sdmx
Statistical Data and Metadata eXchange

SDMX STANDARDS: SECTION 6

SDMX
TECHNICAL NOTES

FOR SDMX VERSION 2.1

Revision 2.0

July 2020

Revision History

Revision	Date	Contents
	April 2011	Initial release
1.0	April 2013	Added section 9 - Transforming between versions of SDMX
2.0	July 2020	Added section 10 – Validation and Transformation Language – before the Annex 1.

Contents

Statistical Data and Metadata eXchange

1 Purpose and Structure

1.1 Purpose

The intention of this document 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 schema 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, partial code list

2 General Notes on This Document

At this version of the standards, the term "Key family" is 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 term "Key family" is still used in the SDMX-EDI specification.

There has been much work within the SDMX community on the creation of user guides, tutorials, and other aides 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 and SDMX-EDI – that are concerned with data, i.e. Data Structure Definitions and Data Sets. This section is intended to provide information which 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* and *SDMX-EDI: Syntax 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 or EDI 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.

The Data Structure Definition and Data Set parts of the information model are consistent with the GESMES/TS version 3.0 Data Model (called SDMX-EDI in the SDMX standard), with these exceptions:

the "sibling group" construct has been generalized to permit any dimension or dimensions to be wildcarded, and not just frequency, as in GESMES/TS. It has been renamed a "group" to distinguish it from the "sibling group" where only frequency is wildcarded. The set of allowable partial "group" keys must be declared in the DSD, and attributes may be attached to any of these group keys;

furthermore, whilst the "group" has been retained for compatibility with version 2.0 and with SDMX-EDI, it has, at version 2.1, been replaced by the "Attribute Relationship" definition which is explained later

the section on data representation is now a convention, to support interoperability with EDIFACT-syntax implementations (see section 3.3.2);

DSD-specific data formats are derived from the model, and some supporting features for declaring multiple measures have been added to the structural metadata descriptions

Clearly, this is not a coincidence. The GESMES/TS Data Model provides the foundation for the EDIFACT messages in SDMX-EDI, and also is the starting point for the development of SDMX-ML.

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

3.3 SDMX-ML and SDMX-EDI: Comparison of 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 SDM-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 Definitionand the Date Set.

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 are just two types of data formats: *GenericData* and *StructureSpecificData* (i.e. specific to one Data Structure Definition).

Both of these formats are now flexible enough to allow for data to be oriented in series with any dimension used to disambiguate the observations (as opposed to only time or a cross sectional measure in version 2.0). The formats have also been expanded to allow for ungrouped observations.

To allow for applications which only understand time series data, variations of these formats have been introduced in the form of two data messages; *GenericTimeSeriesData* and *StructureSpecificTimeSeriesData*. It is important to note that these variations are built on the same root structure and can be processed in the same manner as the base format so that they do NOT introduce additional processing requirements.

Structure Definition

The SDMX-ML Structure Message supports the use of annotations to the structure, which is not supported by the SDMX-EDI syntax.

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 not available to those using the SDMX-EDI structure message.

Validation

SDMX-EDI – as is typical of EDIFACT syntax messages – leaves validation to dedicated applications (“validation” being the checking of syntax, data typing, and adherence of the data message to the structure as described in the structural definition.)

The SDMX-ML Generic Data Message also leaves validation above the XML syntax level to the application.

The SDMX-ML DSD-specific messages will allow validation of XML syntax and datatyping 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.

Update and Delete Messages and Documentation Messages

All SDMX data messages allow for both delete messages and messages consisting of only data or only documentation.

Character Encodings

All SDMX-ML messages use the UTF-8 encoding, while SDMX-EDI uses the ISO 8879-1 character encoding. There is a greater capacity with UTF-8 to express some character sets (see the “APPENDIX: MAP OF ISO 8859-1 (UNOC) CHARACTER SET (LATIN 1 OR “WESTERN”) in the document “SYNTAX AND DOCUMENTATION VERSION 2.0”.) Many transformation tools are available which allow XML instances with UTF-8 encodings to be expressed as ISO 8879-1-encoded characters, and to transform UTF-8 into ISO 8879-1. Such tools should be used when transforming SDMX-ML messages into SDMX-EDI messages and vice-versa.

Data Typing

The XML syntax and EDIFACT syntax have different data-typing mechanisms. The section below provides a set of conventions to be observed when support for messages in both syntaxes is required. For more information on the SDMX-ML representations of data, see below.

3.3.2 Data Types

The XML syntax has a very different mechanism for data-typing than the EDIFACT syntax, and this difference may create some difficulties for applications which support both EDIFACT-based and XML-based SDMX data formats. This section provides a set of conventions for the expression in data in all formats, to allow for clean interoperability between them.

It should be noted that this section does not address character encodings – it is assumed that conversion software will include the use of transformations which will map between the ISO 8879-1 encoding of the SDMX-EDI format and the UTF-8 encoding of the SDMX-ML formats.

Note that the following conventions may be followed for ease of interoperation between EDIFACT and XML representations of the data and metadata. For implementations in which no transformation between EDIFACT and XML syntaxes is foreseen, the restrictions below need not apply.

1. **Identifiers** are:
    - Maximum 18 characters;
    - Any of A..Z (upper case alphabetic), 0..9 (numeric), _ (underscore);
    - The first character is alphabetic.

2. **Names** are:
    - Maximum 70 characters.
    - From ISO 8859-1 character set (including accented characters)

3. **Descriptions** are:
    - Maximum 350 characters;
    - From ISO 8859-1 character set.

4. **Code values** are:
    - Maximum 18 characters;
    - Any of A..Z (upper case alphabetic), 0..9 (numeric), _ (underscore), / (solidus, slash), = (equal sign), - (hyphen);

However, code values providing values to a dimension must use only the following characters:

A..Z (upper case alphabetic), 0..9 (numeric), _ (underscore)

5. **Observation values** are:
    - Decimal numerics (signed only if they are negative);
    - The maximum number of significant figures is:
        - 15 for a positive number
        - 14 for a positive decimal or a negative integer
        - 13 for a negative decimal
    - Scientific notation may be used.

6. **Uncoded statistical concept** text values are:
    - Maximum 1050 characters;

- From ISO 8859-1 character set.

7. **Time series keys**:

In principle, the maximum permissible length of time series keys used in a data exchange does not need to be restricted. However, for working purposes, an effort is made to limit the maximum length to 35 characters; in this length, also (for SDMX-EDI) one (separator) position is included between all successive dimension values; this means that the maximum length allowed for a pure series key (concatenation of dimension values) can be less than 35 characters. The separator character is a colon (":") by conventional usage.

3.4 SDMX-ML and SDMX-EDI 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.

<u>Dimensions, Attributes and Code Lists</u>

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 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.

<u>Data Structure Definition Structure</u>

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
- ordinal number of the dimension in the key structure (SDMX-EDI only)
- 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
- assignment status: mandatory or conditional
- attachment level
- 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 data flow definitions. Two (or more) partners performing data exchanges in a certain context need to agree on:

- the list of data set identifiers they will be using;

- for each data flow:
- its content and description
- the relevant DSD that defines the structure of the data reported or disseminated according the the dataflow definition

3.4.1.3 Exchanging Attributes

3.4.1.3.1 Attributes on series, sibling and data set level

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 or SDMX-EDI messages (e.g. paper, telephone) it is expected that partner institutions will be in a position to provide this information in SDMX-ML or SDMX-EDI format over time.

- A centre may agree with its data exchange partners special procedures for authorising the setting of attributes' initial values.

- Attribute values at a data set level are set and maintained exclusively by the centre administrating the exchanged data set.

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 sibling 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".

- In SDMX-EDI, the observation level attribute “observation status” is part of the fixed syntax of the ARR segment used for observation reporting. Whenever an observation is exchanged, the corresponding observation status must also be exchanged attached to the observation, regardless of whether it has changed or not since the previous data exchange. This rule also applies to the use of the SDMX-ML formats, although the syntax does not necessarily require this.

- If the “observation status” changes and the observation remains unchanged, both components would have to be reported.

- For Data Structure Definitions having also the observation level attributes “observation confidentiality” and "observation pre-break" defined, this rule applies to these attribute 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-EDI formats, and might also use the SDMX-ML DSD-specific data set. The following points apply equally to both formats.

3.4.2.2 Positioning of the Dimension "Frequency"

The position of the “frequency” dimension is unambiguously identified in the data structure definition. Moreover, most central institutions devising structural definitions have decided to assign to this dimension the first position in the key structure. This facilitates the easy identification of this dimension, something that it 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, most 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.

3.4.2.4 Identification of the Data Flows

In order to facilitate the easy and immediate recognition of the institution administrating a data flow definitions, many central institutions prefer to use the first characters of the data flow definition identifiers to identify their institution: e.g. BIS_EER, ECB_BOP1, ECB_BOP1, etc. Note that in GESMES/TS the Data Set plays the role of the data flow definition (see *DataSet* in the SDMX-IM).

The statistical information in SDMX is broken down into two fundamental parts - structural metadata (comprising the Data Structure Definition, and associated

Concepts and Code Lists) - 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 organized into data sets (*DataSet*) These data sets are structured according to a specific Data Structure Definition (*DataStructureDefinition*) and are described in the data flow definition (*DataflowDefinition)* The Data Structure Definition describes the metadata that allows an understanding of what is expressed in the data set, whilst the data flow definition provides the identifier and other important information (such as the periodicity of reporting) that is common to all of its component data sets.

Note that the role of the Data Flow (called *DataflowDefintion* in the model) and Data Set is very specific in the model, and the terminology used may not be the same as used in all organisations, and specifically the term Data Set is used differently in SDMX than in GESMES/TS. Essentially the GESMES/TS term "Data Set" is, in SDMX, the "Dataflow Definition" whist 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. However, for data exchange purposes, such series can already be exchanged in the SDMX-EDI format by using the option to send observations with the associated time stamp.

4 General Notes for Implementers

This section discusses a number of topics other than the exchange of data sets in SDMX-ML and SDMX-EDI. Supported only in SDMX-ML, 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 some of the conventional mechanisms within the SDMX-ML Structure message regarding versioning and external referencing.

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.

4.1 Representations

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.TimeSpa	javax.xml.datatype

SDMX-ML Data Type	XML Schema Data Type	.NET Framework Type	Java Data Type
		n	.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, start DateTime + Duration,)
- ObservationalTimePeriod (common: ObservationalTimePeriodType, a union of StandardTimePeriod and TimeRange).
- StandardTimePeriod (common: StandardTimePeriodType, a union of BasicTimePeriod and TimeRange).
- 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 identifiable object)
- DataSetReference (common:DataSetReferenceType)
- AttachmentConstraintReference (common:AttachmentConstraintReferenceType)

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.s

4.2 Time and Time Format

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)[1]

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

[1] The seconds can be reported fractionally

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)

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.[2] 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).

---

[2] 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.

