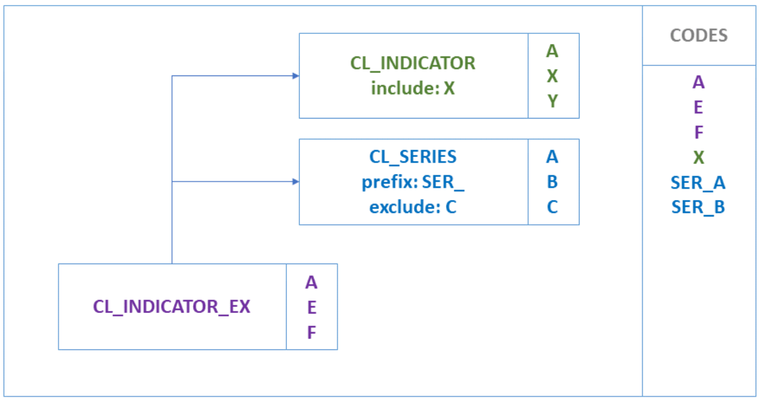


Figure 10: Extended Codelist with include/exclude terms

6.1.3 Parent Ids

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

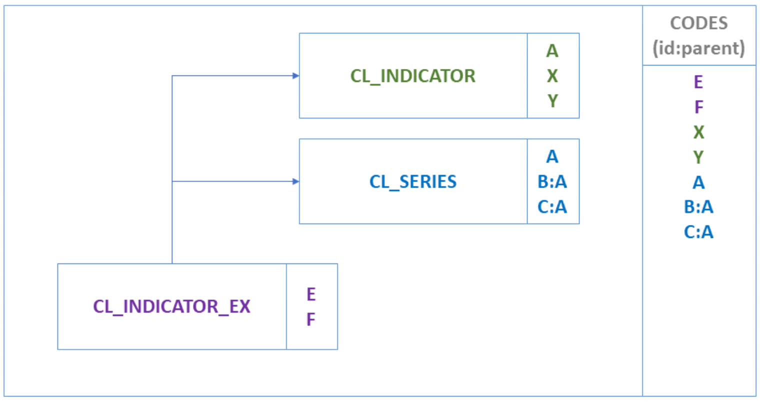
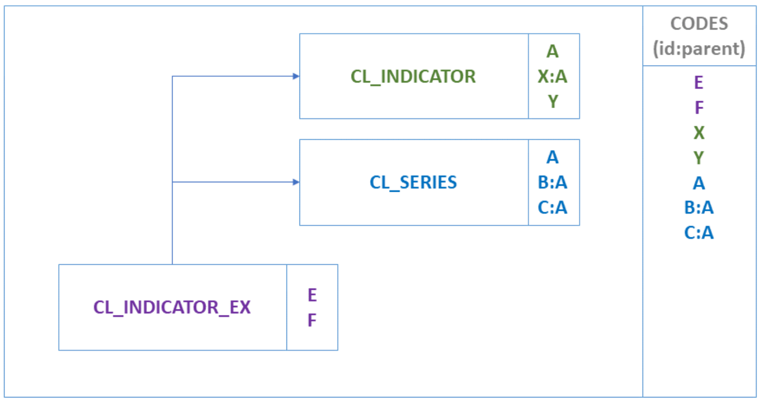
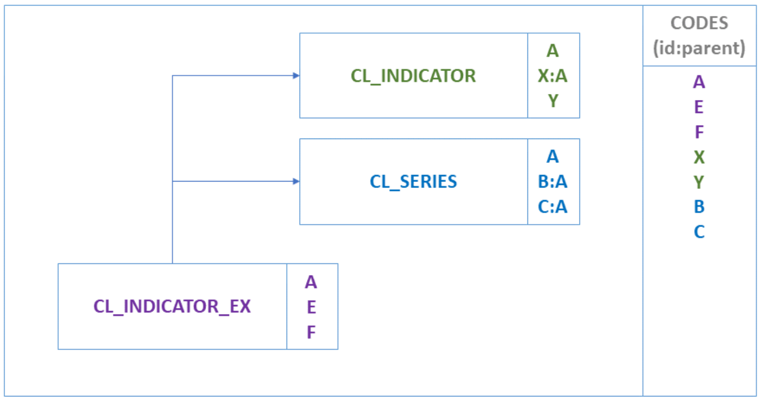


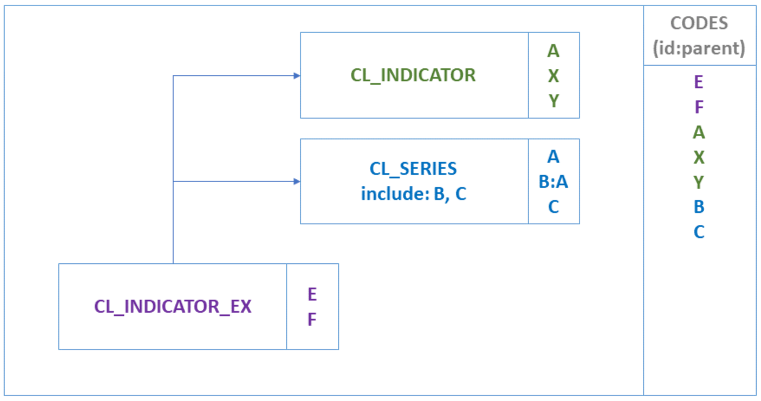
Figure 11: Parent Code included



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**

6.1.4 Discriminated Unions

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

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

- **Independent Codelists per variant**: Having each variant in a separate `Codelist` facilitates the maintenance and allows keeping the original codes, even if different versions of the classification have the same code for different concepts. For example, in ISIC Rev. 4 the code "A" represents "Agriculture, forestry and fishing", while in ISIC 3.1 "A" means "Agriculture, hunting and forestry".
- **Prefixing Code Ids**: When extending `Codelist`s, the reference to an extension `Codelist` may contain a prefix. If a prefix is provided, this prefix will be applied to all the codes in the `Codelist` before they are imported into the extended `Codelist`. In this case, the reference to `CL_ISIC4` includes a prefix of "`ISIC4_`" and the reference to `ISIC3` includes "`ISIC3_`", so the resulting `Codelist` will have no conflict for the "`A`" items which will become "`ISIC3_A`" and "`ISIC4_A`".
- **Including / Excluding Specific Codes**: As explained above, there will be independent DFs/PAs with specific `Constraint` attached, in order to keep the proper items according to the variant in use by each data provider.

For example, assuming:

- DSD `DSD_EXDU` contains a Dimension: `ACTIVITY` enumerated by `CL_ACTIVITY`.
- `CL_ACTIVITY` has no items and is extended by:
- `CL_ISIC4`, `prefix="ISIC4_"`
- `CL_ISIC3`, `prefix="ISIC3_"`
- `CL_NACE2`, `prefix="NACE2_"`
- `CL_AGGR`, `prefix="AGGR_"`
- Dataflow `DF1`, with a `DataConstraint` `CC_NACE2`, `CubeRegion` for `ACTIVITY` and Value="`NACE2_%`"
- Dataflow `DF2`, with a `DataConstraint` `CC_ISIC3`, `CubeRegion` for `ACTIVITY` and Value="`ISIC3_%`"
- Dataflow `DF3`, with a `DataConstraint` `CC_ISIC4`, `CubeRegion` for `ACTIVITY` and Value="`ISIC4_%`", Value="`AGGR_TOTAL`", Value="`AGGR_Z`"

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

DataConstraint, referencing the Dataflow, applied. Thus, the CL_ACTIVITY will only include Codes prefixed according to the Dataflow, i.e.:

- Prefix "NACE2_%" for DF1;
- Prefix "ISIC3_%" for DF2;
- Prefix "ISIC4_%" for DF3; note that Codes "AGGR_TOTAL" and "AGGR_Z" are also included in this case.

6.2 Linking Hierarchies

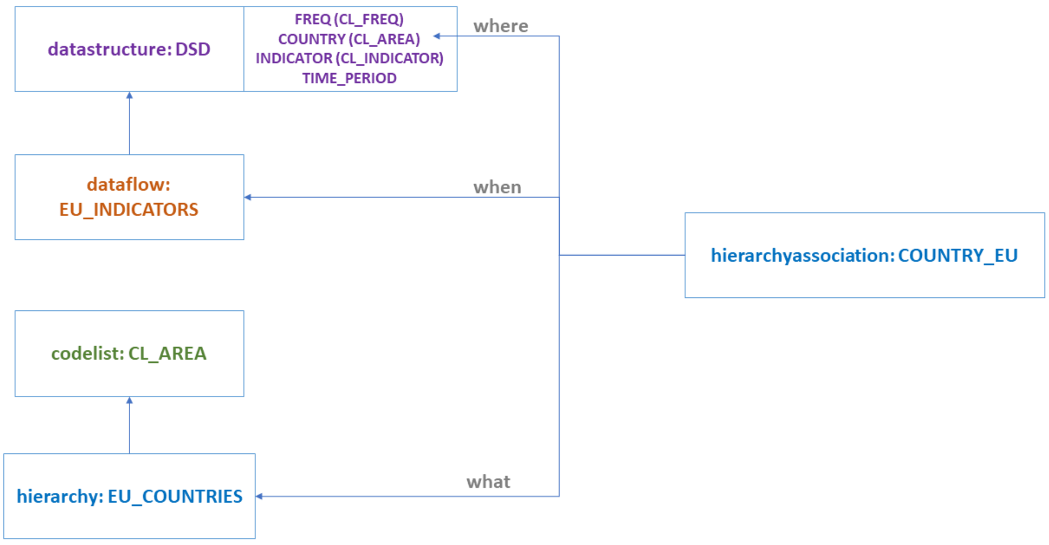
To facilitate the usage of Hierarchy within other SDMX Artefacts, the HierarchyAssociation is defined to link any Hierarchy with any IdentifiableArtefact within a specific context.

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

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

As an example, let's assume:

- A DSD with a **COUNTRY** Dimension that uses Codelist **CL_AREA** as representation.
- A Hierarchy (e.g., **EU_COUNTRIES**) that builds a hierarchy for the **CL_AREA** Codelist.
- In order to use this Hierarchy for data of a Dataflow (e.g., **EU_INDICATORS**), we need to build the following HierarchyAssociation:
- Links to the Hierarchy **EU_COUNTRIES (what is associated?)**
- Links to the Dimension **COUNTRY (where is it associated?)**
- Links to the context: Dataflow **EU_INDICATORS (when is it associated?)**
- The above are also shown in the schematic below:



-

Figure 15: Hierarchy Association

-

7 Geospatial information support

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

SDMX can support three different ways for referencing geospatial data:

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

7.1 Indirect Reference to Geospatial Information.

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

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

- **GEO_INFO_TEXT**. A description of the kind of information being referenced.
- **GEO_INFO_URL**. A URL which points to the resource containing the referred geospatial information. The resource might be a file with static geodesic information or a web service providing dynamic construction of geometries.
- **GEO_INFO_TYPE**. Coded information describing a standard format of the file that contains the geospatial information. The format types are taken from the list of Format descriptions for Geospatial Data managed by the US Library of the Congress (https://www.loc.gov/preservation/digital/formats/fdd/gis_fdd.shtml). Allowed types in SDMX are listed in the **Geographical Formats** code list (**CL_GEO_FORMATS**). Examples of the codes contained in the document are:

• Code	• Description
• `GML`	• Geography Markup Language

• **GeoTIFF**	• GeoTIFF
• **KML_2_2**	• KML Version 2.2
• **GEOJSON_1_1**	• GeoJSON Version 1.1

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.

7.2 Geographic Coordinates

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

The `GeospatialInformation` data type can be assigned to a `Dimension`, `DataAttribute`, `MetadataAttribute` or a `Measure` with the "`GEO_FEATURE_SET`" role assigned; it can be included in a dataset or metadataset.

Any `Component` used for representing a Geographical Feature Set, i.e., used to describe geographical characteristics, must have a “`GEO_FEATURE_SET`” role. Its Representation would be of `textType="GeospatialInformation"`. The `GeospatialInformation` type is not intended to replace standard geospatial information formats, but instead provide a simple way to describe precise geographical characteristics in SDMX data sets agnostic of any particular geospatial software product or use case.

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

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

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

- `CRS`: The code of the Coordinate Reference System is used to reference the coordinates in the flow, optional.

  The code of the CRS will be as it appears in the EPSG Geodetic Parameter Registry (http://www.epsg-registry.org/) maintained by the International Association of Oil and Gas Producers. If this element is omitted, by default, the CRS will be the World Geodetic System 1984 (WGS 84, EPSG:4326).
- `PRECISION`: Precision of the coordinates, expressing the possible deviation in meters from the exact geodesic point, optional.

  This component is only allowed if the CRS is specified too. If omitted, it will be interpreted as limited it to the expected measurement accuracy (e. g. a standard GPS has an accuracy of ~ 10m).
- `GEO_DESCRIPTION`: Text for additional information about the place, geographical feature, or set of geographical features, optional.

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

`(GEO_FEATURE, GEO_FEATURE): GEO_DESCRIPTION`

- `GEO_FEATURE` represents a set of points defining a feature following the ISO/IEC 13249-3:2016 standard to conform Well-known Text (WKT) for the representation of geometries in a format defined in the following way:

  `GEOMETY_TYPE (GEOMETRY_REP)`

- `GEOMETRY_TYPE`: A string with a closed vocabulary defining the type of the geometry that represents a geographical component of the `GEO_FEATURES` collection, mandatory.