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 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[3] (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[3] (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[3] this to the [REPORTING_YEAR_BASE]. The result is the [PERIOD_START].
4. **Determine the [PERIOD_END]:**
   Multiply the [PERIOD_VALUE] by the [PERIOD_DURATION]. Add[3] this to the [REPORTING_YEAR_BASE] add[3] -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] = 2011-12-31

   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

---

[3] 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.

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 postive.

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)

Code	Format
RW	Reporting Week (YYYY-Www)
RD	Reporting Day (YYYY-Dddd)

Table 1: SDMX-ML Time Format Codes

4.2.9 Transformation between SDMX-ML and SDMX-EDI

When converting SDMX-ML data structure definitions to SDMX-EDI data structure definitions, only the identifier of the time format attribute will be retained. The representation of the attribute will be converted from the SDMX-ML format to the fixed SDMX-EDI code list. If the SDMX-ML data structure definition does not define a time format attribute, then one will be automatically created with the identifier "TIME_FORMAT".

When converting SDMX-ML data to SDMX-EDI, the source time format attribute will be irrelevant. Since the SDMX-ML time representation types are not ambiguous, the target time format can be determined from the source time value directly. For example, if the SDMX-ML time is 2000-Q2 the SDMX-EDI format will always be 608/708 (depending on whether the target series contains one observation or a range of observations)

When converting a data structure definition originating in SDMX-EDI, the time format attribute should be ignored, as it serves no purpose in SDMX-ML.
When converting data from SDMX-EDI to SDMX-ML, the source time format is only necessary to determine the format of the target time value. For example, a source time format of 604 will result in a target time in the format YYYY-Ss whereas a source format of 608 will result in a target time value in the format YYYY-Qq.

4.2.10 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.

4.2.11 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"/>
```

4.2.12 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.

4.2.13 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.

4.2.14 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. In addition, an explicit value for the reporting year start day can be provided, or this can be set to "Any". 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 is 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 based on whether the value of the reportingYearStartDay XML attribute is an explicit month and day or "Any":

> If the time parameter provides an explicit month and day value for the reportingYearStartDay XML attribute, then the parameter value is converted to a distinct range and processed as any other time period would be processed.
>
> If the reportingYeartStartDay XML attribute has a value of "Any", then 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.

Note that the reportingYearStartDay XML attribute on the time value parameter is only used to qualify a reporting period value for the given time value parameter. The usage of this is different than using the attribute value parameter for the actual reporting year start day attribute. In the case that the attribute value parameters is used for the reporting year start day data structure attribute, it will be treated as any other attribute value parameter; data will be filtered to that which matches the values specified for the given attribute. For example, if the attribute value parameter references the reporting year start day attribute and specifies a value of "--07-01", then only data which has this attribute with the value "--07-01" will be returned. In terms of processing any time value parameters, the value supplied in the attribute value parameter will be irrelevant.

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 with explicit start day

Query Parameter: Greater than or equal to 2009-Q3, reporting year start day = "--07-01"

Literal Interpretation: Any data where the start period occurs on after 2010-01-01T00:00:00 (Note that in this case 2009-Q3 is converted to the explicit date range of 2010-01-01/2010-03-31 because of the reporting year start day value).

Example Matches: Same as previous example

Reporting Period with "Any" start day

Query Parameter: Greater than or equal to 2010-Q3, reporting year start day = "Any"

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)[4]
- 2010-D182 or later (reporting year start day --01-01)
- 2010-W28 or later (reporting year start day --07-01)[5]

---

[4] 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.

- 2010-D185 or later (reporting year start day --07-01)

4.3 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.

4.4 Versioning and External Referencing

Within the SDMX-ML Structure Message, there is a pattern for versioning and external referencing which should be pointed out. The 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” is a different object than one with an Agency of “A’, an ID of “X”, and a version of “1.1”.

The production versions of identifiable objects/resources are assumed to be static – that is, they have their isFinal attribute set to ‘true”. Once in production, and object cannot change in any way, or it must be versioned. For cases where an object is not static, the isFinal attribute must have a value of “false”, but non-final objects should not be used outside of a specific system designed to accommodate them. For most purposes, all objects should be declared final before use in production.

This mechanism is an “early binding” one – everything with a versioned identity is a known quantity, and will not change. It is worth pointing out that in some cases relationships are essentially one-way references: an illustrative case is that of Categories. While a Category may be referenced by many dataflows and metadata flows, the addition of more references from flow objects does not version the Category. This is because the flows are not properties of the Categories – they merely make references to it. If the name of a Category changed, or its sub-Categories changed, then versioning would be necessary.

---

[5] 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.

Versioning operates at the level of versionable and maintainable objects in the SDMX information model. If any of the children of objects at these levels change, then the objects themselves are versioned.

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, code lists, concept schemes, etc.) In an SDMX-ML 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-ML 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.

Because the version number is part of the identifier for an object, versions are a necessary part of determining that a given resource is the one which was called for. It should be noted that whenever a version number is not supplied, it is assumed to be “1.0”. (The “x.x” versioning notation is conventional in practice with SDMX, but not required.)

5 Metadata Structure Definition (MSD)

5.1 Scope

The scope of the MSD is enhanced in this version to better support the types of construct to which metadata can be attached. In particular it is possible to specify an attachment to any key or partial key of a data set. This is particularly useful for web dissemination where metadata may be present for the data, but is not stored with the data but is related to it. For this use case to be supported it is necessary to be able to specify in the MSD that metadata is attached to a key or partial key, and the actual key or partial key to be identified in the Metadata Set.

In addition to the increase in the scope of objects that can be included in an MSD, the way the identifier mechanism works in this version, and the terminology used, is much simpler.

5.2 Identification of the Object Type to which the Metadata is to be Attached

The following example shows the structure and naming of the MSD components for the use case of defining full and partial keys.

The schematic structure of an MSD is shown below.

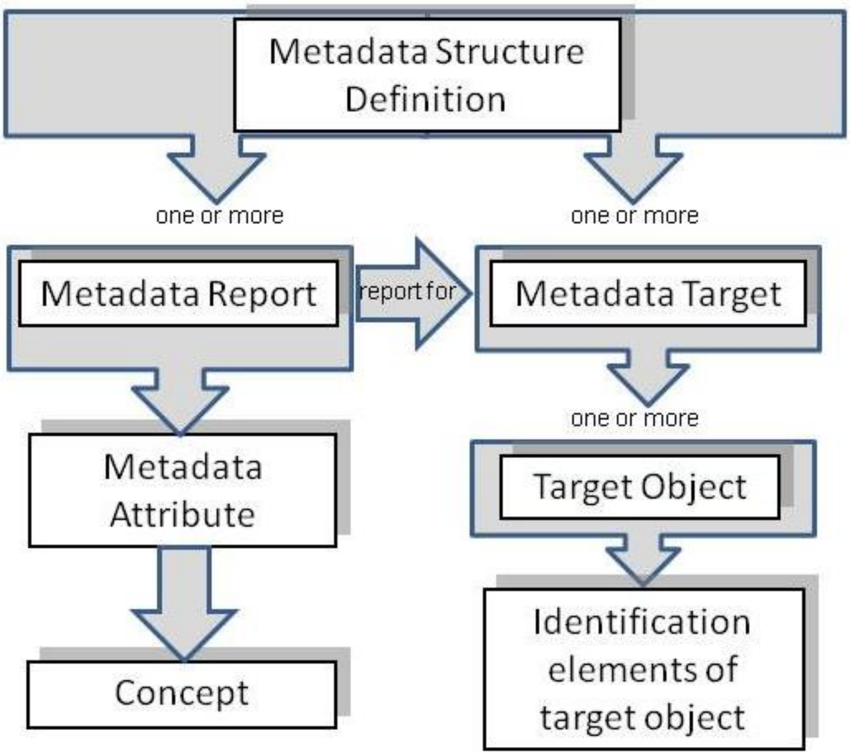


Figure 1: Schematic of the Metadata Structure Definition

The MSD comprises the specification of the object types to which metadata can be reported in a Metadata Set (Metadata Target(s)), and the Report Structure(s) comprising the Metadata Attributes that identify the Concept for which metadata may be reported in the Metadata Set. Importantly, one Report Structure references the Metadata Target for which it is relevant. One Report Structure can reference many Metadata Target i.e. the same Report Structure can be used for different target objects.

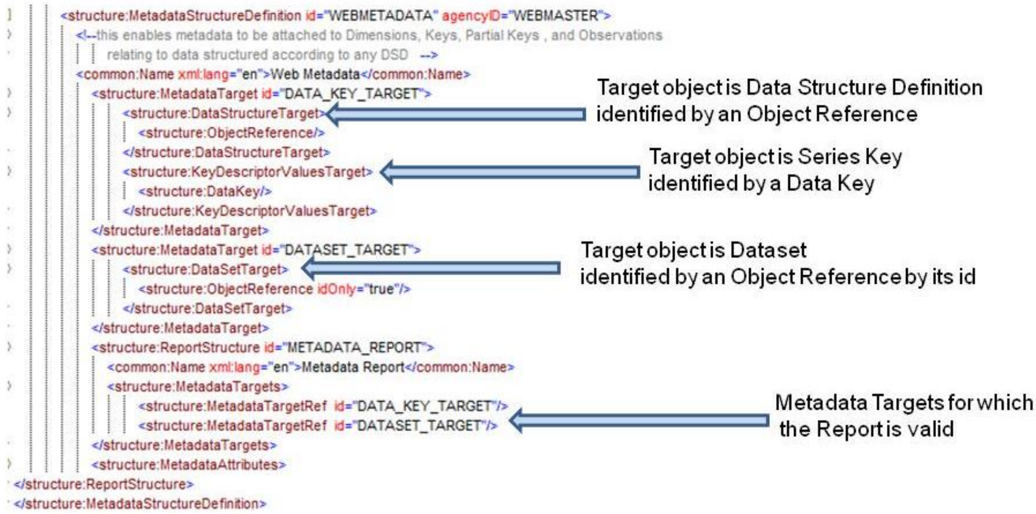


Figure 2: Example MSD showing Metadata Targets

Note that the SDMX-ML schemas have explicit 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 Report Structure

An example is shown below.

```xml
<structure:MetadataStructureDefinition id="WEBMETADATA" agencyID="WEBMASTER">
  <!--this enables metadata to be attached to Dimensions, Keys, Partial Keys , and Observations
      relating to data structured according to any DSD  -->
  <common:Name xml:lang="en">Web Metadata</common:Name>
    <structure:MetadataTarget id="DATA_KEY_TARGET">
    <structure:MetadataTarget id="DATASET_TARGET">
    <structure:ReportStructure id="METADATA_REPORT">
      <common:Name xml:lang="en">Metadata Report</common:Name>
      <structure:MetadataTargets>
          <structure:MetadataTargetRef  id="DATA_KEY_TARGET"/>
          <structure:MetadataTargetRef  id="DATASET_TARGET"/>
    </structure:MetadataTargets>
    <structure:MetadataAttributes>
      <structure:MetadataAttribute isPresentational="true">
        <structure:ConceptReference>
          <common:ConceptSchemeRef>
            <common:Ref id="IMETADATA_CONCEPTS" agencyID="WEBMASTER" version="1.0"/>
          </common:ConceptSchemeRef>
          <common:ConceptRef id="SOURCE"/>
        </structure:ConceptReference>
        <structure:MetadataAttribute>
          <structure:ConceptReference>
            <common:ConceptSchemeRef>
              <common:Ref id="IMETADATA_CONCEPTS" agencyID="WEBMASTER" version="1.0"/>
            </common:ConceptSchemeRef>
            <common:ConceptRef id="SOURCE_TYPE"/>
          </structure:ConceptReference>
        </structure:MetadataAttribute>
        <structure:MetadataAttribute>
          <structure:ConceptReference>
            <common:ConceptSchemeRef>
              <common:Ref id="METADATA_CONCEPTS" agencyID="WEBMASTER" version="1.0"/>
            </common:ConceptSchemeRef>
            <common:ConceptRef id="COLLECTION_SOURCE_NAME"/>
          </structure:ConceptReference>
        </structure:MetadataAttribute>
      <structure:MetadataAttribute>
      <!-- and so on for the reamaining metadata attribute -->
    </structure:MetadataAttribute>
  </structure:MetadataAttribute>
 </structure:MetadataAttributes>
</structure:ReportStructure>
</structure:MetadataStructureDefinition>
```

Figure 3: Example MSD showing specification of three Metadata Attributes

This example shows the following hierarchy of Metadata Attributes:

Source – this is presentational and no metadata is expected to be reported at this level

- Source Type
- Collection Source Name

5.4 Metadata Set

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


```
<g:MetadataSet>
  <c:MetadataStructureDefinitionReference>
    <c:Ref id="WEBMETADATA" agencyID="WEBMASTER" version="1.0"/>
  </c:MetadataStructureDefinitionReference>
  <g:AttributeValueSet>
    <g:ReportRef>METADATA_REPORT</g:ReportRef>
    <!-- This is a partial key report (combination of codes from different dimensions) -->
    <g:TargetValues>
      <g:MetadataTargetValue id="DATA_KEY_TARGET">
        <g:ReferenceValue>
          <c:DataStructureReference>
            <c:Ref id="FINANCE_DSD" agencyID="WEBMASTER" version="1.0"/>
          </c:DataStructureReference>
        </g:ReferenceValue>
        <g:ReferenceValue>
          <c:DataKey>
          <c:DataKeyValue dimensionID="ECONOMICCONCEPT">
            <c:DimensionValue>ENDA</c:DimensionValue>
          </c:DataKeyValue>
          <c:DataKeyValue dimensionID="DATASOURCE">
            <c:DimensionValue>IFS</c:DimensionValue>
          </c:DataKeyValue>
        </c:DataKey>
        </g:ReferenceValue>
      </g:MetadataTargetValue>
    </g:TargetValues>
    <g:ReportedAttribute id="SOURCE">
      <g:ReportedAttribute id="SOURCE_TYPE">
        <g:Value>Market Values</g:Value>
      </g:ReportedAttribute>
      <g:ReportedAttribute id="COLLECTION_SOURCE_NAME">
        <g:Value>These series are typically the monthly average of market rates or official rates of the reporting countr
are not available, they are the monthly average rates in New York. Or if the latter are not available, they are estimates basec
averages of the end-of-month market rates quoted in the reporting country.
        </g:Value>
      </g:ReportedAttribute>
    </g:ReportedAttribute>
  </g:AttributeValueSet>
</g:MetadataSet>
```


Figure 4: Example Metadata Set

This example shows:

1. The reference to the MSD, Metadata Report, and Metadata Target (MetadataTargetValue)

2. The reported metadata attributes (AttributeValueSet)

6 Maintenance Agencies

All structural metadata in SDMX is owned and maintained by a maintenance agency (Agency identified by `agencyID` in the schemas). It is vital to the integrity of the structural metadata that there are no conflicts in `agencyID`. In order to achieve this SDMX adopts the following rules:

1. Agencies are maintained in an Agency Scheme (which is a sub class of Organisation Scheme)
2. The maintenance agency of the Agency Scheme must also be declared in a (different) Agency Scheme.
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 Agency Scheme. SDMX is an agency in the SDMX agency scheme. Agencies in this scheme can themselves maintain a single Agency Scheme and so on.
5. The `AgencyScheme` cannot be versioned and so take a default version number of 1.0 and cannot be made “final”.
6. There can be only one `AgencyScheme` maintained by any one Agency. It has a fixed Id of AgencyScheme.
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.

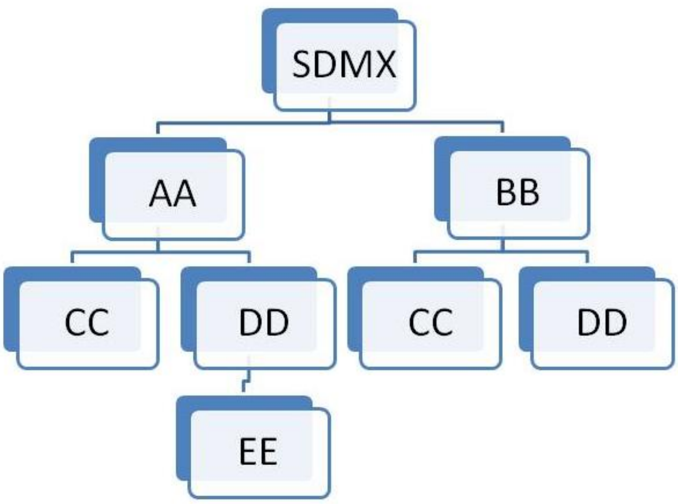


Figure 5: Example of Hierarchic Structure of Agencies

Each agency is identified by its full hierarchy excluding SDMX.

The XML representing this structure is shown below.

```
<structure:Organisations>
  <structure:AgencyScheme agencyID="SDMX" id="AGENCY_SCHEME">
    <common:Name>name</common:Name>
    <structure:Agency id="AA">
      <common:Name>AA Name</common:Name>
    </structure:Agency>
    <structure:Agency id="BB">
      <common:Name>BB Name</common:Name>
    </structure:Agency>
  </structure:AgencyScheme>
<structure:AgencyScheme agencyID="AA" id="AGENCY_SCHEME">
    <common:Name>name</common:Name>
    <structure:Agency id="CC">
      <common:Name>CC Name</common:Name>
    </structure:Agency>
    <structure:Agency id="DD">
      <common:Name>DD Name</common:Name>
    </structure:Agency>
</structure:AgencyScheme>
<structure:AgencyScheme agencyID="BB" id="AGENCY_SCHEME">
    <common:Name>name</common:Name>
    <structure:Agency id="CC">
      <common:Name>CC Name</common:Name>
    </structure:Agency>
    <structure:Agency id="DD">
      <common:Name>DD Name</common:Name>
    </structure:Agency >
</structure:AgencyScheme>
<structure:AgencyScheme agencyID="AA.CC" id="AGENCY_SCHEME">
    <common:Name>name</common:Name>
    <structure:Agency id="EE">
      <common:Name>EE Name</common:Name>
    </structure:Agency >
</structure:AgencyScheme>
</structure:Organisations>
```

Figure 6: Example Agency Schemes Showing a Hierarchy

<u>Example of Structure Definitions:</u>

```
<structure:Codelists>
<structure:Codelist id"=CL_BOP" agencyID="SDMX" version="1.0"
 urn="urn:sdmx:org.sdmx.infomodel. codelist.Codelist =SDMX:CL_BOP[1.0]">
    <common:Name>name</common:Name>
</structure:Codelist>
<structure:Codelist id"=CL_BOP" agencyID="AA" version="1.0"
 urn="urn:sdmx:org.sdmx.infomodel. codelist.Codelist =AA:CL_BOP[1.0]" >
    <common:Name>name</common:Name>
</structure:Codelist>
<structure:Codelist id"=CL_BOP" agencyID="AA.CC" version="1.0"
 urn="urn:sdmx:org.sdmx.infomodel.codelist.Codelist=AA.CC:CL_BOP[1.0]" >
    <common:Name>name</common:Name>
</structure:Codelist>
<structure:Codelist id"=CL_BOP" agencyID="BB.CC" version="1.0"
 urn="urn:sdmx:org.sdmx.infomodel. codelist.Codelist =BB.CC:CL_BOP[1.0]">
    <common:Name>name</common:Name>
</structure:Codelist>
</structure:Codelists>
```

Figure 7: Example Showing Use of Agency Identifiers

Each of these maintenance agencies has an identical Codelist with the Id CL_BOP. However, each is uniquely identified by means of the hierarchic agency structure.

7 Concept Roles

7.1 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
Unit of Measure - in a data set the content of this Component contains information on the unit of measure of the observation values

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.

7.2 Information Model

The Information Model for this is shown below:

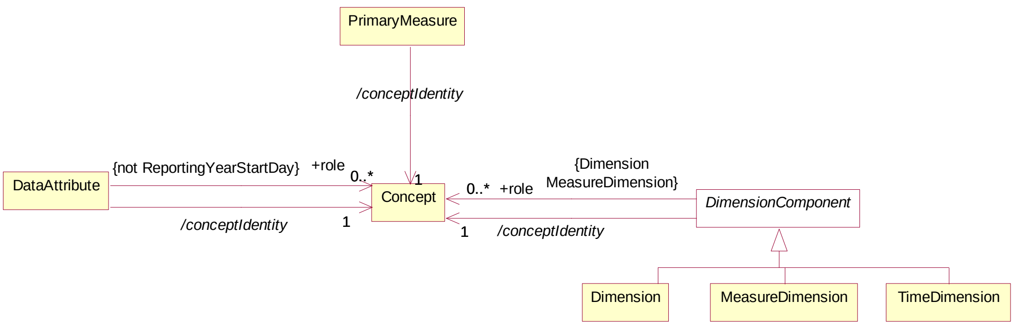


Figure 8: Information Model Extract for Concept Role

It is possible to specify zero or more concept roles for a Dimension, Measure Dimension and Data Attribute (but not the ReportingYearStartDay). The Time Dimension, Primary Measure, and the Attribute ReportingYearStartDay have explicitly defined roles and cannot be further specified with additional concept roles.

7.3 Technical Mechanism

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

1. 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.

2. The concept scheme that contains the “role” concepts can contain concepts that do not play a role.

3. There is no explicit indication on the Concept whether it is a ‘role” concept.

4. Therefore, any concept in any concept scheme is capable of being a “role” concept.

5. 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”.

6. 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.

7. 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.

7.4 SDMX-ML Examples in a DSD

The Cross-Domain Concept Scheme maintained by SDMX contains concept role concepts (FREQ chosen as an example).