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

  If the `GEOMETRY_REP` is going to be including the height (ALT) component, a "Z" must be added after the string qualifying the `GEOMETRY_TYPE`. In this way, we will have: "`POINT Z`", "`LINESTRING Z`" and "`POLYGON Z`"

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

- `GEOMETRY_REP`: Representation of each of the types The way to represent each `GEO_FEATURE_TYPE` will be:

- A point (`POINT`): “`COORDINATES`”
- A line (`LINESTRING`): “`COORDINATES, COORDINATES, …`”
- An area (`POLYGON`): “`(COORDINATES, COORDINATES, …), (COORDINATES, COORDINATES, …)`”

Where:

- `COORDINATES`: Represents an individual set of coordinates composed by the `X_COORDINATE (X)`, `Y_COORDINATE (Y)`, and `ALT (Z)` in the following way “`X Y Z`” or “`X Y`” defining a single point of the polygon. Altitude is to be reported in meters.

In an expanded way, `GEO_FEATURE` may be represented in the following ways:

```
POINT (X_COORDINATE Y_COORDINATE): GEO_DESCRIPTION
POINT Z (X_COORDINATE Y_COORDINATE ALT): GEO_DESCRIPTION
LINESTRING (X_COORDINATE Y_COORDINATE, X_COORDINATE
Y_COORDINATE, …): GEO_DESCRIPTION
LINESTRING Z (X_COORDINATE Y_COORDINATE ALT, X_COORDINATE
Y_COORDINATE ALT, …): GEO_DESCRIPTION
POLYGON  ((X_COORDINATE Y_COORDINATE, X_COORDINATE
Y_COORDINATE, …), (X_COORDINATE Y_COORDINATE, X_COORDINATE
Y_COORDINATE, …), …): GEO_DESCRIPTION
POLYGON Z ((X_COORDINATE Y_COORDINATE ALT, X_COORDINATE
Y_COORDINATE ALT, …), (X_COORDINATE Y_COORDINATE ALT,
X_COORDINATE Y_COORDINATE ALT, …), …): GEO_DESCRIPTION
```

An example of how `GEO_FEATURES` may be represented in an expanded way would be:

```
(POLYGON Z ((X_COORDINATE Y_COORDINATE ALT, X_COORDINATE
Y_COORDINATE ALT, …), (X_COORDINATE Y_COORDINATE ALT,
X_COORDINATE Y_COORDINATE ALT, …), …), POLYGON Z
((X_COORDINATE Y_COORDINATE ALT, X_COORDINATE Y_COORDINATE
ALT, …), (X_COORDINATE Y_COORDINATE ALT, X_COORDINATE
Y_COORDINATE ALT, …), …), POLYGON Z ((X_COORDINATE
Y_COORDINATE ALT, X_COORDINATE Y_COORDINATE ALT, …),
(X_COORDINATE Y_COORDINATE ALT, X_COORDINATE Y_COORDINATE ALT,
…), …), …): GEO_DESCRIPTION
```

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

```
"CRS, PRECISION: {(POLYGON Z ((X_COORDINATE Y_COORDINATE ALT,
X_COORDINATE Y_COORDINATE ALT, …), (X_COORDINATE Y_COORDINATE
ALT, X_COORDINATE Y_COORDINATE ALT, …), …), POLYGON Z
((X_COORDINATE Y_COORDINATE ALT, X_COORDINATE Y_COORDINATE
ALT, …), (X_COORDINATE Y_COORDINATE ALT, X_COORDINATE
Y_COORDINATE ALT, …), …), POLYGON Z ((X_COORDINATE
Y_COORDINATE ALT, X_COORDINATE Y_COORDINATE ALT, …),
(X_COORDINATE Y_COORDINATE ALT, X_COORDINATE Y_COORDINATE ALT,
…), …), …): GEO _DESCRIPTION}, {(POLYGON Z ((X_COORDINATE
Y_COORDINATE ALT, X_COORDINATE Y_COORDINATE ALT, …),
(X_COORDINATE Y_COORDINATE ALT, X_COORDINATE Y_COORDINATE ALT,
…), …), POLYGON Z ((X_COORDINATE Y_COORDINATE ALT,
X_COORDINATE Y_COORDINATE ALT, …), (X_COORDINATE Y_COORDINATE
ALT, X_COORDINATE Y_COORDINATE ALT, …), …), POLYGON Z
((X_COORDINATE Y_COORDINATE ALT, X_COORDINATE Y_COORDINATE
ALT, …), (X_COORDINATE Y_COORDINATE ALT, X_COORDINATE
Y_COORDINATE ALT, …), …), …): GEO _DESCRIPTION}, …:
GEO_DESCRIPTION"
```

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

7.3 A Geographic Codelist

Geography is represented by geospatial information. Within SDMX, geospatial information is conceptually represented by the "`GEO_FEATURE_SET`" role/specification. This approach uses a specialized form of SDMX Codelist, named "`GeoCodelist`", which is a Codelist containing the Geography used to demarcate the geographic extent. This is implemented in two ways:

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

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

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

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

- `CRS`: The code of the Coordinate Reference System is used to reference the coordinates in the flow, optional. The code of the CRS will be as it appears in the EPSG Geodetic Parameter Registry (http://www.epsg-registry.org/) maintained by the International Association of Oil and Gas Producers. If this component is omitted, by default the CRS will be the World Geodetic System 1984 (WGS 84, EPSG:4326).

- `REFERENCE_CORNER`: A code composed of two characters to define the position of the coordinates that will be used as a starting reference to locate the cells. The possible values of this code can be `UL` (Upper Left), `UR` (Upper Right), `LL` (Lower Left), or `LR` (Lower Right). If this component is omitted the value `LL` (Lower Left) will be taken by default. This element is optional.

- `REFERENCE_COORDINATES`: Represents the starting point to reference the cells of the grid, accordingly to the `CRS` and the `REFERENCE_CORNER`. It is represented by an individual set of coordinates composed by the `X_COORDINATE (X)` and `Y_COORDINATE (Y)` in the following way "`X,Y`". This element is mandatory if `GEO_STD` is omitted.
- `CELL_WIDTH`: The size in meters of a horizontal side of the cells in the grid. This element is mandatory if `GEO_STD` is omitted.
- `CELL_HEIGHT`: The size in meters of a vertical side of the cells in the grid. This element is mandatory if `GEO_STD` is omitted .
- `GEO_STD`: A restricted text value expressing that the cells in the grid will provide information about matching codes existing in another reference system that establishes a mechanism to define the grid. This element is optional.

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

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

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

In an expanded way we would have:
"CRS:REFERENCE_CORNER; X_COORDINATE, Y_COORDINATE; CELL_WIDTH, CELL_HEIGHT: GEO_STD"

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

- `GEO_COL`: The number of the column in the grid starting by zero.
- `GEO_ROW`: The number of the row in the grid starting by zero.
- `GEO_TAG`: An optional text to include additional information to the cell.

- `GEO_CELL` will have values with the following format: "**`GEO_COL, GEO_ROW: GEO_TAG`**"

When using a gridded `GeoCodelist` we may use the `GEO_TAG` to integrate the cells in the grid to the codes used by other standard defined grids. As an example, `GEO_TAG` can take the values of the Open Location Codes, GeoHash, etc. If this is done, the `GEO_STD` component must have been added to the definition of the grid. If the `GEO_STD` is omitted, the `GEO_TAG` contents will be taken just as free text.

-

8 Maintenance Agencies and Metadata Providers

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

1. Agencies are maintained in an `AgencyScheme` (which is a sub class of *`OrganisationScheme`*); Metadata Providers are maintained in a `MetadataProviderScheme`.
2. The maintenance agency of the Agency/Metadata Provider Scheme must also be declared in a (different) `AgencyScheme`.
3. The "top-level" agency is SDMX and this agency scheme is maintained by SDMX.
4. Agencies registered in the top-level scheme can themselves maintain a single `AgencyScheme` and a single `MetadataProviderScheme`. SDMX is an agency in the SDMX `AgencyScheme`. Agencies in any `AgencyScheme` can themselves maintain a single `AgencyScheme` and so on.
5. The `AgencyScheme` and `MetadataProvideScheme` cannot be versioned and thus have a fixed version set to ‘1.0’.
6. There can be only one `AgencyScheme` maintained by any one Agency. It has a fixed Id of '`AGENCIES`'. Similarly, only one `MetadataProvideScheme` is maintained by one Agency and has a fixed id of '`METADATA_PROVIDERS`'.
7. The format of the agency identifier is `agencyId.agencyID` etc. The top-level agency in this identification mechanism is the agency registered in the SDMX agency scheme. In other words, SDMX is not a part of the hierarchical ID structure for agencies. SDMX is, itself, a maintenance agency.

This supports a hierarchical structure of `agencyID.`

An example is shown below.

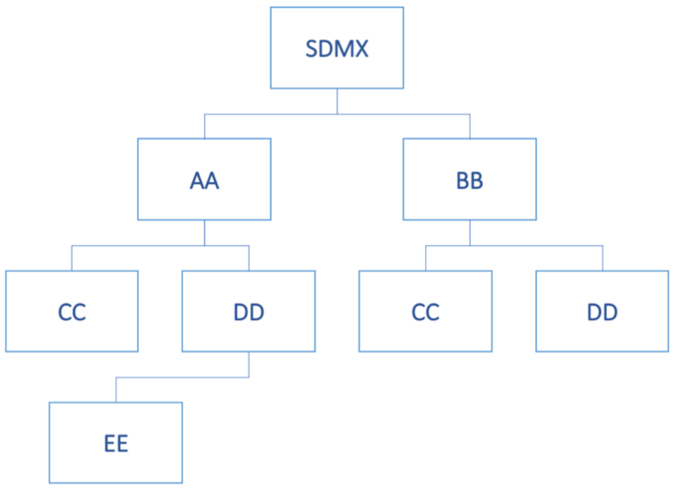


Figure 16: Example of Hierarchic Structure of Agencies

Each agency is identified by its full hierarchy excluding SDMX.

The XML representing this structure is shown below.

```
<str:AgencySchemes>
  <str:AgencyScheme agencyID="SDMX" id="AGENCIES">
    <com:Name xml:lang="en">Top-level Agency Scheme</com:Name>
    <str:Agency id="AA">
      <com:Name xml:lang="en">AA Name</com:Name>
    </str:Agency>
    <str:Agency id="BB">
      <com:Name xml:lang="en">BB Name</com:Name>
    </str:Agency>
  </str:AgencyScheme>

  <str:AgencyScheme agencyID="AA" id="AGENCIES">
    <com:Name xml:lang="en">AA Agencies</com:Name>
    <str:Agency id="CC">
      <com:Name xml:lang="en">CC Name</com:Name>
    </str:Agency>
    <str:Agency id="DD">
      <com:Name xml:lang="en">DD Name</com:Name>
    </str:Agency>
  </str:AgencyScheme>

  <str:AgencyScheme agencyID="BB" id="AGENCIES">
    <com:Name xml:lang="en">BB Agencies</com:Name>
    <str:Agency id="CC">
      <com:Name xml:lang="en">CC Name</com:Name>
    </str:Agency>
    <str:Agency id="DD">
      <com:Name xml:lang="en">DD Name</com:Name>
    </str:Agency>
  </str:AgencyScheme>

  <str:AgencyScheme agencyID="AA.DD" id="AGENCIES">
    <com:Name xml:lang="en">AA.DD Agencies</com:Name>
    <str:Agency id="EE">
      <com:Name xml:lang="en">EE Name</com:Name>
    </str:Agency>
  </str:AgencyScheme>

</str:AgencySchemes>
```

Figure 17: Example Agency Schemes Showing a Hierarchy

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

```
  <str:Codelist  agencyID="SDMX" id="CL_FREQ" version="1.0.0"
    urn="urn:sdmx:org.sdmx.infomodel.codelist.Codelist=SDMX:CL_FREQ(1.0.0)">
    <com:Name xml:lang="en">Frequency</com:Name>
  </str:Codelist>
  <str:Codelist  agencyID="AA" id="CL_FREQ" version="1.0.0"
    urn="urn:sdmx:org.sdmx.infomodel.codelist.Codelist=AA:CL_FREQ(1.0.0)">
    <com:Name xml:lang="en">Frequency</com:Name>
  </str:Codelist>
  <str:Codelist  agencyID="AA.CC" id="CL_FREQ" version="1.0.0"
   urn="urn:sdmx:org.sdmx.infomodel.codelist.Codelist=AA.CC:CL_FREQ(1.0.0)">
    <com:Name xml:lang="en">Frequency</com:Name>
  </str:Codelist>
  <str:Codelist  agencyID="BB.CC" id="CL_FREQ" version="1.0.0"
   urn="urn:sdmx:org.sdmx.infomodel.codelist.Codelist=BB.CC:CL_FREQ(1.0.0)">
```

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

Figure 18: Example Showing Use of Agency Identifiers

Each of these maintenance agencies has a `Codelist` with an identical id '`CL_FREQ`'. However, each is uniquely identified by means of the hierarchic agency structure.