```
<structure:Dimension id="FREQ">
  <structure:ConceptIdentity>
    <URN>
urn:sdmx:org.sdmx.infomodel.conceptscheme.Concept=SDMX:CROSS_DOMAIN_CONCEPTS[1.0].FREQ
</URN>
  </structure:ConceptIdentity>
</structure:Dimension>
```

Whether this is a role or not depends upon the application understanding that FREQ in the Cross-Domain Concept Scheme is a role of Frequency.

Using a Concept Scheme that is not the Cross-Domain Concept Scheme where it is required to assign a role using the Cross-Domain Concept Scheme. Again FREQ is chosen as the example.

```xml
<structure:Dimension id="FREQ">
  <structure:ConceptIdentity>
    <URN>
urn:sdmx:org.sdmx.infomodel.conceptscheme.Concept=JBG:MY_CONCEPTS[1.0].FREQ
    </URN>
  </structure:ConceptIdentity>
  <structure:ConceptRole>
    <URN>
urn:sdmx:org.sdmx.infomodel.conceptscheme.Concept=SDMX:CROSS_DOMAIN_CONCEPTS[1.0].FREQ
    </URN>
  </structure: ConceptRole >
</structure:Dimension>
```

This explicitly states that this Dimension is playing a role identified by the FREQ concept in the Cross-Domain Concept Scheme. Again the application needs to understand what FREQ in the Cross-Domain Concept Scheme implies in terms of a role.

This is all that is required for interoperability within a community. The important point is that a community must recognise a specific Agency as having the authority to define concept roles and to maintain these “role” concepts in a concept scheme together with documentation on the meaning of the role and any relevant processing implications. This will then ensure there is interoperability between systems that understand the use of these concepts.

Note that each of the Components (Data Attribute, Primary Measure, Dimension, Measure Dimension, Time Dimension) has a mandatory identity association (Concept Identity) and if this Concept also identifies the role then it is possible to state this by

7.5 SDMX Cross Domain Concept Scheme

All concepts in the SDMX Cross Domain Concept Scheme are capable of playing a role and this scheme will contain all of the roles that were allowed at version 2.0 and will be maintained with new roles that are agreed at the level of the community using the Cross Domain Concept Scheme.

The table below lists the Concepts that need to be in this scheme either for compatibility with version 2.0 or because of requests for additional roles at version 2.1 which have been accepted.

Note that each of the Components (Data Attribute, Primary Measure, Dimension, Measure Dimension, Time Dimension) has a mandatory identity association (Concept Identity) and if this Concept also identifies the role then it is possible to state this by means of the `isRole` attribute (`isRole=true`) Additional roles can still be specified by means of the `+role` association to additional Concepts that identify the role.

8 Constraints

8.1 Introduction

In this version of SDMX the Constraints is a Maintainable Artefact can be associated to one or more of:

- Data Structure Definition
- Metadata Structure Definition
- Dataflow
- Metadataflow
- Provision Agreement
- Data Provider (this is restricted to a Release Calendar Constraint)
- Simple or Queryable Datasources

Note that regardless of the artifact to which the Constraint is associated, it is constraining the contents of code lists in the DSD to which the constrained object is related. This does not apply, of course, to a Data Provider as the Data Provider can be associated, via the Provision Agreement, to many DSDs. Hence the reason for the restriction on the type of Constraint that can be attached to a Data Provider.

8.2 Types of Constraint

The Constraint can be of one of two types:

- Content constraint
- Attachable constraint

The attachable constraint is used to define “cube slices” which identify sub sets of data in terms of series keys or dimension values. The purpose of this is to enable metadata to be attached to the constraint, and thereby to the cube slices defined in the Constraint. The metadata can be attached via the “reference metadata” mechanism – MSD and Metadata Set – or via a Group in the DSD. Below is snippet of the schema for a DSD that shows the constructs that enable the Constraint to referenced from a Group in a DSD.

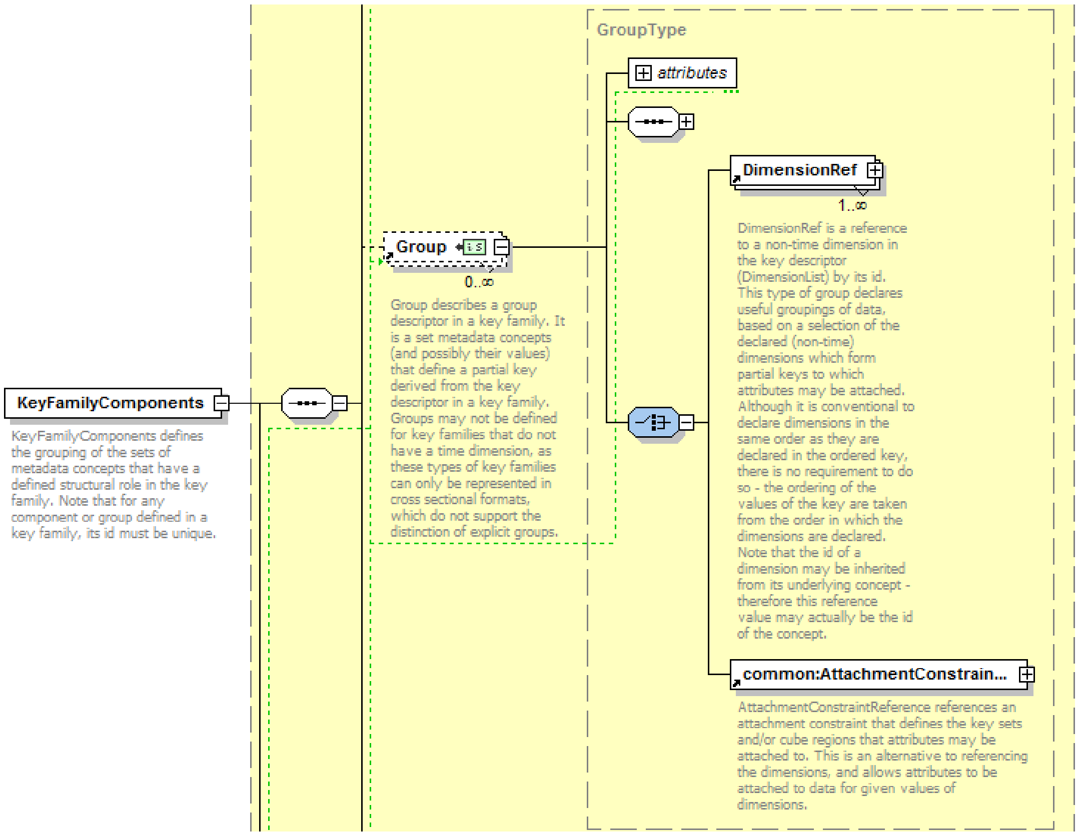


Figure 9: Extract from the SDMX-ML Schema showing reference to Attachment Constraint

For the Content Constraint specific “inheritance” rules apply and these are detailed below.

8.3 Rules for a Content Constraint

8.3.1 Scope of a Content Constraint

A Content 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
- Measure Dimension
- Time Dimension
- Data Attribute
- Primary Measure

And the keys are the content of the KeyDescriptor – i.e. the series keys composed, for each key, by a value for each Dimension and Measure Dimension

In terms of reference metadata the components are:

- Target Object which is one of:

  - Key Descriptor Values
  - Data Set
  - Report Period
  - IdentifiableObject

- Metadata Attribute

The “key” is therefore the combination of the Target Objects that are defined for the Metadata Target.

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

- Data Structure Definition
- Dataflow
- Provision Agreement

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

- Metadata Structure Definition
- Metadataflow
- Provision Agreement

Furthermore, there can be more than one Content 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.

8.3.2 Multiple Content Constraints

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

8.3.2.1 Cube Region

1. The constraint can contain multiple Member Selections (e.g. Dimension) but:
2. A specific Member Selection (e.g. Dimension FREQ) can only be contained in one Content Constraint for any one attached object (e.g. a specific DSD or specific Dataflow)

8.3.2.2 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.

8.3.3 Inheritance of a Content Constraint

8.3.3.1 Attachment levels of a Content 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

Note that these rules do not apply to the Simple Datasoucre or Queryable Datasource: the Content Constraint(s) attached to these artefacts are resolved for this artefact only and do not take into account Constraints attached to other artefacts (e.g. Provision Agreement. Dataflow, DSD).

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

8.3.3.2 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.

8.3.3.3 Cube Region

1. It is not necessary to have a constraint on the higher level artifact (e.g. DSD referenced by the Dataflow) but if there is such a constraint at the higher level(s) then:
    a. 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 constraints that Member Selection (e.g. 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).
    b. 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).
2. Any Member Selection which is not referenced in a Content Constraint is deemed to be constrained according to the Content Constraint specified at the next higher level which constraints that Member Selection.

3. 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 Content Constraint at a higher level to constrain, say, four Dimensions in a single constraint, and a Content Constraint at a lower level to constrain the same four in two, three, or four Content Constraints.

8.3.3.4 Key Set

1. 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:

    a. The lower level constraint cannot be less restrictive than the constraint specified at the higher level.
    b. The constraint at the lower level for any one Member Selection further constrains the keys specified at the higher level(s).
2. Any Member Selection which is not referenced in a Content Constraint is deemed to be constrained according to the Content Constraint specified at the next higher level which constraints that Member Selection.
3. 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).

8.3.4 Constraints Examples

The following scenario is used.

<u>DSD</u>

This 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 Content Constraint is attached.

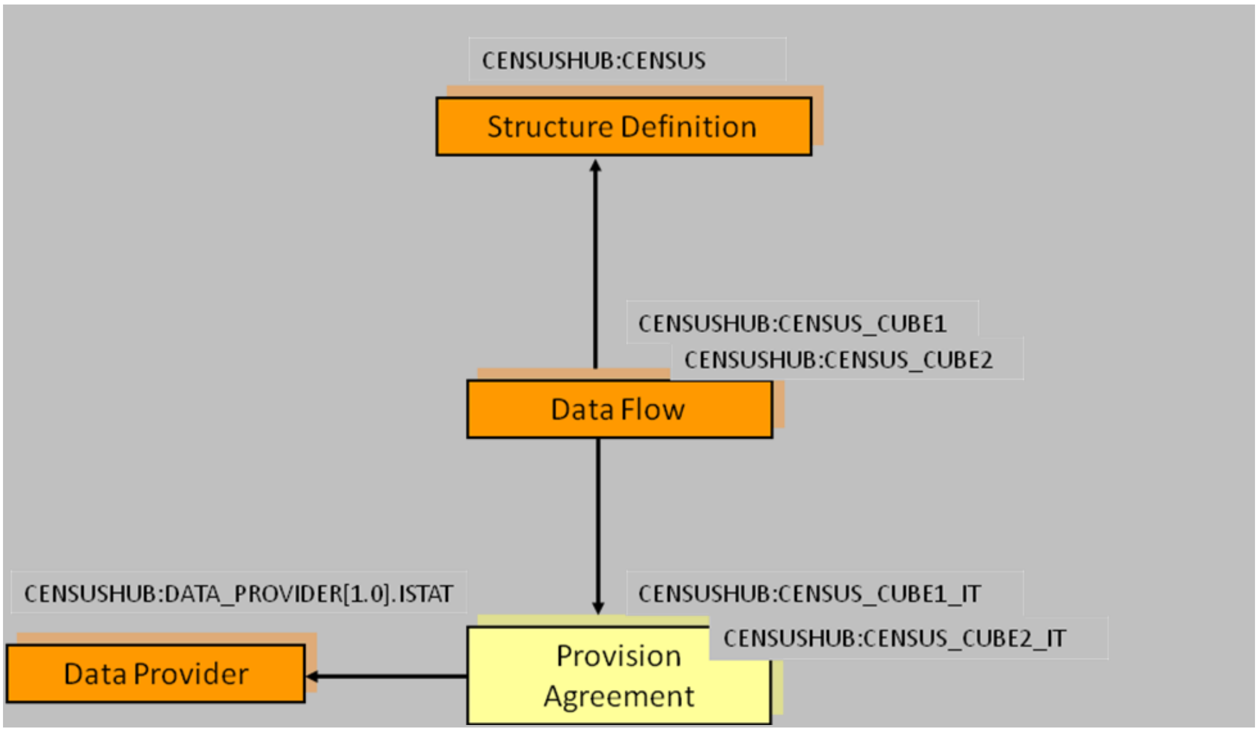


Figure 10: Example Scenario for Constraints

Constraints are declared as follows:

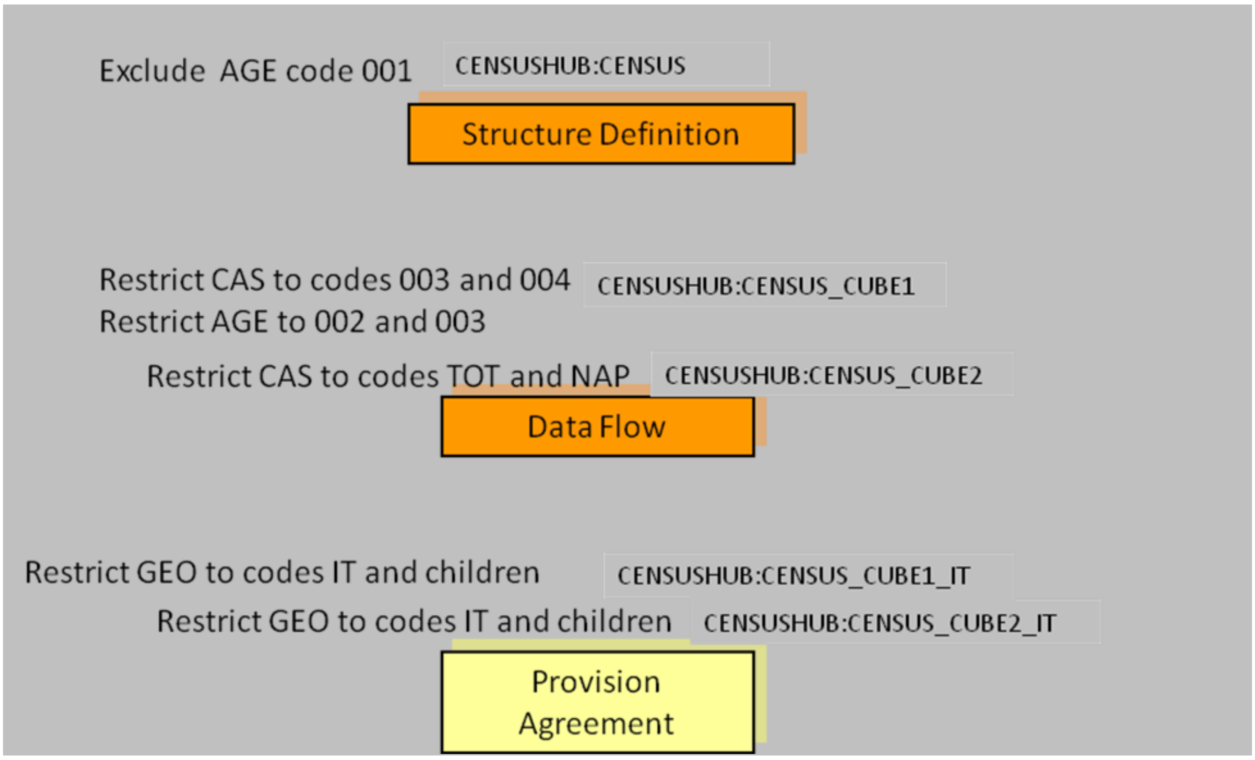


Figure 11: Example Content Constraints

Notes:

1. AGE is constrained for the DSD and is further restricted for the Dataflow CENSUS_CUBE1.
2. The same Constraint applies to both Provision Agreements.

The cascade rules elaborated above result as follows:

<u>DSD</u>

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

<u>Dataflow CENSUS_CUBE1</u>

1. 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).
2. Restricts the CAS codes to 003 and 004.

<u>Dataflow CENSUS_CUBE2</u>

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

Provision Agreements CENSUS_CUBE1_IT

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

Provision Agreements CENSUS_CUBE2_IT

1. Restricts the codes for the GEO Dimension to IT and its children.
2. Inherits the constraints from Dataflow CENSUS_CUBE2 for the CAS Dimension.
3. Inherits the AGE constraint applied at the level of the DSD.

The constraints are defined as follows:

DSD Constraint