9 Concept Roles

9.1 Overview

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

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

9.2 Information Model

The Information Model for this is shown below:

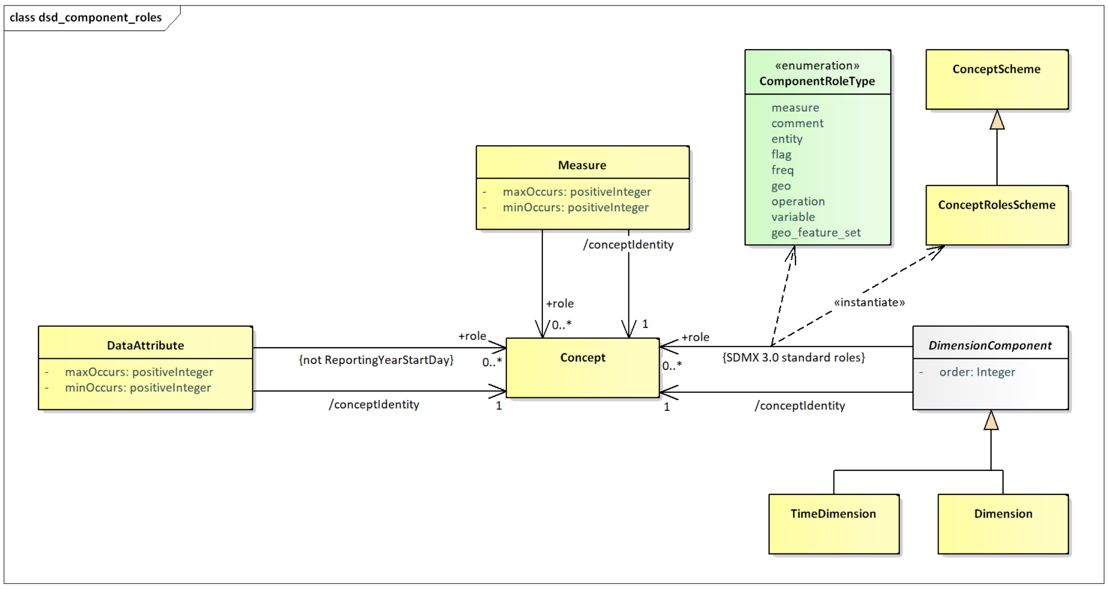


Figure 19: Information Model Extract for Concept Role

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

9.3 Technical Mechanism

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

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

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

9.4 SDMX-ML Examples in a DSD

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

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

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

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

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

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

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

```
<str:Dimension id="FREQ">
  <str:ConceptIdentity>urn:sdmx:org.sdmx.infomodel.conceptscheme.Concept=
         SDMX:CONCEPTS(1.0.0).FREQ</str:ConceptIdentity>
  <str:ConceptRole>urn:sdmx:org.sdmx.infomodel.conceptscheme.Concept=
         SDMX:MY_CONCEPT_ROLES(1.0.0).FREQ</str:ConceptRole>
</str:Dimension>
```

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

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

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

9.5 SDMX standard roles Concept Scheme

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

10 Constraints

10.1 Introduction

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

- Availability Constraint
- Dimension Constraint
- Reporting Constraints

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

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

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

10.2 Availability Constraint

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

- Data Structure Definition
- Dataflow
- Provision Agreement

10.3 Dimension Constraint

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

Dimension Constraints were introduced in SDMX 3.1 and are not required for most Dataflows where the dataset must always contain the full complement of Dimensions as defined by the corresponding DSD. However, for some complex data collections, which may span long periods and where the full complement of required Dimensions are not necessarily known at design time, the DSD is subject to increasing its Dimensionality over time. In this scenario it is possible to define the DSD as an evolving structure, this property tells the user that the DSD can have new Dimensions added without having to undergo a major version change; a DSD at version 1.0.0 for example would be able to add a new Dimension and move to version 1.1.0; a change that would not ordinarily be allowed. A minor version change on the addition of a new Dimension is only possible if the DSD defines itself as an evolving structure. This is a new property of the DSD introduced in version 3.1 to satisfy this use case. The evolving structure property is either true or false, defaulting to false if not specified. Setting the evolving structure property to true requires a major version change, and therefore can only be introduced on an x.0.0 release (e.g. 1.0.0). The evolving structure property can be set to false to indicate that there will be no additional Dimensions added to the Data Structure under the same major version number; setting the evolving structure property to false does not require require a major version change on the Data Structure.

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

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

<u>Rules for a Dimension Constraint</u>

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

<u>Example Datasets with Evolving Structures</u>

A dataset is built against a Data Structure Definition. The dataset contains data for two Dataflows. Dataflows 'DF_POP' uses a Dimension Constraint which fixes its Dimensions to FREQ and REF_AREA. Dataflow 'DF_POP_SA' does not reference a Dimension Constraint, and as such includes all Dimensions as specified by the DSD.

The resulting dataset contains values '~' for both the SEX and AGE Dimension for the 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

10.4 Reporting Constraints

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

A reporting constraint is one of the following concrete types:

- Data Constraint
- Metadata Constraint

10.4.1 Data Constraint

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

- Data Structure Definition
- Dataflow
- Provision Agreement
- Data Provider

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

10.4.2 Metadata Constraint

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

- Metadata Structure Definition
- Metadataflow
- Metadata Provision Agreement
- Metadata Provider

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

10.4.3 Scope of a Constraint

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

In terms of data the components are:

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

In terms of reference metadata the components are:

- Metadata Attribute

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

- Data Structure Definition
- Dataflow
- Provision Agreement
- Data Provider

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

- Metadata Structure Definition
- Metadataflow
- Metadata Provision Agreement
- Metadata Provider
- Metadata Set

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

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

10.4.4 Multiple Constraints

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

10.4.4.1 Cube Region

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

- A specific Member Selection (e.g., Dimension FREQ) can only be contained in one Cube Region for any one attached object (e.g., a specific DSD or specific Dataflow).
- Component values within a Member Selection may define a validity period. Otherwise, the value is valid for the whole validity of the Cube Region.
- For partial reference resolution purposes (as per the SDMX REST API), the latest non-draft Constraint must be considered.
- A Member Selection may include wildcarding of values (using character '%' to represent zero or more occurrences of any character), as well as cascading through hierarchic structures (e.g., parents in Codelist), or localised values (e.g., text for English only). Lack of locale means any language may match. Cascading values are mutual exclusive to localised values, as the former refer to coded values, while the latter refer to uncoded values.
- Any values included in a Member Selection for Components with an array data type (i.e., Measures, Attributes or Metadata Attributes), will be applied as single values and will not be assessed combined with other values to match all possible array values. For example, including the Code 'A' for an Attribute will allow any instance of the Attribute that includes 'A', like ['A', 'B'] or ['A', 'C', 'D']. Similarly, if Code 'A' was excluded, all those arrays of values would also be excluded.

10.4.4.2 Key Set

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

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

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

Finally, a validity period may be specified per Key.

10.4.5 Versioning

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

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

Example:

Data Constraint 1.0.0

Component	Valid Value	Valid from	Valid to
COUNTRY	UK		
	FR		
	DE		

Data Constraint 1.1.0

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

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.

10.4.6 Inheritance

10.4.6.1 Attachment levels of a Constraint

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

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.

10.4.6.2 Cascade rules for processing Constraints

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.

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.

10.4.6.3 Cube Region

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

- The Constraint at the lower level for any one Member Selection further constrains the content for the same Member Selection at the higher level(s).
- Any Member Selection, which is not referenced in a Constraint, is deemed to be constrained according to the Constraint specified at the next higher level which constraints that Member Selection.
- If there is a conflict when resolving the Constraint in terms of a lower-level Constraint being less restrictive than a higher-level Constraint, then the Constraint at the higher-level is used.

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

10.4.6.4 Key Set

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 at the higher level.
- The Constraint at the lower level for any one Member Selection further constrains the keys specified at the higher level(s).
- Any Member Selection, which is not referenced in a Constraint, is deemed to be constrained according to the Constraint specified at the next higher level which constraints that Member Selection.
- If there is a conflict when resolving the keys in the Constraint at two levels, in terms of a lower-level constraint being less restrictive than a higher-level Constraint, then the offending keys specified at the lower level are not deemed part of the Constraint.

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

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

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

10.4.7 Constraints Examples

10.4.7.1 Data Constraint and Cascading

The following scenario is used.

A DSD contains the following Dimensions:

- GEO – Geography
- SEX – Sex
- AGE – Age
- CAS – Current Activity Status

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

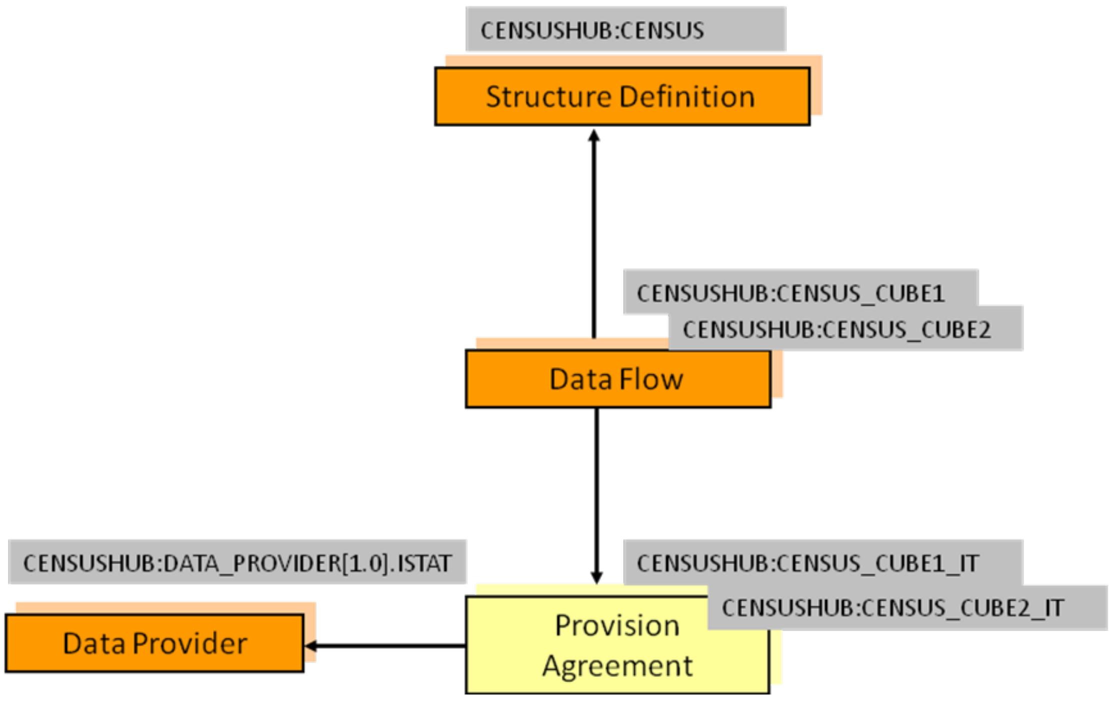


Figure 20: Example Scenario for Constraints

Constraints are declared as follows:

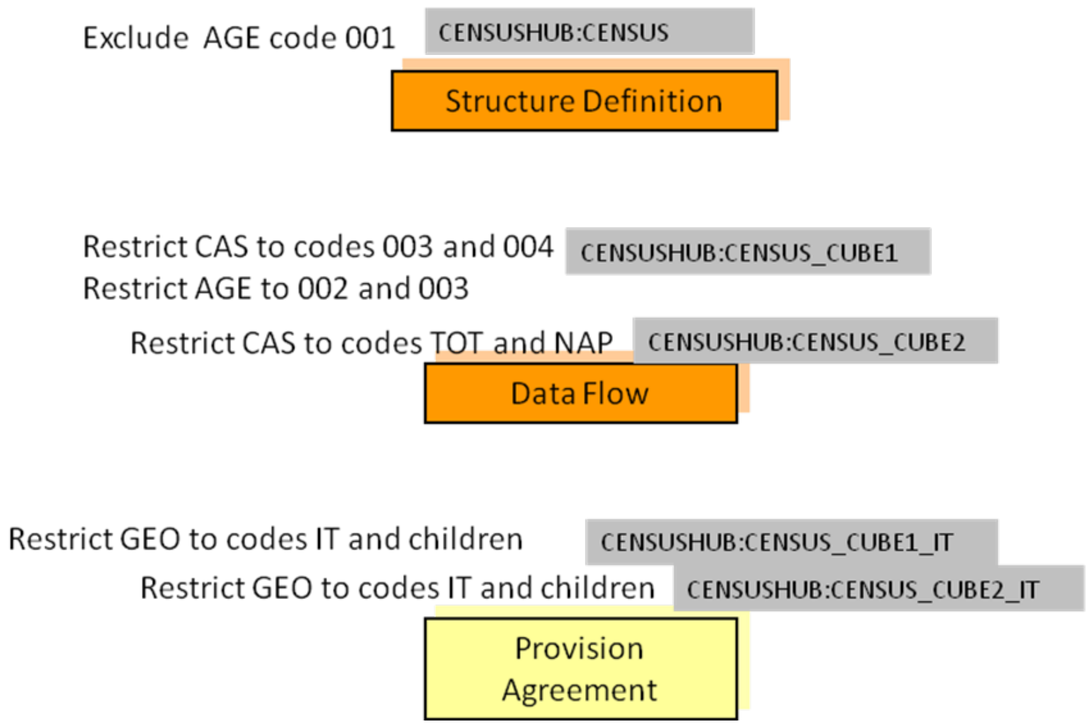

Exclude AGE code 001
CENSUSHUB:CENSUS
Structure Definition
Restrict CAS to codes 003 and 004
Restrict AGE to 002 and 003
CENSUSHUB:CENSUS_CUBE1
Restrict CAS to codes TOT and NAP
CENSUSHUB:CENSUS_CUBE2
Data Flow
Restrict GEO to codes IT and children
CENSUSHUB:CENSUS_CUBE1_IT
Restrict GEO to codes IT and children
CENSUSHUB:CENSUS_CUBE2_IT
Provision Agreement

Figure 21: Example Constraints

Notes:
AGE is constrained for the DSD and is further restricted for the Dataflow CENSUS_CUBE1.

- The same Constraint applies to both Provision Agreements.

The cascade rules elaborated above result as follows:

DSD

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

Dataflow CENSUS_CUBE1

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

Dataflow CENSUS_CUBE2

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

Provision Agreement CENSUS_CUBE1_IT

- Restricts the codes for the GEO Dimension to IT and its children.
    - Inherits the constraints from Dataflow CENSUS_CUBE1 for the AGE and CAS Dimensions.

Provision Agreement CENSUS_CUBE2_IT

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

- Inherits the constraints from Dataflow CENSUS_CUBE2 for the CAS Dimension.
- Inherits the AGE constraint applied at the level of the DSD.

The Constraints are defined as follows:

DSD Constraint

```
<str:DataConstraint agencyID="SDMX" id="DATA_CONSTRAINT" version="1.0.0-
draft" type="Allowed">
  <com:Name xml:lang="en">SDMX 3.0 Data Constraint sample</com:Name>
  <str:ConstraintAttachment>
    <str:DataStructure>urn:sdmx:org.sdmx.infomodel.datastructure.
      DataStructure=CENSUSHUB:CENSUS(1.0.0)</str:DataStructure>
  </str:ConstraintAttachment>
  <str:CubeRegion include="true">
    <!-- the ability to exclude values is illustrated - i.e., all values
valid except this one -->
    <com:KeyValue id="AGE" include="false">
      <com:Value>001</com:Value>
    </com:KeyValue>
  </str:CubeRegion>
</str:DataConstraint>
```

Dataflow Constraints

```
<str:DataConstraint agencyID="SDMX" id="DATA_CONSTRAINT_2" version="1.0.0-
draft" type="Allowed">
  <com:Name xml:lang="en">SDMX 3.0 Data Constraint sample</com:Name>
  <str:ConstraintAttachment>
    <str:Dataflow>urn:sdmx:org.sdmx.infomodel.datastructure.Dataflow=
         CENSUSHUB:CENSUS_CUBE1(1.0.0)</str:Dataflow>
  </str:ConstraintAttachment>
  <str:CubeRegion include="true">
    <com:KeyValue id="AGE" include="true">
      <com:Value>002</com:Value>
      <com:Value>003</com:Value>
    </com:KeyValue>
    <com:KeyValue id="CAS">
      <com:Value>003</com:Value>
      <com:Value>004</com:Value>
    </com:KeyValue>
  </str:CubeRegion>
</str:DataConstraint>

<str:DataConstraint agencyID="SDMX" id="DATA_CONSTRAINT_3" version="1.0.0-
draft" type="Allowed">
  <com:Name xml:lang="en">SDMX 3.0 Data Constraint sample</com:Name>
  <str:ConstraintAttachment>
    <str:Dataflow>urn:sdmx:org.sdmx.infomodel.datastructure.Dataflow=
         CENSUSHUB:CENSUS_CUBE2(1.0.0)</str:Dataflow>
  </str:ConstraintAttachment>
  <str:CubeRegion include="true">
    <com:KeyValue id="CAS" include="true">
      <com:Value>TOT</com:Value>
      <com:Value>NAP</com:Value>
    </com:KeyValue>
  </str:CubeRegion>
</str:DataConstraint>
```

Provision Agreement Constraint

```
<str:DataConstraint agencyID="SDMX" id="DATA_CONSTRAINT_4" version="1.0.0-
draft" type="Allowed">
  <com:Name xml:lang="en">SDMX 3.0 Data Constraint sample</com:Name>
  <str:ConstraintAttachment>
    <str:ProvisionAgreement>urn:sdmx:org.sdmx.infomodel.registry.
      ProvisionAgreement=CENSUSHUB:CENSUS_CUBE1_IT(1.0.0)
    </str:ProvisionAgreement>
    <str:ProvisionAgreement>urn:sdmx:org.sdmx.infomodel.registry.
      ProvisionAgreement=CENSUSHUB:CENSUS_CUBE2_IT(1.0.0)
    </str:ProvisionAgreement>
  </str:ConstraintAttachment>
  <str:CubeRegion include="true">
    <com:KeyValue id="GEO" include="true">
      <com:Value cascadeValues="true">IT</com:Value>
    </com:KeyValue>
  </str:CubeRegion>
</str:DataConstraint
```

10.4.7.2 Combination of Constraints

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

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

On the above, let’s assume the following use cases with their constraining requirements:

Use Case 1: A Constraint on allowed values for some Dimensions
R1: Allow `monthly` and `quarterly` data
R2: Allow `Mexico` for vis-à-vis country

This is expressed with the following `CubeRegion`:

FREQ	M, Q
VIS_CTY	MX

Use Case 2: A Constraint on allowed combinations for some Dimensions
R1: Allow `monthly` data for `Germany`
R2: Allow `quarterly` data for `Mexico`

This is expressed with the following `DataKeySet`:

Key1	FREQ	M

	VIS_CTY	DE
Key2	FREQ	Q
	VIS_CTY	MX

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

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

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

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).

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

DSD CubeRegion:

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

Dataflow1 CubeRegion:

FREQ	Q, A
VIS_CTY	MX, DE
OBS_STATUS	F

Dataflow2 CubeRegion:

FREQ	M
VIS_CTY	MX, DE
OBS_STATUS	A

Use case 6: A Constraint on allowed values for some Dimensions combined with allowed values for Measures

R1: Allow monthly data, for Germany, with unit euro, and measure choice is 'A'
R2: Allow quarterly data, for Mexico, with unit usd, and measure choice is 'B'

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

Use Case 7: A Constraint with wildcards for Codes and removePrefix property

For this example, we assume that the VIS_CTY representation has been prefixed with prefix 'AREA_'. In this Constraint, we need to remove the prefix.
R1: Allow monthly and quarterly data
R2: Allow vis-à-vis countries that start with M
R3: Remove the prefix 'AREA_'

This may be expressed with the following CubeRegion:

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

Use Case 8: A Constraint with multilingual support on Attributes

R1: Allow `monthly` and `quarterly` data
R2: Allow `Mexico` for vis-à-vis country
R3: Allow a comment, in English, which includes the term `adjusted` for status

This may be expressed with the following `CubeRegion`:

FREQ	M, Q
VIS_CTY	MX
COMMENT (lang='en')	%adjusted%

Use Case 9: A Constraint on allowed values for Dimensions combined with allowed values for Metadata Attributes
R1: Allow `monthly` and `quarterly` data
R2: Allow `Mexico` for vis-à-vis country
R3: Allow `John Doe` for contact

This may be expressed with the following `CubeRegion`:

FREQ	M, Q
VIS_CTY	MX
CONTACT	John Doe

11 Transforming between versions of SDMX

11.1 Scope

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

11.2 Compatibility and new DSD features

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

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

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

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”	

12 Validation and Transformation Language (VTL)

12.1 Introduction

The Validation and Transformation Language (VTL) supports the definition of Transformations, which are algorithms to calculate new data starting from already existing ones[7]. The purpose of the VTL in the SDMX context is to enable the:

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

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

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

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

[7] 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.

[8] 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

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

12.2 References to SDMX artefacts from VTL statements

12.2.1 Introduction

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

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

In any case, the aliases used in the VTL Transformations have to be mapped to the SDMX artefacts through the `VtlMappingScheme` and `VtlMapping` classes (see the section of the SDMX IM relevant to the VTL). A `VtlMapping` allows specifying the aliases to be used in the VTL Transformations, Rulesets[9] or User Defined Operators[10] to reference SDMX artefacts. A `VtlMappingScheme` is a container for zero or more `VtlMapping`.

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

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

12.2.2 References through the URN

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

The SDMX URN[11] is the concatenation of the following parts, separated by special symbols like dot, equal, asterisk, comma, and parenthesis:

- `SDMXprefix`
- `SDMX-IM-package-name`
- `class-name`
- `agency-id`

[9] See also the section "VTL-DL Rulesets" in the VTL Reference Manual.

[10] 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.

[11] 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)").

- maintainedobject-id
- maintainedobject-version
- container-object-id [12]
- object-id

The generic structure of the URN is the following:

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

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

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

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

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

The `maintainedobject-id` is the name of the maintained object which the artefact belongs to, and in case the artefact itself is maintainable[13], coincides with the name of the artefact. Therefore the `maintainedobject-id` depends on the class of the artefact:

- if the artefact is a `Dataflow`, which is a maintainable class, the `maintainedobject-id` is the `Dataflow` name (`dataflow-id`);
- if the artefact is a `Dimension`, `Measure`, `TimeDimension` or `DataAttribute`, which are not maintainable and belong to the `DataStructure` maintainable class, the `maintainedobject-id` is the name of the `DataStructure` (`dataStructure-id`) which the artefact belongs to;
- if the artefact is a `Concept`, which is not maintainable and belongs to the `ConceptScheme` maintainable class, the `maintainedobject-id` is the name of the `ConceptScheme` (`conceptScheme-id`) which the artefact belongs to;
- if the artefact is a `Codelist`, which is a maintainable class, the `maintainedobject-id` is the `Codelist` name (`codelist-id`).

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

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

[12] The container-object-id can repeat and may not be present.
[13] i.e., the artefact belongs to a maintainable class

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

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

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

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

12.2.3 Abbreviation of the URN

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

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

- The `SDMXprefix` can be omitted for all the SDMX objects, because it is a prefixed string (urn:sdmx:org), always the same for SDMX objects.
- The `SDMX-IM-package-name` can be omitted as well because it can be deduced from the `class-name` that follows it (the table of the SDMX-IM packages and classes that allows this deduction is in the SDMX 2.1 Standards - Section 5 - Registry Specifications, paragraph 6.2.3). In particular, considering the object classes of the artefacts that VTL can reference, the package is:
    - "`datastructure`" for the classes `Dataflow`, `Dimension`, `TimeDimension`, `Measure`, `DataAttribute`,
    - "`conceptscheme`" for the class `Concept`,
    - "`codelist`" for the class `Codelist`.
- The `class-name` can be omitted as it can be deduced from the VTL invocation. In particular, starting from the VTL class of the invoked artefact (e.g. dataset, component, identifier, measure, attribute, variable, valuedomain), which is known given the syntax of the invoking VTL operator[15], the SDMX class can be

[14] 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.
[15] For the syntax of the VTL operators see the VTL Reference Manual

deduced from the mapping rules between VTL and SDMX (see the section "Mapping between VTL and SDMX" hereinafter)[16].

- If the `agency-id` is not specified, it is assumed by default equal to the `agency-id` of the `TransformationScheme`, `UserDefinedOperatorScheme` or `RulesetScheme` from which the artefact is invoked. For example, the `agency-id` can be omitted if it is the same as the invoking `TransformationScheme` and cannot be omitted if the artefact comes from another agency[17]. Take also into account that, according to the VTL consistency rules, the agency of the result of a `Transformation` must be the same as its `TransformationScheme`, therefore the `agency-id` can be omitted for all the results (left part of `Transformation` statements).
- As for the `maintainedobject-id`, this is essential in some cases while in other cases it can be omitted:
  - if the referenced artefact is a `Dataflow`, which is a maintainable class, the `maintainedobject-id` is the dataflow-id and obviously cannot be omitted;
  - if the referenced artefact is a `Dimension`, `TimeDimension`, `Measure`, `DataAttribute`, which are not maintainable and belong to the `DataStructure` maintainable class, the `maintainedobject-id` is the dataStructure-id and can be omitted, given that these components are always invoked within the invocation of a `Dataflow`, whose dataStructure-id can be deduced from the SDMX structural definitions;
  - if the referenced artefact is a `Concept`, which is not maintainable and belong to the `ConceptScheme` maintainable class, the maintained object is the `conceptScheme-id` and cannot be omitted;
  - if the referenced artefact is a `Codelist`, which is a maintainable class, the `maintainedobject-id` is the `codelist-id` and obviously cannot be omitted.
- When the `maintainedobject-id` is omitted, the `maintainedobject-version` is omitted too. When the `maintainedobject-id` is not omitted and the `maintainedobject-version` is omitted, the version 1.0 is assumed by default.
  - As said, the `container-object-id` does not apply to the classes that can be referenced in VTL Transformations, therefore is not present in their URN
  - The `object-id` does not exist for the artefacts belonging to the `Dataflow`, and `Codelist` classes, while it exists and cannot be omitted for the artefacts belonging to the classes `Dimension`, `TimeDimension`, `Measure`, `DataAttribute` and `Concept`, as for them the `object-id` is the main identifier of the artefact

[16] 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.

[17] 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)

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

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

```
'urn:sdmx:org.sdmx.infomodel.datastructure.Dataflow=AG:DFR(1.0.0)' :=
'urn:sdmx:org.sdmx.infomodel.datastructure.Dataflow=AG:DF1(1.0.0)' +
'urn:sdmx:org.sdmx.infomodel.datastructure.Dataflow=AG:DF2(1.0.0)'
```

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

```
   DFR  :=  DF1 + DF2
```

The references to the `Codelists` can be simplified similarly. For example, given the non-abbreviated reference to the `Codelist` AG:CL_FREQ(1.0.0), which is[18]:

```
'urn:sdmx:org.sdmx.infomodel.codelist.Codelist=AG:CL_FREQ(1.0.0)'
```

if the `Codelist` is referenced from a `RulesetScheme` belonging to the agency AG, omitting all the optional parts, the abbreviated reference would become simply[19]:

```
   CL_FREQ
```

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

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

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

```
   SECTOR
```

For example, the Transformation for renaming the component SECTOR of the `Dataflow` DF1 into SEC can be written as[20]:

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

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

[18] Single quotes are needed because this reference is not a VTL regular name.
[19] Single quotes are not needed in this case because CL_FREQ is a VTL regular name.
[20] The result DFR(1.0.0) is be equal to DF1(1.0.0) save that the component SECTOR is called SEC

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

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

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

```
   CS1(1.0.0).SECTOR
```

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

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

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

12.2.4 User-defined alias

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

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

12.2.5 References to SDMX artefacts from VTL Rulesets

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

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

In the signature, given the mapping between VTL and SDMX better described in the following paragraphs, a reference to a VTL Value Domain becomes a reference to a SDMX `Codelist`, while a reference to a VTL Represented Variable becomes a reference to a SDMX `Concept`, assuming for it a definite representation[21].