```
<structure:ContentConstraint id="CONSTRAINT1" agencyID="CENSUSHUB" type="Allowed" >
    <common:Name>name</common:Name>
    <structure:ConstraintAttachment>
        <structure:DataStructure>
        <Ref agencyID="CENSUSHUB" id="CENSUS"></Ref>
        </structure:DataStructure>
    </structure:ConstraintAttachment>
    <structure:CubeRegion include="true">
            <!-- note uses the ability ot exclude values - i.e all values valid except this one -->
    <common:KeyValue id="AGE" include="false">
        <common:Value>001</common:Value>
    </common:KeyValue>
    </structure:CubeRegion>
</structure:ContentConstraint>
```

Dataflow Constraints


```xml
<structure:ContentConstraint id="CONSTRAINT2" agencyID="CENSUSHUB" type="Allowed" >
    <common:Name>name</common:Name>
    <structure:ConstraintAttachment>
    <structure:Dataflow>
        <Ref agencyID="CENSUSHUB" id="CENSUS_CUBE1"></Ref>
    </structure:Dataflow>
    </structure:ConstraintAttachment>
    <structure:CubeRegion include="true">
    <common:KeyValue id="AGE" include="true">
        <common:Value>002</common:Value>
        <common:Value>003</common:Value>
    </common:KeyValue>
    <common:KeyValue id="CAS">
        <common:Value>003</common:Value>
        <common:Value>004</common:Value>
    </common:KeyValue>
    </structure:CubeRegion>
</structure:ContentConstraint>
```



```xml
<structure:ContentConstraint id="CONSTRAINT3" agencyID="CENSUSHUB" type="Allowed" >
    <common:Name>name</common:Name>
    <structure:ConstraintAttachment>
    <structure:Dataflow>
        <Ref agencyID="CENSUSHUB" id="CENSUS_CUBE2"></Ref>
    </structure:Dataflow>
    </structure:ConstraintAttachment>
    <structure:CubeRegion include="true">
    <common:KeyValue id="CAS" include="true">
        <common:Value>TOT</common:Value>
        <common:Value>NAP</common:Value>
    </common:KeyValue>
    </structure:CubeRegion>
</structure:ContentConstraint>
```

<u>Provision Agreement Constraint</u>