[21] Rulesets of this kind cannot be reused when the referenced Concept has a different representation.

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

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

12.3 Mapping between SDMX and VTL artefacts

12.3.1 When the mapping occurs

The mapping methods between the VTL and SDMX object classes allow transforming a SDMX definition in a VTL one and vice-versa for the artefacts to be manipulated.
It should be remembered that VTL programs (i.e. Transformation Schemes) are represented in SDMX through the `TransformationScheme` maintainable class which is composed of `Transformations` (nameable artefacts). Each `Transformation` assigns the outcome of the evaluation of a VTL expression to a result: the input operands of the expression and the result can be SDMX artefacts.
Every time a SDMX object is referenced in a VTL Transformation as an input operand, there is the need to generate a VTL definition of the object, so that the VTL operations can take place. This can be made starting from the SDMX definition and applying a SDMX-VTL mapping method in the direction from SDMX to VTL. The possible mapping methods from SDMX to VTL are described in the following paragraphs and are conceived to allow the automatic deduction of the VTL definition of the object from the knowledge of the SDMX definition.
In the opposite direction, every time an object calculated by means of VTL must be treated as a SDMX object (for example for exchanging it through SDMX), there is the need of a SDMX definition of the object, so that the SDMX operations can take place. The SDMX definition is needed for the VTL objects for which a SDMX use is envisaged[23].

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

[22] See also the section "VTL-DL Rulesets" in the VTL Reference Manual.
[23] 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).

12.3.2 General mapping of VTL and SDMX data structures

This section makes reference to the VTL "Model for data and their structure"[24] and the correspondent SDMX "Data Structure Definition"[25].
The main type of artefact that the VTL can manipulate is the VTL Data Set, which in general is mapped to the SDMX `Dataflow`. This means that a VTL Transformation, in the SDMX context, expresses the algorithm for calculating a derived `Dataflow` starting from some already existing `Dataflows` (either collected or derived).[26]
While the VTL Transformations are defined in term of `Dataflow` definitions, they are assumed to be executed on instances of such `Dataflows`, provided at runtime to the VTL engine (the mechanism for identifying the instances to be processed are not part of the VTL specifications and depend on the implementation of the VTL-based systems). As already said, the SDMX `Datasets` are instances of SDMX `Dataflows`, therefore a VTL Transformation defined on some SDMX `Dataflows` can be applied on some corresponding SDMX `Datasets`.

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

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

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

12.3.3 Mapping from SDMX to VTL data structures

12.3.3.1 Basic Mapping

The main mapping method from SDMX to VTL is called **Basic** mapping. This is considered as the default mapping method and is applied unless a different method is specified through the `VtlMappingScheme` and `VtlDataflowMapping` classes.
When transforming **from SDMX to VTL**, this method consists in leaving the components unchanged and maintaining their names and roles, according to the following table:

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

[24] See the VTL 2.0 User Manual
[25] See the SDMX Standards Section 2 – Information Model
[26] 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

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[27] generates one VTL data point.

12.3.3.2 Pivot Mapping

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.

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[28].

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).

[27] Here an SDMX observation is meant to correspond to one combination of values of the DimensionComponents.

[28] 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 Cj of the MeasureDimension …" it means "each possible combination of values (Cj, Dk, Ew) of the joint MeasureDimension Z=(C,D,E)".

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

- 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:

SDMX	VTL
`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

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.

At observation / data point level, calling Cj (j=1, … n) the jth Code of the MeasureDimension:

- 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 Cj of the SDMX MeasureDimension;
- 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.
- The value of the Measure of the SDMX observation belonging to the set above and having MeasureDimension=Cj becomes the value of the VTL Measure Cj
- 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=Cj becomes the value of the VTL Attribute DA_Cj

12.3.3.3 From SDMX DataAttributes to VTL Measures

- 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.

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.

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.

12.3.4 Mapping from VTL to SDMX data structures

12.3.4.1 Basic Mapping

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

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

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

Mapping table:

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

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

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

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

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

12.3.4.2 Unpivot Mapping

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

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

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

The **unpivot** mapping behaves like follows:

- like in the basic mapping, a VTL (simple) identifier becomes a SDMX `Dimension` and a VTL (time) identifier becomes a SDMX `TimeDimension` (as said, a measure identifier cannot exist in multi-measure VTL structures);
- a `MeasureDimension` component called "measure_name" is added to the SDMX `DataStructure`;
- a `Measure` component called "obs_value" is added to the SDMX `DataStructure`;
- each VTL Measure is mapped to a `Code` of the SDMX `MeasureDimension` having the same name as the VTL Measure (therefore all the VTL Measure Components do not originate `Components` in the SDMX `DataStructure`);
- a VTL Attribute becomes a SDMX `DataAttribute` having `AttributeRelationship` referred to all the SDMX `DimensionComponents` including the `TimeDimension` and except the `MeasureDimension`.

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`

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^{th} VTL Measure (e.g. "Cj") becomes the `Code` of the SDMX `MeasureDimension` of the j^{th} observation of the set;
- the value of the j^{th} VTL Measure becomes the value of the SDMX `Measure` of the j^{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).

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

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

12.3.4.3 From VTL Measures to SDMX Data Attributes

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

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

The mapping table is the following:

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

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

12.3.5 Declaration of the mapping methods between data structures

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

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

It is possible to specify the `toVtlMappingMethod` and `fromVtlMappingMethod` also for the conventional `dataflow` called "generic_dataflow": in this case the specified mapping methods are intended to become the default ones, overriding the "Basic" methods. In turn, the `toVtlMappingMethod` and `fromVtlMappingMethod` declared for a specific `Dataflow` are intended to override the default ones for such a `Dataflow`.

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

12.3.6 Mapping dataflow subsets to distinct VTL Data Sets

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

As a matter of fact, in some cases it can be useful to define VTL operations involving definite parts of a SDMX `Dataflow` instead than the whole.[29]

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

Given a SDMX `Dataflow` and some predefined `Dimensions` of its `DataStructure`, it is allowed to map the subsets of observations that have the same combination of values for such `Dimensions` to correspondent VTL datasets.
For example, assuming that the SDMX `Dataflow` DF1(1.0.0) has the `Dimensions` INDICATOR, TIME_PERIOD and COUNTRY, and that the user declares the `Dimensions` INDICATOR and COUNTRY as basis for the mapping (i.e. the mapping dimensions): the observations that have the same values for INDICATOR and COUNTRY would be mapped to the same VTL dataset (and vice-versa).
In practice, this kind mapping is obtained like follows:

- For a given SDMX `Dataflow`, the user (VTL definer) declares the `DimensionComponents` on which the mapping will be based, in a given order.[31] Following the example above, imagine that the user declares the `Dimensions` INDICATOR and COUNTRY.

[29] 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.

[30] 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.

[31] 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.

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

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

```
'DF(1.0.0)/INDICATORvalue.COUNTRYvalue'
```

Where DF(1.0.0) is the `Dataflow` and *INDICATORvalue* and *COUNTRYvalue* are placeholders for one value of the INDICATOR and COUNTRY dimensions.
Instead the specific name of one of these VTL datasets would be:

```
‘DF(1.0.0)/POPULATION.USA’
```

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

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

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

[32] 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`.

[33] 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.

[34] 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.

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

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

In order to obtain the data structure of these VTL Data Sets from the SDMX one, it is assumed that the SDMX `DimensionComponents` on which the mapping is based are dropped, i.e. not maintained in the VTL data structure; this is possible because their values are fixed for each one of the invoked VTL Data Sets[35]. After that, the mapping method from SDMX to VTL specified for the `Dataflow` DF1(1.0.0) is applied (i.e. basic, pivot …).

In the example above, for all the datasets of the kind 'DF1(1.0.0)/*INDICATORvalue*.*COUNTRYvalue*', the dimensions INDICATOR and COUNTRY would be dropped so that the data structure of all the resulting VTL Data Sets would have the identifier TIME_PERIOD only.
It should be noted that the desired VTL Data Sets (i.e. of the kind 'DF1(1.0.0)/ *INDICATORvalue*.*COUNTRYvalue*') can be obtained also by applying the VTL operator "**sub**" (subspace) to the `Dataflow` DF1(1.0.0), like in the following VTL expression:

```
'DF1(1.0.0)/POPULATION.USA' :=
DF1(1.0.0) [ sub  INDICATOR="POPULATION", COUNTRY="USA" ];

'DF1(1.0.0)/POPULATION.CANADA' :=
DF1(1.0.0) [ sub  INDICATOR="POPULATION", COUNTRY="CANADA" ];

… … …
```

In fact the VTL operator "sub" has exactly the same behaviour. Therefore, mapping different parts of a SDMX `Dataflow` to different VTL Data Sets in the direction from SDMX to VTL through the ordered concatenation notation is equivalent to a proper use of the operator "**sub**" on such a `Dataflow`. [36]

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

[35] 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).

[36] 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.

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

This is equivalent to the application of the VTL “sub” operator only to the identifier *INDICATOR*:

```
‘DF1(1.0.0)/POPULATION.’ :=
DF1(1.0.0) [ sub  INDICATOR=“POPULATION” ];
```

Therefore the VTL Data Set ‘DF1(1.0.0)/POPULATION.’ would have the identifiers COUNTRY and TIME_PERIOD.
Heterogeneous invocations of the same `Dataflow` are allowed, i.e. omitting different `Dimensions` in different invocations.
Let us now analyse <u>the mapping direction from VTL to SDMX</u>.
In this situation, distinct parts of a SDMX `Dataflow` are calculated as distinct VTL datasets, under the constraint that they must have the same VTL data structure.

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

- each part is calculated as a VTL derived Data Set, result of a dedicated VTL Transformation; [37]
- the data structure of all these VTL Data Sets has the TIME_PERIOD identifier and does not have the INDICATOR and COUNTRY identifiers.[38]

Under these hypothesis, such derived VTL Data Sets can be mapped to DF2(1.0.0) by declaring the `DimensionComponents` INDICATOR and COUNTRY as mapping dimensions[39].

The corresponding VTL Transformations, assuming that the result needs to be persistent, would be of this kind: [40]

```
        ‘DF2(1.0.0)/INDICATORvalue.COUNTRYvalue’  <-  expression
```

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

```
        ‘DF2(1.0.0)/GDPPERCAPITA.USA’       <-   expression11;
        ‘DF2(1.0.0)/GDPPERCAPITA.CANADA’    <-   expression12;
```

[37] 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.

[38] This is possible as each VTL dataset corresponds to one particular combination of values of INDICATOR and COUNTRY.

[39] The mapping dimensions are defined as `FromVtlSpaceKeys` of the `FromVtlSuperSpace` of the `VtlDataflowMapping` relevant to DF2(1.0).

[40] the symbol of the VTL persistent assignment is used (<-)

```
             …    …    …
             ‘DF2(1.0.0)/POPGROWTH.USA’          <-   expression21;
             ‘DF2(1.0.0)/POPGROWTH.CANADA’     <-   expression22;
             …    …    …
```

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

VTL dataset	INDICATOR value	COUNTRY value
‘DF2(1.0.0)/GDPPERCAPITA.USA’	GDPPERCAPITA	USA
‘DF2(1.0.0)/GDPPERCAPITA.CANADA’	GDPPERCAPITA	CANADA
… … …		
‘DF2(1.0.0)/POPGROWTH.USA’	POPGROWTH	USA
‘DF2(1.0.0)/POPGROWTH.CANADA’	POPGROWTH	CANADA
… … …		

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

```
DF2bis_GDPPERCAPITA_USA      :=   ‘DF2(1.0.0)/GDPPERCAPITA.USA’
                            [calc  identifier INDICATOR := ”GDPPERCAPITA”,
                                 identifier  COUNTRY := ”USA”];

DF2bis_GDPPERCAPITA_CANADA :=   ‘DF2(1.0.0)/GDPPERCAPITA.CANADA’
                            [calc   identifier INDICATOR:=”GDPPERCAPITA”,
                                 identifier COUNTRY:=”CANADA”];
   …    …    …
DF2bis_POPGROWTH_USA      :=  ‘DF2(1.0.0)/POPGROWTH.USA’
                            [calc  identifier INDICATOR := ”POPGROWTH”,
                                 identifier  COUNTRY := ”USA”];
DF2bis_POPGROWTH_CANADA’  :=  ‘DF2(1.0.0)/POPGROWTH.CANADA’
                            [calc  identifier INDICATOR := ”POPGROWTH”,
                                 identifier  COUNTRY := ”CANADA”];
   …    …    …
DF2(1.0)     <-        UNION          (DF2bis_GDPPERCAPITA_USA’,
                                    DF2bis_GDPPERCAPITA_CANADA’,
                                    … ,
                                    DF2bis_POPGROWTH_USA’,
                                    DF2bis_POPGROWTH_CANADA’
                                    …);
```

In other words, starting from the datasets explicitly calculated through VTL (in the example ‘DF2(1.0)/GDPPERCAPITA.USA’ and so on), the first step consists in calculating other (non-persistent) VTL datasets (in the example DF2bis_GDPPERCAPITA_USA and so on) by adding the identifiers INDICATOR and

COUNTRY with the desired values (*INDICATORvalue* and *COUNTRYvalue)*. Finally, all these non-persistent Data Sets are united and give the final result DF2(1.0)[41], which can be mapped one-to-one to the homonymous SDMX `Dataflow` having the dimension components TIME_PERIOD, INDICATOR and COUNTRY.