```xml
<structure:ContentConstraint id="CONSTRAINT4" agencyID="CENSUSHUB" type="Allowed" >
    <common:Name>name</common:Name>
    <structure:ConstraintAttachment>
    <structure:ProvisionAgreement>
        <Ref agencyID="CENSUSHUB" id="CENSUS_CUBE1_IT"></Ref>
    </structure:ProvisionAgreement>
    <structure:ProvisionAgreement>
        <Ref agencyID="CENSUSHUB" id="CENSUS_CUBE2_IT"></Ref>
    </structure:ProvisionAgreement>
    </structure:ConstraintAttachment>
    <structure:CubeRegion include="true">
    <common:KeyValue id="GEO" include="true">
        <common:Value cascadeValues="true">IT</common:Value>
    </common:KeyValue>
    </structure:CubeRegion>
</structure:ContentConstraint>
```

9 Transforming between versions of SDMX

9.1 Scope

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

9.2 Groups and Dimension Groups

9.2.1 Issue

Version 2.1 introduces a more granular mechanism for specifying the relationship between a Data Attribute and the Dimensions to which the attribute applies. The technical construct for this is the Dimension Group. This Dimension Group has no direct equivalent in versions 2.0 and 1.0 and so the application transforming data from a version 2.1 data set to a version 2.0 or version 1.0 data set must decide to which construct the attribute value, whose Attribute is declared in a Dimension Group, should be attached. The closest construct is the “Series” attachment level and in many cases this is the correct construct to use.

However, there is one case where the attribute MUST be attached to a Group in the version 2.0 and 1.0 message. The conditions of this case are:

1. A Group is defined in the DSD with exactly the same Dimensions as a Dimension Group in the same DSD.

2. The Attribute is defined in the DSD with an Attribute Relationship to the Dimension Group. This attribute is NOT defined as having an Attribute Relationship to the Group.

9.2.2 Structural Metadata

If the conditions defined in 9.2.1are true then on conversion to a version 2.0 or 1.0 DSD (Key Family) the Component/Attribute.attachmentLevel must be set to “Group” and the Component/Attribute/AttachmentGroup” is used to identify the Group. Note that under rule(1) in 1.2.1 this group will have been defined in the V 2.1 DSD and so will be present in the V 2.0 transformation.

9.2.3 Data

If the conditions defined in 9.2.1are true then, on conversion from a 2.1 data set to a 2.0 or 1.0 dataset the attribute value will be placed in the relevant <Group>. If these conditions are not true then the attribute value will be placed in the <Series>.

9.2.4 Compact Schema

If the conditions defined in 9.2.1are true then the Compact Schema must be generated with the Group present and the Attribute(s) present in that group definition.

10 Validation and Transformation Language (VTL)

10.1 Introduction

The Validation and Transformation Language (VTL) supports the definition of Transformations, which are algorithms to calculate new data starting from already existing ones[6]. 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. 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.

[6] 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.

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.

10.2 References to SDMX artefacts from VTL statements

10.2.1 Introduction

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

The alias of a 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[7] or user defined operators[8] 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.

10.2.2 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[9] 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

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

[8] The `VTLMapping` are used also for User Defined Operators (UDO). Although UDOperators 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 UDOperator 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.

[9] 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 [10]
- 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 **SDMX prefix** 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, MeasureDimension, TimeDimension, PrimaryMeasure, DataAttribute, Concept, ConceptScheme, 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[11], 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, MeasureDimension, TimeDimension, PrimaryMeasure` 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 `ConceptScheme`, which is a maintainable class, the maintainedobject-id is the name of the `ConceptScheme` (`conceptScheme-id`);
- if the artefact is a `Codelist`, which is a maintainable class, the maintainedobject-id is the `Codelist` name (`codelist-id`).

---

[10] The container-object-id can repeat and may not be present.

[11] i.e., the artefact belongs to a maintainable class

The **maintainedobject-version** is the version of the maintained object which the artefact belongs to (for example, possible versions are 1.0, 2.1, 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

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`, `MeasureDimension`, `TimeDimension`, `PrimaryMeasure` 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 and their Agency is AG, would be written as[12]:

'urn:sdmx:org.sdmx.infomodel.datastructure.Dataflow=AG:DFR(1.0)' <-
'urn:sdmx:org.sdmx.infomodel.datastructure.Dataflow=AG:DF1(1.0)' +
'urn:sdmx:org.sdmx.infomodel.datastructure.Dataflow=AG:DF2(1.0)'

10.2.3 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`, `MeasureDimension`, `TimeDimension`, `PrimaryMeasure`, `DataAttribute`,

[12] 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.

- "`conceptscheme`" for the classes `Concept` and `ConceptScheme`
- "`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[13], the SDMX class can be deduced from the mapping rules between VTL and SDMX (see the section "Mapping between VTL and SDMX" hereinafter)[14].

- 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.[15] 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`, `MeasureDimension`, `TimeDimension`, `PrimaryMeasure`, `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 `ConceptScheme`, which is a maintainable class, the maintained object is the `conceptScheme-id` and obviously cannot be omitted;

[13] For the syntax of the VTL operators see the VTL Reference Manual

[14] 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`, `MeasureDimension`, `TimeDimension`, `PrimaryMeasure` or `DataAttribute`) can be deduced from the data structure of the SDMX `Dataflow` which the component belongs to.

[15] 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)

   - 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`, `ConceptScheme` and `Codelist` classes, while it exists and cannot be omitted for the artefacts belonging to the classes `Dimension`, `MeasureDimension`, `TimeDimension`, `PrimaryMeasure`, `DataAttribute` and `Concept`, as for them the object-id is the main identifier of the artefact

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)’ :=
‘urn:sdmx:org.sdmx.infomodel.datastructure.Dataflow=AG:DF1(1.0)’ +
‘urn:sdmx:org.sdmx.infomodel.datastructure.Dataflow=AG:DF2(1.0)’

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), which is[16]:

‘urn:sdmx:org.sdmx.infomodel.codelist.Codelist=AG:CL_FREQ(1.0)’

if the `Codelist` is referenced from a ruleset scheme belonging to the agency AG, omitting all the optional parts, the abbreviated reference would become simply[17]:

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).SECTOR’

[16] Single quotes are needed because this reference is not a VTL regular name.

[17] Single quotes are not needed in this case because CL_FREQ is a VTL regular name.

The corresponding fully abbreviated reference, if made from a transformation scheme 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[18]:

'DFR(1.0)' := 'DF1(1.0)' [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.

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).SECTOR'

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

CS1(1.0).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)' := between ('DF1(1.0)', 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).

10.2.4 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 would 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).

---

[18] The result DFR(1.0) is be equal to DF1(1.0) save that the component SECTOR is called SEC

10.2.5 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` or to a SDMX `ConceptScheme` (for SDMX measure dimensions), while a reference to a VTL Represented Variable becomes a reference to a SDMX `Concept`, assuming for it a definite representation[19].

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.[20]

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`, `ConceptScheme`, `Concept`) to can be deduced from the Ruleset signature.

10.3 Mapping between SDMX and VTL artefacts

10.3.1 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

---

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

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

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[21].

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 assignmentStatus 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).

10.3.2 General mapping of VTL and SDMX data structures

This section makes reference to the VTL "Model for data and their structure"[22] and the correspondent SDMX "Data Structure Definition"[23].

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).[24]

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`.

---

[21] If a calculated artefact is persistent, it needs a persistent definition, i.e. a SDMX definition in a SDMX environment. Also 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).

[22] See the VTL 2.0 User Manual

[23] See the SDMX 2.1 Section 2 – Information Model

[24] 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.

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, PrimaryMeasure and DataAttributes. In turn, a SDMX DimensionComponent can be a Dimension, a TimeDimension or a MeasureDimension. Correspondingly, in the SDMX implementation of the VTL, the VTL Identifiers can be (optionally) distinguished in three sub-classes (Simple Identifier, Time Identifier, Measure Identifier) even if such a distinction is not evidenced in the VTL IM.

However, a VTL Data Structure can have any number of Identifiers, Measures and Attributes, while a SDMX 2.1 DataStructureDefinition can have any number of Dimensions and DataAttributes but just one PrimaryMeasure[25]. This is due to a difference between SDMX 2.1 and VTL in the possible representation methods of the data that contain more measures.

As for SDMX, because the data structure cannot contain more than one measure component (i.e., the primaryMeasure), the representation of data having more measures is possible only by means of a particular dimension, called MeasureDimension, which is aimed at containing the name of the measure concepts, so that for each observation the value contained in the PrimaryMeasure component is the value of the measure concept reported in the MeasureDimension component.

Instead VTL allows either the method above (an identifier containing the name of the measure together with just one measure component) or a more generic method that consists in defining more measure components in the data structure, one for each measure.

Therefore for multi-measure data more mapping options are possible, as described in more detail in the following sections.

10.3.3 Mapping from SDMX to VTL data structures

10.3.3.1 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.

---

[25] The SDMX community is evaluating the opportunity of allowing more than one measure component in a DataStructureDefinition in the next SDMX major version.

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
`Time Dimension`	(Time) Identifier
`Measure Dimension`	(Measure) Identifier
`Primary Measure`	Measure
`Data Attribute`	Attribute

According to this method, the resulting VTL structures are always mono-measure (i.e., they have just one measure component) and their Measure is the SDMX `PrimaryMeasure`. Nevertheless, if the SDMX data structure has a `MeasureDimension`, which can convey the name of one or more measure concepts, such unique measure component can contain the value of more (conceptual) measures (one for each observation).

As for 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 generates one VTL data point.

10.3.3.2 Pivot Mapping

An alternative mapping method from SDMX to VTL is the **Pivot** mapping, which is different from the Basic method only for the SDMX data structures that contain a `MeasureDimension`, which are mapped to multi-measure VTL data structures.

The SDMX structures that do not contain a `MeasureDimension` are mapped like in the Basic mapping (see the previous paragraph).

The SDMX structures that contain a `MeasureDimension` are mapped as follows (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 Concept Cj of the SDMX `MeasureDimension` is mapped to a VTL Measure, having the same name as the SDMX Concept (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 `PrimaryMeasure` is not mapped to VTL as well (it disappears in the VTL Data Structure);
- A 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 Concept 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 Concept of the MeasureDimension separated by underscore; for example, if the SDMX DataAttribute is named DA and the possible concepts 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 & PrimaryMeasure	One Measure for each Concept of the SDMX Measure Dimension
DataAttribute not depending on the MeasureDimension	Attribute
DataAttribute depending on the MeasureDimension	One Attribute for each Concept of the SDMX Measure Dimension

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 Concept 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 Concept 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 PrimaryMeasure 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

10.3.3.3 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.

10.3.4 Mapping from VTL to SDMX data structures

10.3.4.1 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.

This mapping method cannot be applied for SDMX 2.1 if the VTL data structure has more than one measure component, given that the SDMX 2.1 DataStructureDefinition allows just one measure component (the PrimaryMeasure). In this case it becomes mandatory to specify a different mapping method through the VtlMappingScheme and VtlDataflowMapping classes.[26]

Please note that the VTL measures can have any name while in SDMX 2.1 the MeasureComponent has the mandatory name “obs_value”, therefore the name of the VTL measure name must become “obs_value” in SDMX 2.1.

Mapping table:

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

If the distinction between simple identifier, time identifier and measure identifier is not maintained in the VTL environment, the classification between Dimension, TimeDimension and MeasureDimension 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 (AttributeRelationships referred to the PrimaryMeasure), unless some other information about their AttributeRelationship is available.

Note that the basic mappings in the two directions (from SDMX 2.1 to VTL 2.0 and vice-versa) are (almost completely) reversible. In fact, if a SDMX 2.1 structure is mapped to a VTL structure and then the latter is mapped back to SDMX 2.1, the resulting data structure is like the original one (apart for the AttributeRelationship, that can be different if the original SDMX 2.1 structure contains DataAttributes that are not at observation level). In reverse order, if a VTL 2.0 mono-measure structure is mapped to SDMX 2.1 and then the latter is mapped back to VTL 2.0, the original data structure is obtained (apart from the name of the VTL measure, that in SDMX 2.1 must become “obs_value”).

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

10.3.4.2 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

[26] If future SDMX major versions will allow multi-measures data structures, this method is expected to become applicable even if the VTL data structure has more than one measure

VTL structure). For such VTL structures, in fact, the basic method cannot be applied, given that by maintaining the data structure unchanged the resulting SDMX data structure would have more than one measure component, which is not allowed by SDMX 2.1 (it allows just one measure component, the `PrimaryMeasure`, called "obs_value").

The multi-measures VTL structures have not a Measure Identifier (because the Measures are separate components) and need to be converted to SDMX `dataflows` having an added `MeasureDimension` which disambiguates the multiple measures, and an added `PrimaryMeasure`, in which the measures' values are maintained.

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 `PrimaryMeasure` component called "obs_value" is added to the SDMX `DataStructure`;
- each VTL Measure is mapped to a Concept 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 Measure Concept for each VTL measure component) & `PrimaryMeasure`
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 `Dimensions`, for all the observations of the set above

- the name of the j^{th} VTL measure (e.g. "Cj") becomes the value of the SDMX `MeasureDimension` of the j^{th} observation of the set (i.e. the `Concept` Cj)
- the value of the j^{th} VTL measure becomes the value of the SDMX `PrimaryMeasure` 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" would be added to the SDMX `DataStructure` and would have just one possible measure concept, corresponding to the unique VTL measure. The original VTL measure component would not become a `Component` in the SDMX data structure. The value of the VTL measure would be assigned to the SDMX `PrimaryMeasure` called "obs_value".

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

10.3.4.3 From VTL Measures to SDMX Data Attributes

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 one VTL measure in the SDMX `primaryMeasure` and all the other VTL Measures in SDMX `DataAttributes`. This method is called M2A ("M2A" stands for "Measures to `DataAttributes`").

When applied to <u>mono-measure VTL structures</u> (having one Measure component), the M2A method behaves like the Basic mapping (the VTL Measure component becomes the SDMX primary measure "obs_value", there is no additional VTL measure to be converted to SDMX DataAttribute). Therefore the mapping table is the same as for the Basic method:

VTL	SDMX
(Simple) Identifier	`Dimension`
(Time) Identifier	`TimeDimension`
(Measure) Identifier (if any)	`MeasureDimension`
Measure	`PrimaryMeasure`
Attribute	`DataAttribute`

For multi-measure VTL structures (having more than one Measure component), one VTL Measure becomes the SDMX `PrimaryMeasure` while the other VTL Measures maintain their names and values but assume the role of `DataAttribute` in SDMX. The choice of the VTL Measure that correspond to the SDMX `PrimaryMeasure` is left to the definer of the SDMX data structure definition.

Taking into account that the multi-measure VTL structures do not have a measure identifier, the mapping table is the following:

VTL	SDMX
(Simple) Identifier	`Dimension`
(Time) Identifier	`TimeDimension`
One of the Measures	`PrimaryMeasure`
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 `assignmentStatus`, which does not exist in VTL, the `attributeRelationship` for the `DataAttributes` and so on. In particular, the `primaryMeasure` of the SDMX 2.1 DSD must be called "obs_value" and must be one of the VTL Measures, chosen by the DSD definer.

10.3.5 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` `(besides possible mappings to artefacts other than dataflows).`

10.3.6 Mapping dataflow subsets to distinct VTL data sets[27]

Until now it as been assumed to map one SMDX `Dataflow` to one VTL dataset and vice-versa. This mapping one-to-one is not mandatory according to VTL because a

---

[27] The kind of mapping explained here works in combination with a SDMX specific naming convention that requires pre-processing before parsing the VTL expressions. As highlighted below, the identifiers of the VTL datasets are a shortcut of some specific VTL operators applied to the SDMX Dataflows. This is not safe to use outside an SDMX context, as the naming convention may have no meaning there.

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.[28]

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.[29]

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) 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 dimension components on which the mapping will be based, in a given order.[30] Following the example above, imagine that the user declares the `dimensions` INDICATOR and COUNTRY.
- The VTL dataset is given a name using a special notation also called “ordered concatenation” and composed of the following parts:

---

[28] 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 dimension components of the `Dataflow` except the time `Dimension`. 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.

[29] 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 datasets.

[30] 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 datasets is not used.

- 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);
- a slash ("/") as a separator; [31]
- The reference to a specific part of the SDMX `dataflow` above, expressed as the concatenation of the values that the SDMX dimensions declared above must have, separated by dots (".") and written in the order in which these dimensions are defined[32] . For example POPULATION.USA would mean that such a VTL dataset 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)/*INDICATORvalue*.*COUNTRYvalue*'

Where DF(1.0) 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)/POPULATION.USA'

In particular, this is the VTL dataset that contains all the observations of the `dataflow` DF(1.0) 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 VTL datasets are operand of VTL transformations, instead the mapping from VTL to SDMX happens when the VTL datasets are result of VTL transformations[33] and 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.

[31] As a consequence of this formalism, a slash in the name of the VTL dataset assumes the specific meaning of separator between the name of the `Dataflow` and the values of some of its `Dimensions`.

[32] 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.

[33] 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)) need to be mapped to distinct VTL datasets that are operand of some VTL transformations.

As already said, each VTL dataset 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)/POPULATION.USA' would contain all the observations of DF1(1.0) having INDICATOR = POPULATION and COUNTRY = USA.

In order to obtain the data structure of these VTL datasets from the SDMX one, it is assumed that the SDMX dimensions 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 datasets[34]. After that, the mapping method from SDMX to VTL specified for the dataflow DF1(1.0) is applied (i.e. basic, pivot …).

In the example above, for all the datasets of the kind 'DF1(1.0)/*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 datasets (i.e. of the kind 'DF1(1.0)/ *INDICATORvalue*.*COUNTRYvalue*') can be obtained also by applying the VTL operator "**sub**" (subspace) to the `dataflow` DF1(1.0), like in the following VTL expression:

```
'DF1(1.0)/POPULATION.USA' :=
DF1(1.0) [ sub  INDICATOR="POPULATION", COUNTRY="USA" ];

'DF1(1.0)/POPULATION.CANADA' :=
DF1(1.0) [ 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 datasets 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`. [35]

In the direction from SDMX to VTL it is allowed to omit the value of one or more Dimensions on which the mapping is based, but maintaining all the separating dots

---

[34] If these dimensions would not be dropped, taking into account that the typical binary VTL operations at dataset level (+, -, *, / and so on) are executed on the observations having matching identifiers, the VTL datasets resulting from this kind of mapping would have non-matching values for the mapping dimensions (e.g. POPULATION and COUNTRY), therefore 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).

[35] 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 and, 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.

(therefore it may happen to find two or more consecutive dots and dots 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)/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) 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)/POPULATION.' :=
DF1(1.0) [sub INDICATOR="POPULATION"];

Therefore the VTL dataset 'DF1(1.0)/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 the mapping direction from VTL to SDMX.

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) 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) that have different combinations of values for INDICATOR and COUNTRY:

- each part is calculated as a VTL derived dataset, result of a dedicated VTL transformation; [36]
- the data structure of all these VTL datasets has the TIME_PERIOD identifier and does not have the INDICATOR and COUNTRY identifiers.[37]

Under these hypothesis, such derived VTL datasets can be mapped to DF2(1.0) by declaring the `Dimensions` INDICATOR and COUNTRY as mapping dimensions[38].

The corresponding VTL transformations, assuming that the result needs to be persistent, would be of this kind: [39]

---

[36] 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..

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

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

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

‘DF2(1.0)/*INDICATORvalue*.*COUNTRYvalue*’ <- expression

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

```
‘DF2(1.0)/GDPPERCAPITA.USA’       <-  expression11;
‘DF2(1.0)/GDPPERCAPITA.CANADA’    <-  expression12;
… … …
‘DF2(1.0)/POPGROWTH.USA’          <-  expression21;
‘DF2(1.0)/POPGROWTH.CANADA’       <-  expression22;
… … …
```

As said, it is assumed that these VTL derived datasets 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)/GDPPERCAPITA.USA’	GDPPERCAPITA	USA
‘DF2(1.0)/GDPPERCAPITA.CANADA’	GDPPERCAPITA	CANADA
… … …		
‘DF2(1.0)/POPGROWTH.USA’	POPGROWTH	USA
‘DF2(1.0)/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)), 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)/GDPPERCAPITA.USA’
                          [calc  identifier INDICATOR := ”GDPPERCAPITA”,
                                 identifier  COUNTRY := ”USA”];

DF2bis_GDPPERCAPITA_CANADA :=  ‘DF2(1.0)/GDPPERCAPITA.CANADA’
                          [calc  identifier INDICATOR:=”GDPPERCAPITA”,
                                 identifier COUNTRY:=”CANADA”];
   … … …
DF2bis_POPGROWTH_USA       :=  ‘DF2(1.0)/POPGROWTH.USA’
                          [calc  identifier INDICATOR := ”POPGROWTH”,
                                 identifier  COUNTRY := ”USA”];
DF2bis_POPGROWTH_CANADA’    :=  ‘DF2(1.0)/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)[40], 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. [41]

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).

10.3.7 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 `Dimension` or a `PrimaryMeasure` or a `DataAttribute`) belonging to one specific `Dataflow`[42]
Represented Variable	**`Concept`** with a definite `Representation`

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

[41] 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.

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

Value Domain	**`Representation`** (see the Structure Pattern in the Base Package)
Enumerated Value Domain / Code List	**`Codelist`** (for enumerated `Dimension`, `PrimaryMeasure`, `DataAttribute`) or **`ConceptScheme`** (for `MeasureDimension`)
Code	**`Code`** (for enumerated `Dimension`, `PrimaryMeasure`, `DataAttribute`) or **`Concept`** (for `MeasureDimension`)
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 to a valid **`value`** (for non-enumerated `Representations`) or to a **`Concept`** (for `MeasureDimension`)
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[43], while the SDMX `Concepts` can have different `Representations` in different `DataStructures`.[44] 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 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.

10.4 Mapping between SDMX and VTL Data Types

10.4.1 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

---

[43] 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.

[44] 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`.

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:

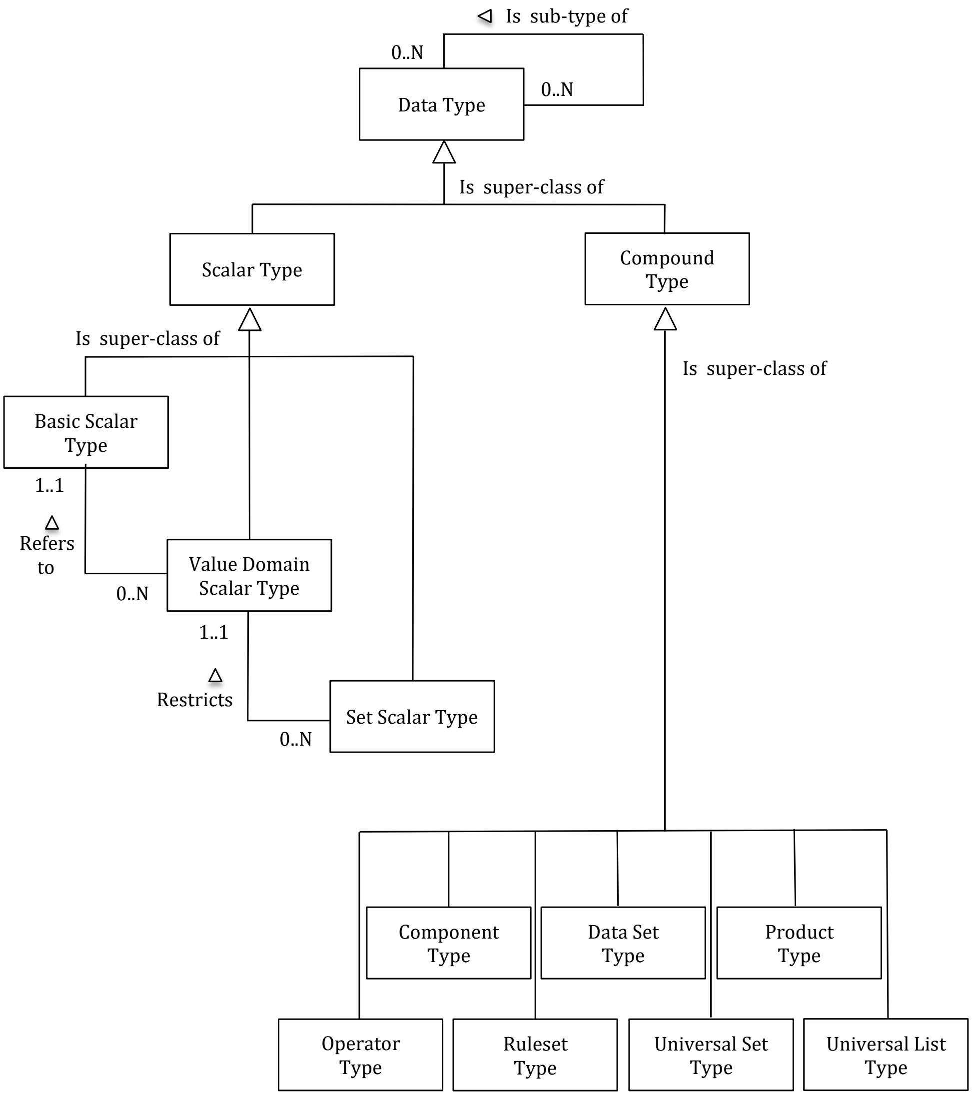


Figure 12 – VTL Data Types

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):

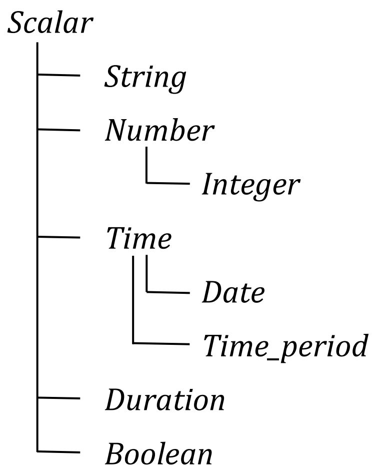


Figure 13 – VTL Basic Scalar Types

10.4.2 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.