Therefore, mapping different VTL datasets having the same data structure to different parts of a SDMX `Dataflow`, i.e. in the direction from VTL to SDMX, through the ordered concatenation notation is equivalent to a proper use of the operators "calc" and "union" on such datasets. [42]

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

12.3.7 Mapping variables and value domains between VTL and SDMX

With reference to the VTL "model for Variables and Value domains", the following additional mappings have to be considered:

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

[41] The result is persistent in this example but it can be also non persistent if needed.

[42] 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.

[43] Through SDMX `Constraints`, it is possible to specify the values that a `Component` of a `Dataflow` can assume.

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

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[44], while the SDMX `Concepts` can have different `Representations` in different `DataStructures`.[45] 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

[44] 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.

[45] 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`.

Transformations to ensure that the VTL expressions are consistent with the actual representations of the correspondent SDMX `Concepts`.
It remains up to the SDMX-VTL definer also the assurance of the consistency between a VTL Ruleset defined on Variables and the SDMX `Components` on which the Ruleset is applied. In fact, a VTL Ruleset is expressed by means of the values of the Variables (i.e. SDMX `Concepts`), i.e. assuming definite representations for them (e.g. ISO-alpha-3 for country). If the Ruleset is applied to SDMX Components that have the same name of the Concept they refer to but different representations (e.g. ISO-alpha-2 for country), the Ruleset cannot work properly.

12.4 Mapping between SDMX and VTL Data Types

12.4.1 VTL Data types

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

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

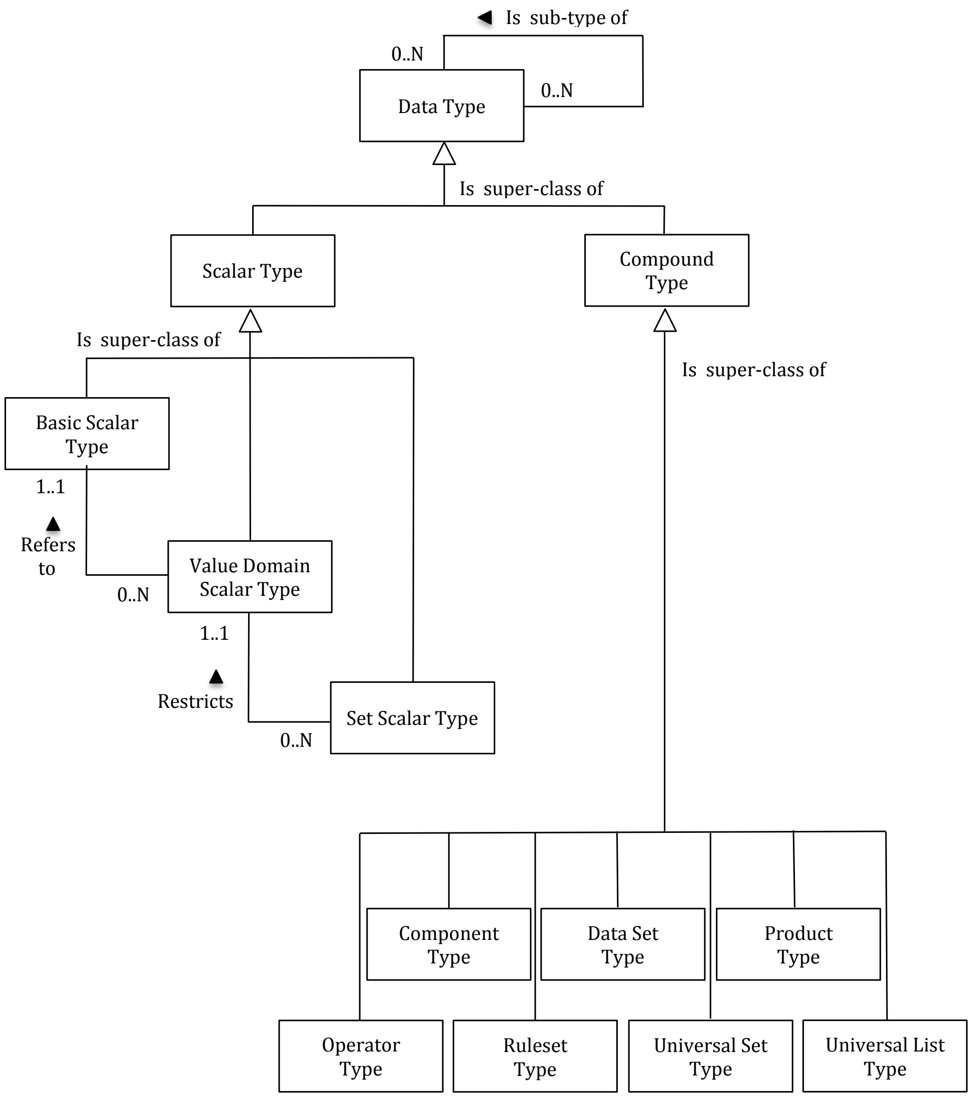


Figure 22 – VTL Data Types

The VTL scalar types are in turn subdivided in basic scalar types, which are elementary (not defined in term of other data types) and Value Domain and Set scalar types, which are defined in terms of the basic scalar types.
The VTL basic scalar types are listed below and follow a hierarchical structure in terms of supersets/subsets (e.g. "scalar" is the superset of all the basic scalar types):

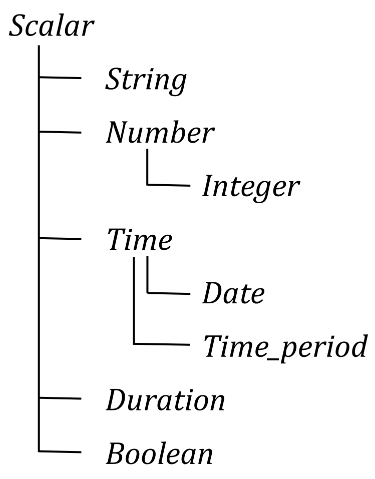


Figure 23 – VTL Basic Scalar Types

12.4.2 VTL basic scalar types and SDMX data types

The VTL assumes that a basic scalar type has a unique internal representation and can have more external representations.
The internal representation is the format used within a VTL system to represent (and process) all the scalar values of a certain type. In principle, this format is hidden and not necessarily known by users. The external representations are instead the external formats of the values of a certain basic scalar type, i.e. the formats known by the users.
For example, the internal representation of the dates can be an integer counting the days since a predefined date (e.g. from 01/01/4713 BC up to 31/12/5874897 AD like in Postgres) while two possible external representations are the formats YYYY-MM-GG and MM-GG-YYYY (e.g. respectively 2010-12-31 and 12-31-2010).
The internal representation is the reference format that allows VTL to operate on more values of the same type (for example on more dates) even if such values have different external formats: these values are all converted to the unique internal representation so that they can be composed together (e.g. to find the more recent date, to find the time span between these dates and so on).
The VTL assumes that a unique internal representation exists for each basic scalar type but does not prescribe any particular format for it, leaving the VTL systems free to using they preferred or already existing internal format. By consequence, in VTL the basic scalar types are abstractions not associated to a specific format.
SDMX data types are conceived instead to support the data exchange, therefore they do have a format, which is known by the users and correspond, in VTL terms, to external representations. Therefore, for each VTL basic scalar type there can be more SDMX data types (the latter are explained in the section "General Notes for Implementers" of this document and are actually much more numerous than the former).

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

The conversion from SDMX to VTL happens when an SDMX artefact acts as inputs of a VTL Transformation. As already said, in fact, at compile time the VTL needs to know the VTL type of the operands in order to check their compliance with the VTL operators and at runtime it must convert the values from their external (SDMX) representations to the corresponding internal (VTL) ones.

The opposite conversion, i.e. from VTL to SDMX, happens when a VTL result, i.e. a VTL Data Set output of a Transformation, must become a SDMX artefact (or part of it). The values of the VTL result must be converted into the desired (SDMX) external representations (data types) of the SDMX artefact.

12.4.3 Mapping SDMX data types to VTL basic scalar types

The following table describes the default mapping for converting from the SDMX data 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; speicifies 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 25th; used to specify reporting year start day)	string
Day (---DD; specifies a day independent of a month or year; e.g. the 15th 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 KeyValues IdentifiableReference DataSetReference	Metadata type – not applicable Metadata type – not applicable Metadata type – not applicable Metadata type – not applicable

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

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

12.4.4 Mapping VTL basic scalar types to SDMX data types

The following table describes the default conversion from the VTL basic scalar types 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

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

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

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[46].

12.4.5 Null Values

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`.

[46] The representation given in the DSD should obviously be compatible with the VTL data type.

12.4.6 Format of the literals used in VTL Transformations

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

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

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

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

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

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

13 Structure Mapping

13.1 Introduction

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

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

An input series maps to:

a. Exactly one output series; or

b. Multiple output series with different Series Keys, but the same observation values; or

c. Zero output series where no source rule matches the input Component values.

Typical use cases include:

- Transforming received data into a common internal structure;
- Transforming reported data into the data collector's preferred structure;
- Transforming unidimensional datasets[47] to multi-dimensional; and
- Transforming internal datasets with a complex structure to a simpler structure with fewer dimensions suitable for dissemination.

13.2 1-1 structure maps

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

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

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

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

[47] 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

13.3 N-n structure maps

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

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

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"

13.4 Ambiguous mapping rules

A structure map is ambiguous if the rules result in a dataset containing multiple series with the same Series Key.

A simple example mapping a source dataset with a single dimension to one with multiple dimensions is shown below:

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

The above behaviour can be okay if the series XMAN_Z_21 contains observations for different periods of time then the series XMAN_Z_34. If however both series contain observations for the same point in time, the output for this mapping will be two observations with the same series key, for the same period in time.

13.5 Representation maps

Representation Maps replace the SDMX 2.1 Codelist Maps and are used describe explicit mappings between source and target Component values.

The source and target of a Representation Map can reference any of the following:

a. Codelist
b. Free Text (restricted by type, e.g String, Integer, Boolean)
c. Valuelist

A Representation Map mapping ISO 2-character to ISO 3-character Codelists would take the following form:

CL_ISO_ALPHA2	CL_ISO_ALPHA3
AF	AFG
AL	ALB
DZ	DZA
AS	ASM
AD	AND
etc…	

A Representation Map mapping free text country names to an ISO 2-character Codelist could be similarly described:

Text	CL_ISO_ALPHA2
"Germany"	DE
"France"	FR
"United Kingdom"	GB
"Great Britain"	GB
"Ireland"	IE
"Eire"	IE
etc…	

Valuelists, introduced in SDMX 3.0, are equivalent to Codelists but allow the maintenance of non-SDMX identifiers. Importantly, their IDs do not need to conform to IDType, but as a consequence are not Identifiable.

When used in Representation Maps, Valuelists allow Non-SDMX identifiers containing characters like £, $, % to be mapped to Code IDs, or Codes mapped to non-SDMX identifiers.

In common with Codelists, each item in a Valuelist has a multilingual name giving it a human-readable label and an optional description. For example:

Value	Locale	Name
$	en	United States Dollar
%	En	Percentage
	fr	Pourcentage

Other characteristics of Representation Maps:

- Support the mapping of multiple source Component values to multiple Target Component values as described in section 13.3 on n-to-n mappings; this covers also the case of mapping an Attribute with an array representation to map combinations of values to a single target value;

- Allow source or target mappings for an Item to be optional allowing rules such as 'A maps to nothing' or 'nothing maps to A'; and

- Support for mapping rules where regular expressions or substrings are used to match source Component values. Refer to section 13.6 for more on this topic.

13.6 Regular expression and substring rules

It is common for classifications to contain meanings within the identifier, for example the code Id 'XULADS' may refer to a particular seasonality because it starts with the letters XU.

With SDMX 2.1 each code that starts with XU had to be individually mapped to the same seasonality, and additional mappings added when new Codes were added to the Codelists. This led to many hundreds or thousands of mappings which can be more efficiently summarised in a single conceptual rule:

If starts with 'XU' map to 'Y'

These rules are described using either regular expressions, or substrings for simpler use cases.

13.6.1 Regular expressions

Regular expression mapping rules are defined in the Representation Map.

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

Like all mapping rules, the output is either a Code, a Value or free text depending on the representation of the Component in the target Data Structure Definition.

If the regular expression contains capture groups, these can be used in the definition of the output value, by specifying ***n*** as an output value where ***n*** is the number of the capture group starting from 1. For example

Regex	Target output	Example Input	Example Output
`([0-9]{4})[0-9]([0-9]{1})`	`\1-Q\2`	200933	2009-Q3

As regular expression rules can be used as a general catch-all if nothing else matches, the ordering of the rules is important. Rules should be tested starting with the highest priority, moving down the list until a match is found.

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

The input 'A' matches both the first and the last rule, but the first takes precedence having the higher priority. The output is OUT_A.

The input 'G' matches on the last rule which is used as a catch-all or default in this example.

13.6.2 Substrings

Substrings provide an alternative to regular expressions where the required section of an input value can be described using the number of the starting character, and the length of the substring in characters. The first character is at position 1.

For instance:

Input String	Start	Length	Output
ABC_DEF_XYZ	5	3	DEF
XULADS	1	2	XU

Sub-strings can therefore be used for the conceptual rule *If starts with 'XU' map to Y* as shown in the following example:

Start	Length	Source	Target
1	2	XU	Y

13.7 Mapping non-SDMX time formats to SDMX formats

Structure mapping allows non-SDMX compliant time values in source datasets to be mapped to an SDMX compliant time format.