10.4.3 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:	**boolean**

{true, false})	
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 (YYYY-Www – 7 day period; following ISO 8601 definition of a week in a year)	**time_period**
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	Metadata type – not applicable
KeyValues	Metadata type – not applicable
IdentifiableReference	Metadata type – not applicable
DataSetReference	Metadata type – not applicable
AttachmentConstraintReference	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).

10.4.4 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[45].

10.4.5 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`.

10.4.6 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.

---

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

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.

11 Annex I: How to eliminate extra element in the .NET SDMX Web Service

11.1 Problem statement

For implementing an SDMX compliant Web Service the standardised WSDL file should be used that describes the expected request/response structure. The request message of the operation contains a wrapper element (e.g. “GetGenericData”) that wraps a tag called “GenericDataQuery”, which is the actual SDMX query XML message that contains the query to be processed by the Web Service. In the same way the response is formulated in a wrapper element “GetGenericDataResponse”.

As defined in the SOAP specification, the root element of a SOAP message is the Envelope, which contains an optional Header and a mandatory Body. These are illustrated below along with the Body contents according to the WSDL:

XML

```xml
<SOAP-ENV:Envelope

   <SOAP-ENV:Body>

      <GetGenericData>

         <sdmx:GenericDataQuery>

            ...

         </sdmx:GenericDataQuery>

      </GetGenericData>

   </SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

The problem that initiated the present analysis refers to the difference in the way SOAP requests are when trying to implement the aforementioned Web Service in .NET framework.

Building such a Web Service using the .NET framework is done by exposing a method (i.e. the getGenericData in the example) with an XML document argument (lets name it “Query”). **The difference that appears in Microsoft .Net implementations is that there is a need for an extra XML container around the SDMX GenericDataQuery.** This is the expected behavior since the framework is let to publish automatically the Web Service as a remote procedure call, thus wraps each parameter into an extra element. The .NET request is illustrated below:

XML

```xml
<SOAP-ENV:Envelope
```

```
  <SOAP-ENV:Body>
    <GetGenericData>
      <Query>        <!-- MS .Net implementation -->
        <GenericDataQuery>
          ...
        </GenericDataQuery>
      </Query>      <!-- MS .Net implementation -->
    </GetGenericData>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

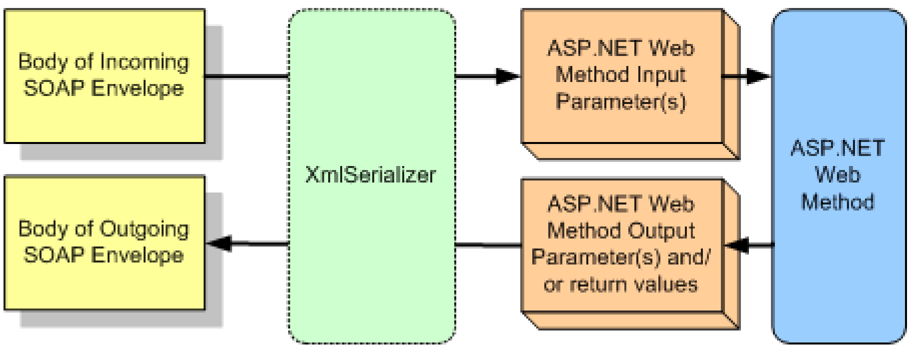
Furthermore this extra element is also inserted in the automatically generated WSDL from the framework. Therefore this particularity requires custom clients for the .NET Web Services that is not an interoperable solution.

11.2 Solution

The solution proposed for conforming the .NET implementation to the envisioned SOAP requests has to do with the manual intervention to the serialisation and deserialisation of the XML payloads. Since it is a Web Service of already prepared XML messages requests/responses this is the indicate way so as to have full control on the XML messages. This is the way the Java implementation (using Apache Axis) of the SDMX Web Service has adopted.

As regards the .NET platform this is related with the usage of **XmlAnyElement** parameter for the .NET web methods.

Web methods use XmlSerializer in the .NET Framework to invoke methods and build the response.



The XML is passed to the XmlSerializer to de-serialize it into the instances of classes in managed code that map to the input parameters for the Web method. Likewise, the output parameters and return values of the Web method are serialized into XML in order to create the body of the SOAP response message.

In case the developer wants more control over the serialization and de-serialization process a solution is represented by the usage of **XmlElement** parameters. This offers the opportunity of validating the XML against a schema before de-serializing it, avoiding de-serialization in the first place, analyzing the XML to determine how you want to de-serialize it, or using the many powerful XML APIs that are available to deal with the XML directly. This also gives the developer the control to handle errors in a particular way instead of using the faults that the XmlSerializer might generate under the covers.

In order to control the de-serialization process of the XmlSerializer for a Web method, **XmlAnyElement** is a simple solution to use.

To understand how the **XmlAnyElement** attribute works we present the following two web methods:

C#

```
// Simple Web method using XmlElement parameter

[WebMethod]

public void SubmitXml(XmlElement input)

{ return; }
```

In this method the **input** parameter is decorated with the **XmlAnyElement** parameter. This is a hint that this parameter will be de-serialized from an **xsd:any** element. Since the attribute is not passed any parameters, it means that the entire XML element for this parameter in the SOAP message will be in the Infoset that is represented by this **XmlElement** parameter.

C#

```
// Simple Web method...using the XmlAnyElement attribute

[WebMethod]

public void SubmitXmlAny([XmlAnyElement] XmlElement input)

{ return; }
```

The difference between the two is that for the first method, **SubmitXml**, the XmlSerializer will expect an element named **input** to be an immediate child of the

SubmitXml element in the SOAP body. The second method, **SubmitXmlAny**, will not care what the name of the child of the **SubmitXmlAny** element is. It will plug whatever XML is included into the input parameter. The message style from ASP.NET Help for the two methods is shown below. First we look at the message for the method without the **XmlAnyElement** attribute.

XML

```
<?xml version="1.0" encoding="utf-8"?>

<soap:Envelope

    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

    xmlns:xsd="http://www.w3.org/2001/XMLSchema"

    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">

  <soap:Body>

    <SubmitXml xmlns="http://msdn.microsoft.com/AYS/XEService">

      <input>xml</input>

    </SubmitXml>

  </soap:Body>

</soap:Envelope>
```

Now we look at the message for the method that uses the **XmlAnyElement** attribute.

XML

```
<?xml version="1.0" encoding="utf-8"?>

<!-- SOAP message for method using XmlAnyElement -->

<soap:Envelope

    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

    xmlns:xsd="http://www.w3.org/2001/XMLSchema"

    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">

  <soap:Body>

    <SubmitXmlAny xmlns="http://msdn.microsoft.com/AYS/XEService">

      Xml

    </SubmitXmlAny>

  </soap:Body>
```

```
</soap:Envelope>
```

The method decorated with the **XmlAnyElement** attribute has one fewer wrapping elements. Only an element with the name of the method wraps what is passed to the **input** parameter.

For more information please consult:

http://msdn.microsoft.com/en-us/library/aa480498.aspx

Furthermore at this point the problem with the different requests has been solved. However there is still the difference in the produced WSDL that has to be taken care. The automatic generated WSDL now doesn't insert the extra element, but defines the content of the operation wrapper element as "xsd:any" type.

XML

```xml
<xs:element name="GetGenericData">
  <xs:complexType>
    <xs:sequence>
      <xs:any minOccurs="0" maxOccurs="1" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Without a common WSDL still the solution doesn't enforce interoperability. In order to "fix" the WSDL, there two approaches. The first is to intervene in the generation process. This is a complicated approach, compared to the second approach, which overrides the generation process and returns the envisioned WSDL for the SDMX Web Service.

This is done by redirecting the request to the "/Service?WSDL" to the envisioned WSDL stored locally into the application. To do this, from the project add a "Global Application Class" item (.asax file) and override the request in the "Application_BeginRequest" method. This is demonstrated in detail in the next section.

This approach has the disadvantage that for each deployment the WSDL end point has to be changed to reflect the current URL. However this inconvenience can be easily eliminated if a developer implements a simple rewriting module for changing the end point to the one of the current deployment.

11.3 Applying the solution

In the context of the SDMX Web Service, applying the above solution translates into the following:

C#

```csharp
[return: XmlAnyElement]

public XmlDocument GetGenericData([XmlAnyElement]XmlDocument Query)

{ return; }
```

The SOAP request/response will then be as follows:

GenericData Request

XML

```xml
<?xml version="1.0" encoding="utf-8"?>

<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">

  <soap:Body>

    <GetGenericData xmlns="http://www.sdmx.org/resources/webservices">

      Xml

    </GetGenericData>

  </soap:Body>

</soap:Envelope>
```

GenericData Response

XML

```xml
<?xml version="1.0" encoding="utf-8"?>

<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">

  <soap:Body>

    <GetGenericDataResponse
xmlns="http://www.sdmx.org/resources/webservices">

      Xml

    </GetGEnericDataResponse>

  </soap:Body>

</soap:Envelope>
```

For overriding the automatically produced WSDL, in the solution explorer right click the project and select “Add” -> “New item…”. Then select the “Global Application Class”. This will create “.asax” class file in which the following code should replace the existing empty method:

C#

```
protected void Application_BeginRequest(object sender, EventArgs e)

{

  System.Web.HttpApplication app = (System.Web.HttpApplication)sender;

  if (Request.RawUrl.EndsWith("/Service1.asmx?WSDL"))

  {

    app.Context.RewritePath("/SDMX_WSDL.wsdl", false);

  }

}
```

The SDMX_WSDL.wsdl should reside in the in the root directory of the application. After applying this solution the returned WSDL is the envisioned. Thus in the request message definition contains:

XML

```xml
<xs:element name="GetGenericData">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="sdmx:GenericQueryData"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```