Two types of time input are defined:

a. **Pattern based dates** – a string which can be described using a notation like dd/mm/yyyy or is represented as the number of periods since a point in time, for example: 2010M001 (first month in 2010), or 2014D123 (123rd day in 2014); and

b. **Numerical based datetime** – a number specifying the elapsed periods since a fixed point in time, for example Unix Time is measured by the number of milliseconds since 1970.

The output of a time-based mapping is derived from the output Frequency, which is either explicitly stated in the mapping or defined as the value output by a specific Dimension or Attribute in the output mapping. If the output frequency is unknown or if the SDMX format is not desired, then additional rules can be provided to specify the 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

In the case where the input frequency is lower than the output frequency, the mapping defaults to end of period, but can be explicitly set to start, end or mid-period.

There are two important points to note:

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.
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.

13.7.1 Pattern based dates

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.

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[48]. An indicative list of examples is presented in the following table:

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

[48] 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

Examples

`22/06/1981` would be described as `dd/MM/YYYY`, with locale `en-GB`
`2008-mars-12` would be described as `YYYY-MMM-DD`, with locale `fr-FR`
`22 July 1981` would be described as `dd MMMM YYYY`, with locale `en-US`
`22 Jul 1981` would be described as `dd MMM YYYY`
`2010 D62` would be described as `YYYYDnn` (day 62 of the year 2010)

The following pattern letters are defined (all other characters from 'A' to 'Z' and from 'a' 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[49])	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

[49] 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

The model is illustrated below:

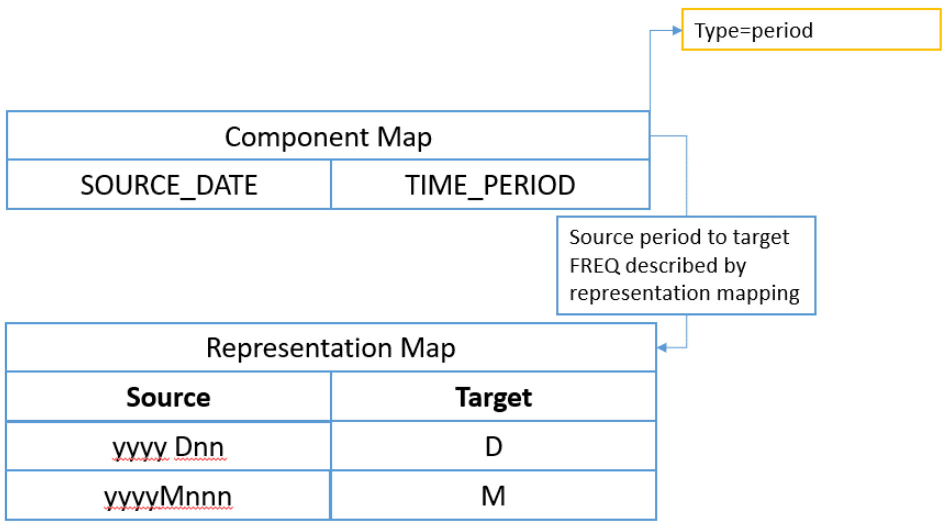


Figure 24 showing the component map mapping the SOURCE_DATE Dimension to the TIME_PERIOD dimension with the additional information on the component map to describe the time format

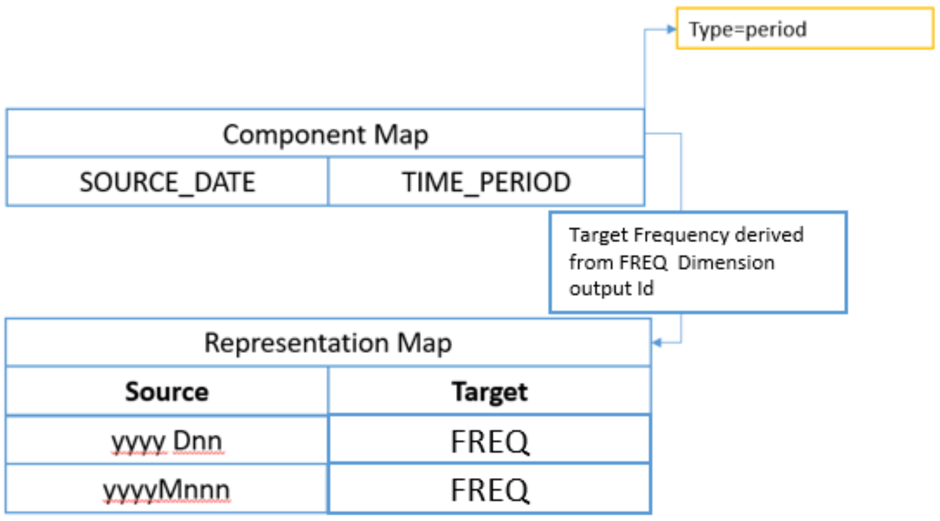


Figure 25 showing an input date format, whose output frequency is derived from the output value of the FREQ Dimension

13.7.2 Numerical based datetime

Where the source datetime input is purely numerical, the mapping rules are defined by the **Base** as a valid SDMX Time Period, and the **Period** which must take one of the following enumerated values:

- day
- second
- millisecond
- microsecond
- 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

The example above illustrates numerical based datetime mapping rules for two commonly used time standards.

The model is illustrated below:

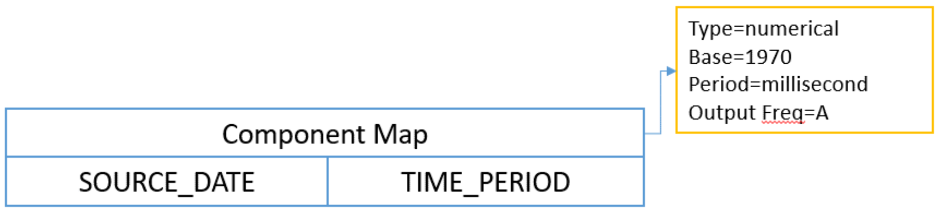


Figure 26 showing the component map mapping the SOURCE_DATE Dimension to the TIME_PERIOD Dimension with the additional information on the component map to describe the numerical datetime system in use

13.7.3 Mapping more complex time inputs

VTL should be used for more complex time inputs that cannot be interpreted using the pattern based on numerical methods.

13.8 Using TIME_PERIOD in mapping rules

The source TIME_PERIOD Dimension can be used in conjunction with other input Dimensions to create discrete mapping rules where the output is conditional on the time period value.

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**

In the example above, OBS_CONF is an Observation Attribute.

13.9 Time span mapping rules using validity periods

Creating discrete mapping rules for each TIME_PERIOD is impractical where rules need to cover a specific span of time regardless of frequency, and for high-frequency data.

Instead, an optional validity period can be set for each mapping.

By specifying validity periods, the example from Section 13.8 can be re-written using 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

In Rule 1, start period resolves to the start of the 2007 period (2007-01-01T00:00:00), and the end period resolves to the very end of 2009 (2009-12-31T23:59:59). The rule will hold true regardless of the input data frequency. Any observations reporting data for the Indicator XULADS that fall into that time range will have an OBS_CONF value of F.

In Rule 2, no end period is specified so remains in effect from the start of the period (2010-01-01T00:00:00) until the end of time. Any observations reporting data for the Indicator XULADS that fall into that time range will have an OBS_CONF value of C.

13.10 Mapping examples

13.10.1 Many to one mapping (N-1)

Source	Map To

FREQ="A"
ADJUSTMENT="N"
REF_AREA="PL"
COUNTERPART_AREA="W0"
REF_SECTOR="S1"
COUNTERPART_SECTOR="S1"
ACCOUNTING_ENTRY="B"
STO="B5G"
	FREQ="A"
	REF_AREA="PL"
	COUNTERPART_AREA="W0"
	INDICATOR="IND_ABC"

The bold Dimensions map from source to target verbatim. The mapping simply specifies:
FREQ => FREQ
REF_AREA=> REF_AREA
COUNTERPART_AREA=> COUNTERPART _AREA

No Representation Mapping is required. The source value simply copies across unmodified.

The remaining Dimensions all map to the Indicator Dimension. This is an example of many Dimensions mapping to one Dimension. In this case a Representation Mapping is required, and the mapping first describes the input 'partial key' and how this maps to the target indicator:

N:S1:S1:B:B5G => IND_ABC

Where the key sequence is based on the order specified in the mapping (i.e ADJUSTMENT, REF_SECTOR, etc will result in the first value N being taken from ADJUSTMENT as this was the first item in the source Dimension list.

Note: The key order is NOT based on the Dimension order of the DSD, as the mapping needs to be resilient to the DSD changing.

13.10.2 Mapping other data types to Code Id

In the case where the incoming data type is not a string and not a code identifier i.e. the source Dimension is of type Integer and the target is Codelist. This is supported by the RepresentationMap. The RepresentationMap source can reference a Codelist, Valuelist, or be free text, the free text can include regular expressions.
The following representation mapping can be used to explicitly map each age to an output code.

Source Input Free Text	Desired Output Code Id
0	A
1	A
2	A
3	B
4	B

If this mapping takes advantage of regular expressions it can be expressed in two rules:

Regular Expression	Desired Output
[0-2]	A
[3-4]	B

13.10.3 Observation Attributes for Time Period

This use case is where a specific observation for a specific time period has an attribute value.

Input INDICATOR	Input TIME_PERIOD	Output OBS_CONF
XULADS	2008	C
XULADS	2009	C
XULADS	2010	C

Or using a validity period on the Representation Mapping:

Input INDICATOR	Valid From/ Valid To	Output OBS_CONF
XULADS	2008/2010	C

13.10.4 Time mapping

This use case is to create a time period from an input that does not respect SDMX Time Formats.
The Component Mapping from SYS_TIME to TIME_PERIOD specifies itself as a time mapping with the following details:

Source Value	Source Mapping	Target Frequency	Output
18/07/1981	dd/MM/yyyy	A	1981

When the target frequency is based on another target Dimension value, in this example the value of the FREQ Dimension in the target DSD.

Source Value	Source Mapping	Target Frequency Dimension	Output
18/07/1981	dd/MM/yyyy	FREQ	1981-07-18 (when FREQ=D)

When the source is a numerical format

Source Value	Start Period	Interval	Target FREQ	Output
1589808220	1970	millisecond	M	2020-05

When the source frequency is lower than the target frequency additional information can be provided for resolve to start of period, end of period, or mid period, as shown in the following example:

Source Value	Source Mapping	Target Frequency Dimension	Output
1981	yyyy	D – End of Period	1981-12-31

When the start of year is April 1st the Structure Map has YearStart=04-01:

Source Value	Source Mapping	Target Frequency Dimension	Output
1981	yyyy	D – End of Period	1982-03-31

14 ANNEX Semantic Versioning

14.1 Introduction to Semantic Versioning

In the world of versioned data modelling exists a dreaded place called "dependency hell." The bigger your data model through organisational, national or international harmonisation grows and the more artefacts you integrate into your modelling, the more likely you are to find yourself, one day, in this pit of despair.

In systems with many dependencies, releasing new artefact versions can quickly become a nightmare. If the dependency specifications are too tight, you are in danger of version lock (the inability to upgrade an artefact without having to release new versions of every dependent artefact). If dependencies are specified too loosely, you will inevitably be bitten by version promiscuity (assuming compatibility with more future versions than is reasonable). Dependency hell is where you are when version lock and/or version promiscuity prevent you from easily and safely moving your data modelling forward.

As a very successful solution to the similar problem in software development, "Semantic Versioning" semver.org proposes a simple set of rules and requirements that dictate how version numbers are assigned and incremented. These rules make also perfect sense in the world of versioned data modelling and help to solve the "dependency hell" encountered with previous versions of SDMX. SDMX 3.0 applies thus the Semantic Versioning rules on all versioned SDMX artefacts. Once you release a versioned SDMX artefact, you communicate changes to it with specific increments to your version number.

This SDMX 3.0(.0) specification inherits the original semver.org 2.0.0 wording (license: Creative Commons - CC BY 3.0) and applies it to versioned SDMX structural artefacts. Under this scheme, version numbers and the way they change convey meaning about the underlying data structures and what has been modified from one version to the next.

14.2 Semantic Versioning Specification for SDMX 3.0(.0)

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

In the following, "versioned" artefacts are understood to be semantically versioned SDMX structural artefacts, and X, Y, Z and EXT are understood as placeholders for the version parts `MAJOR`, `MINOR`, `PATCH`, and `EXTENSION`, as defined in chapter 4.3.

The following rules apply to versioned artefacts:

- 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.

- Once an SDMX artefact with an X.Y.Z version has been shared externally or publicly released, the contents of that version MUST NOT be modified. That artefact version is considered stable. Any modifications MUST be released as a new version.
- `MAJOR` version zero (0.y.z) is for initial modelling. Anything MAY change at any time. The externally released or public artefact SHOULD NOT be considered stable.
- Version 1.0.0 defines the first stable artefact. The way in which the version number is incremented after this release is dependent on how this externally released or public artefact changes.
- `PATCH` version Z (x.y.Z | x > 0) MUST be incremented if only backwards compatible property changes are introduced. A property change is defined as an internal change that does not affect the relationship to other artefacts. These are changes in the artefact's or artefact element's names, descriptions and annotations that MUST NOT alter their meaning.
- `MINOR` version Y (x.Y.z | x > 0) MUST be incremented if a new, backwards compatible element is introduced to a stable artefact. These are additional items in ItemScheme artefacts. It MAY be incremented if substantial new information is introduced within the artefact's properties. It MAY include `PATCH` level changes. `PATCH` version MUST be reset to 0 when `MINOR` version is incremented.
- `MAJOR` version X (X.y.z | X > 0) MUST be incremented if any backwards incompatible changes are introduced to a stable artefact. These often relate to deletions of items in ItemSchemes or to backwards incompatibility introduced due to changes in references to other artefacts. A `MAJOR` version change MAY also include `MINOR` and `PATCH` level changes. `PATCH` and `MINOR` version MUST be reset to 0 when `MAJOR` version is incremented.
- A mutable version, e.g. used for externally released or public drafts or as pre-release, MUST be denoted by appending an `EXTENSION` that consists of a hyphen and a series of dot separated identifiers immediately following the `PATCH` version (x.y.z-EXT). Identifiers MUST comprise only ASCII alphanumerics and hyphen [0-9A-Za-z-]. Identifiers MUST NOT be empty. Numeric identifiers MUST NOT include leading zeroes. However, to foster harmonisation and general comprehension it is generally recommended to use the standard `EXTENSION` "**-draft**". Extended versions have a lower precedence than the associated stable version. An extended version indicates that the version is unstable and it might not satisfy the intended compatibility requirements as denoted by its associated stable version. When making changes to an SDMX artefact with an extended version number then one is not required to increment the version if those changes are kept within the allowed scope of the version increment from the previous version (if that existed), otherwise also here the before mentioned version increment rules for X.Y.Z apply. Concretely, a version X.0.0-EXT will allow for any changes, a version X.Y.0-EXT will allow only for `MINOR` changes and a version X.Y.Z-EXT will allow only for any `PATCH` changes, as defined above. `EXTENSION` examples: 1.0.0-draft, 1.0.0-draft.1, 1.0.0-0.3.7, 1.0.0-x.7.z.92.

- 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.
- The reasons for version changes MAY be documented in brief form in an artefact's annotation of type "CHANGELOG".

14.3 Backus–Naur Form Grammar for Valid SDMX 3.0(.0) Semantic Versions

```
<valid semver> ::= <version core>
                 | <version core> "-" <extension>

<version core> ::= <major> "." <minor> "." <patch>

<major> ::= <numeric identifier>

<minor> ::= <numeric identifier>

<patch> ::= <numeric identifier>

<extension> ::= <dot-separated extension identifiers>

<dot-separated extension identifiers> ::= <extension identifier>
                                        | <extension identifier> "." <dot-
separated extension identifiers>

<extension identifier> ::= <alphanumeric identifier>
                         | <numeric identifier>

<alphanumeric identifier> ::= <non-digit>
                            | <non-digit> <identifier characters>
                            | <identifier characters> <non-digit>
                            | <identifier characters> <non-digit> <identifier
characters>

<numeric identifier> ::= "0"
                       | <positive digit>
                       | <positive digit> <digits>

<identifier characters> ::= <identifier character>
                          | <identifier character> <identifier characters>
```

```
<identifier character> ::= <digit>
                         | <non-digit>

<non-digit> ::= <letter>
              | "-"

<digits> ::= <digit>
           | <digit> <digits>

<digit> ::= "0"
          | <positive digit>

<positive digit> ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

<letter> ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J"
           | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T"
           | "U" | "V" | "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "d"
           | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n"
           | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x"
           | "y" | "z"
```

14.4 Dependency Management in SDMX 3.0(.0):

`MAJOR`, `MINOR` or `PATCH` version parts in SDMX 3.0 artefact references CAN be wildcarded using "+" as extension:

- `X+.Y.Z` means the currently latest available version >= `X.Y.Z`
  - Example: `"2+.3.1"` means the currently latest available version >= `"2.3.1"` (even if not backwards compatible)
  - Typical use case: references in SDMX Categorisations
- `X.Y+.Z` means the currently latest available backwards compatible version >= `X.Y.Z`
  - Example: `"2.3+.1"` means the currently latest available version >= `"2.3.1"` and < `"3.0.0"` (all backwards compatible versions >= `"2.3.1"`)
  - Typical use case: references in SDMX DSD
- `X.Y.Z+` means the currently latest available forwards and backwards compatible version >= `X.Y.Z`
  - Example: `"2.3.1+"` means the currently latest available version >= `"2.3.1"` and < `"2.4.0"` (all forwards and backwards compatible versions >= `"2.3.1"`)
- Non-versioned and 2-digit version SDMX structural artefacts CAN reference any other non-versioned or versioned (whether SemVer or not) SDMX structural artefacts.

- Semantically versioned artefacts MUST only reference other semantically versioned artefacts.
- Wildcarded references in a stable artefact implicitly target only future stable versions of the referenced artefacts within the defined wildcard scope.

    - Example: The reference to "`AGENCY_ID:CODELIST_ID(2.3+.1)`" in an artefact "`AGENCY_ID:DSD_ID(2.2.1)`" resolves to artefact "`AGENCY_ID:CODELIST_ID(2.4.3)`" if that was currently the latest available stable version.

- Wildcarded references in a version-extended artefact implicitly target future stable and version-extended versions of the referenced artefacts within the defined wildcard scope.
    - Example: The reference to "`AGENCY_ID:CODELIST_ID(2.3+.1)`" in an artefact "`AGENCY_ID:DSD_ID(2.2.1-draft)`" resolves to artefact "`AGENCY_ID:CODELIST_ID(2.5.0-draft)`" if that was currently the latest available version.

- References to specific version-extended artefacts MAY be used, but those cannot be combined with a wildcard.
    - Example: The reference to "`AGENCY_ID:CODELIST_ID(2.5.0-draft)`" in an artefact "`AGENCY_ID:DSD_ID(2.2.1)`" resolves to artefact "`AGENCY_ID:CODELIST_ID(2.5.0-draft)`", which might be subject to continued backwards compatible changes.

Because both, wildcarded references and references to version-extended artefacts, allow for changes in the referenced artefacts, care needs to be taken when choosing the appropriate references in order to achieve the required limitation in the allowed scope of changes.

For references to non-dependent artefacts, `MAJOR`, `MINOR` or `PATCH` version parts in SDMX 3.0 artefact references CAN alternatively be wildcarded using "`*`" as replacement:
`*` means all available versions

14.5 Upgrade and conversions of artefacts defined with previous SDMX standard versions to Semantic Versioning

Because SDMX standardises the interactions between statistical systems, which cannot all be upgraded at the same time, the new versioning rules cannot be applied to existing artefacts in EDIFACT, SDMX 1.0, 2.0 or 2.1. SemVer can only be applied to structural artefacts that are newly modelled with the SDMX 3.0 Information Model. Migrating to SemVer means migrating to the SDMX 3.0 Information Model, to its new API version and new versions of its exchange message formats.

To migrate SDMX structural artefacts created previously to SDMX 3.0.0:

If the artefacts do not need versioning, then the new artefacts based on the SDMX 3.0 Information Model SHOULD remain as-is, e.g., a previous artefact with the non-final version 1.0 and that doesn't need versioning becomes non-versioned, i.e., keeps version 1.0. This will be the case for all `AgencyScheme` artefacts.

If artefact versioning is required and SDMX 3.0.0 Semantic Versioning is available within the tools and processes used, then it is recommended to switch to Semantic Versioning with the following steps:

1. Complement the missing version parts with 0s to make the version number SemVer-compliant using the `MAJOR.MINOR.PATCH-EXTENSION` syntax:

   Example: Version 2 becomes version 2.0.0 and version 3.1 becomes version 3.1.0.

2. Replace the "isFinal=false" property by the version extensions "-draft" (or alternatively "-unstable" or "-nonfinal" depending on the use case).

   Example: Version 1.3 with `isFinal=true` becomes version 1.3.0 and version 1.3 with `isFinal=false` becomes version 1.3.0-draft.

If artefact versioning is required but semantic versioning cannot be applied, then version strings used in previous versions of the Standard (e.g., version=1.2) may continue to be used.

Note: Like for other not fully backwards compatible SDMX 3.0 features, also some cases of semantically versioned SDMX 3.0 artefacts cannot be converted back to earlier SDMX versions. This is the case when one or more extensions have been created in parallel to the corresponding stable version. In this case, only the stable version SHOULD be converted to a final version (e.g., 3.2.1 becomes 3.2.1 final, and 3.2.1-draft cannot be converted back).

14.6 FAQ for Semantic Versioning

My organisation is new to SDMX and starts to implement 3.0 or starts to implement a new process fully based on SDMX 3.0. Which versioning scheme should be used?

If all counterparts involved in the process and all tools used for its implementation are SDMX 3.0-ready, then it is recommended to:

- in general, use semantic versioning;
- exceptionally, do not use versioning for artefacts that do not require it, e.g. artefacts that never change, that are only used internally or for which communication on changes with external parties or systems is not required.

How should I deal with revisions in the 0.y.z initial modelling phase?

The simplest thing to do is start your initial modelling release at 0.1.0 and then increment the minor version for each subsequent release.

How do I know when to release 1.0.0?

If your data model is being used in production, it should probably already be 1.0.0. If you have a stable artefact on which users have come to depend, you should be 1.0.0. If you're worrying a lot about backwards compatibility, you should probably already be 1.0.0.

Doesn't this discourage rapid modelling and fast iteration?

Major version zero is all about rapid modelling. If you're changing the artefact every day you should either still be in version 0.y.z or on the next (minor or) major version for a separate modelling.

If even the tiniest backwards incompatible changes to the public artefact require a major version bump, won't I end up at version 42.0.0 very rapidly?

This is a question of responsible modelling and foresight. Incompatible changes should not be introduced lightly to a data model that has a lot of dependencies. The cost that must be incurred to upgrade can be significant. Having to bump major versions to release incompatible changes means you will think through the impact of your changes, and evaluate the cost/benefit ratio involved.

Documenting the version changes in an artefact's annotation of type "CHANGELOG" is too much work!

It is your responsibility as a professional modeller to properly document the artefacts that are intended for use by others. Managing data model complexity is a hugely important part of keeping a project efficient, and that's hard to do if nobody knows how to use your data model, or what artefacts are safe to reuse. In the long run, SDMX 3.0 Semantic Versioning can keep everyone and everything running smoothly.
However, refrain from overdoing. Nobody can and will read too long lists of changes. Thus, keep it to the absolute essence, and mainly use it for short announcements. You can even skip the changelog if the change is impact-less. For all complete reports, a new API feature could be more useful to automatically generate a log of differences between two versions.

What do I do if I accidentally release a backwards incompatible change as a minor version?

As soon as you realise that you've broken the SDMX 3.0 Semantic Versioning specification, fix the problem and release a new minor version that corrects the problem and restores backwards compatibility. Even under this circumstance, it is unacceptable to modify versioned releases. If it's appropriate, document the offending version and inform your users of the problem so that they are aware of the offending version.

What if I inadvertently alter the public artefact in a way that is not compliant with the version number change (i.e. the modification incorrectly introduces a major breaking change in a patch release)?

Use your best judgement. If you have a huge audience that will be drastically impacted by changing the behaviour back to what the public artefact intended, then it may be best to perform a major version release, even though the property change could strictly be considered a patch release. Remember, SDMX 3.0.0 Semantic Versioning is all about conveying meaning by how the version number changes. If these changes are important to your users, use the version number to inform them.

How should I handle deprecating elements?

Deprecating existing elements is a normal part of data modelling and is often required to make forward progress or follow history (changing classifications, evolving reference areas). When you deprecate part of your stable artefact, you should issue a new minor version with the deprecation in place (e.g. add the new country code but still keep the old country code) and with a "CHANGELOG" annotation announcing the deprecation (e.g. the intention to remove the old country code in a future version) . Before you completely remove the functionality in a new major release there should be at least one minor release that contains the deprecation so that users can smoothly transition to the new artefact.

Does SDMX 3.0.0 Semantic Versioning have a size limit on the version string?

No, but use good judgement. A 255 character version string is probably overkill, for example. In addition, specific SDMX implementations may impose their own limits on the size of the string. Remember, it is generally recommended to use the standard extension "-draft".

Is "v1.2.3" a semantic version?

No, "v1.2.3" is not a semantic version. The semantic version is "1.2.3".

Is there a suggested regular expression (RegEx) to check an SDMX 3.0.0 Semantic Versioning string?

There are two:

One with named groups for those systems that support them (PCRE [Perl Compatible Regular Expressions, i.e. Perl, PHP and R], Python and Go).

Reduced version (without original SemVer "build metadata") from: https://regex101.com/r/Ly7O1x/3/

```
^(?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-]*))*))?$
```

And one with numbered capture groups instead (so cg1 = major, cg2 = minor, cg3 = patch and cg4 = extension) that is compatible with ECMA Script (JavaScript), PCRE (Perl Compatible Regular Expressions, i.e. Perl, PHP and R), Python and Go.

Reduced version (without original SemVer "build metadata") from: https://regex101.com/r/vkijKf/1/

```
^(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-]*))*))?$
```

Must I adopt semantic versioning rules when switching to SDMX 3.0?

No. If backwards compatibility with pre-existing tools and processes is required, then it is possible to continue using the previous versioning scheme (with up to two version parts MAJOR.MINOR). Semantic versioning is indicated only for those use cases where a proper artefact versioning is required. If versioning does not apply to some or all of your artefacts, then rather migrate to non-versioned SDMX 3.0 artefacts.

May I mix artefacts that follow semantic versioning with artefacts that don't?

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.

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?

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.

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?

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.

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.

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?

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.

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.

In practice, the migration approach will often mirror the way in which organisations have migrated between earlier SDMX versions. Rarely, the new data models used mixed SDMX standard versions in their dependencies, and if they did then standard conversions were put in place. A typical method is to first migrate the re-used artefacts from the previous SDMX version to SDMX 3.0 and while doing so to apply the appropriate new semantic version string. From that point onwards, you can enjoy the advantages of the new SDMX versioning features for all those artefacts that require appropriate versioning.