SDMX Technical Working Group

VTL Task Force

Validation & Transformation Language

Part 1 - General Description

Version 1.0

February 2015

30

Foreword

The SDMX Technical Working Group is pleased to present the version 1.0 of the Validation and Transformation Language, in short VTL.

The work on VTL was launched at the end of 2012 by the SDMX Secretariat. SDMX already has a package for transformations and expressions which is present in the information model, although a specific language does not yet exist. To make this framework fully operational, a standard "language" for defining validation and transformation rules (set of operators, their syntax and semantics) should be adopted, appropriate IT formats for exchanging such rules and related metadata should be introduced, and the web services to store and retrieve them should be designed.

A task force was put in place, composed of members of SDMX, DDI and GSIM communities and the work started in summer 2013. The intention was to provide a language which is usable by statisticians to express logical validation rules and transformations on data, whether described as dimensional tables or as unit-record data. The assumption is that this logical formalization of validation and transformation rules would be converted into specific programming languages for execution (SAS, R, Java, SQL, etc.) but would provide a "neutral" expression at business level of the processing taking place, against which various implementations can be mapped. Experience with existing examples suggests that this goal would be attainable.

An important point that emerged is that several standards are interested in such a language. However, each standard operates on its model artefacts and produces artefacts within the same model (property of closure). To cope with this, VTL has been built upon a very basic information model, taking the common parts of GSIM, SDMX and DDI, mainly using artefacts from GSIM 1.1, somewhat simplified and with some additional detail. This way the existing standards (SDMX, DDI, others) may adopt VTL by mapping their information model against the VTL one. Therefore, although a work-product of SDMX, the VTL language will be usable also with other standards.

The VTL 1.0 package includes:

a) Part 1, highlighting the main characteristics of VTL, its core assumptions and the information model the language is based on;

b) Part 2, containing the full library of operators ordered by category, including examples; this first version can support validation and basic compilation needs. Future versions will include more features related to transformation of data.

c) BNF notation (Backus-Naur Form) which is the technical notation to be used as a test bed for all the examples throughout the document.

The present document (part 1) contains the general part, highlighting the main characteristics of VTL, its core assumptions and the information model VTL is based on.

The latest version of the VTL is freely available online at www.sdmx.org.

Acknowledgements

This publication has been prepared thanks to the collective input of experts from Bank of Italy, Bank for International Settlements (BIS), European Central Bank (ECB), Eurostat, ILO, ISTAT-Italy, OECD, Statistics Netherlands, and UNESCO. Other experts from the SDMX Technical Working Group, the SDMX Statistical Working Group and the DDI initiative were also consulted and participated in reviewing the documentation.

The list of contributors and reviewers includes the following experts: Sami Airo, Foteini Andrikopoulou, David Barraclough, Luigi Bellomarini, Marc Bouffard, Maurizio Capaccioli, Vincenzo Del Vecchio, Fabio Di Giovanni, Jens Dossé, Heinrich Ehrmann, Bryan Fitzpatrick, Arofan Gregory, Edgardo Greising, Angelo Linardi, Chris Nelson, Stratos Nikoloutsos, Marco Pellegrino, Michele Romanelli, Juan Alberto Sanchez, Angel Simon Delgado, Daniel Suranyi, Olav ten Bosch, Laura Vignola, Nikolaos Zisimos.

Feedback and suggestions for improvement are encouraged and can be sent to the SDMX Technical Working Group (twg@sdmx.org).

Table of contents

Introduction

This document presents the Validation and Transformation Language (aka VTL).

The purpose of the VTL is to allow a formal and standard definition of algorithms to validate statistical data and calculate derived data.

The VTL development is organized in a first phase aimed to allow the formalisation of the data validation algorithms and in following phases aimed to tackle more complex algorithms for data compilation. In fact, the assessment of business cases showed that the majority of the institutions ascribes a higher priority to a standard language for supporting the validation processes and in particular to the possibility of sharing validation rules with the respective data providers, in order to specify the quality requirements and allow validation also before provision.

This document is the outcome of the first phase and therefore presents a first version of the VTL primarily oriented to support the data validation. However, because the features needed for the validation include simple calculations, this first version of the VTL can also support basic compilation needs. In general, validation is assumed to be a particular case of transformation; therefore, the term "Transformation" is meant to be more general and to include validation as well.

The main categories of operators included in this version of the VTL syntax are:

General	(e.g. assignment, data access, data storage …)
String	(e.g. substring, concatenation, length …)
Mathematical	(e.g. +, -, *, /, round, absolute value …)
Boolean	(e.g. and, or, not …)
Relational	(e.g. selection, union, intersection, merge …)
Statistical	(e.g. minimum, maximum, aggregation …)
Validation	(e.g. of value domains, references, figures …)
Conditional	(e.g. if-then-else …)

Although the VTL is developed under the umbrella of the SDMX initiative, DDI and GSIM users may also be highly interested in adopting a language for validation and transformation. In particular, organizations involved in the SDMX, DDI and GSIM communities and in the High-Level Group for the modernisation of statistical production and services (HLG) expressed their wish of having a unique language, usable in SDMX, DDI and GSIM.

Accordingly, the working group for the VTL development includes representatives of institutions involved in the DDI and GSIM initiatives and there has been agreement on the objective of adopting a common language, applicable to SDMX as well as to DDI and GSIM, in the hope of avoiding the risk of having diverging variants.

As a consequence, the VTL is designed as a language relatively independent of the details of SDMX, DDI and GSIM. It is based on an independent information model (IM), made of the very basic artefacts common to these standards. Other models, like SDMX, DDI, GSIM, can inherit the VTL language by (unequivocally) mapping their artefacts to the ones of the VTL IM.

Structure of the document

The first part of the document is dedicated to the description of the general characteristics of the VTL.

The following part describes the Information Model on which the language is based. In particular, it describes the model of the data artefacts that the language is aimed to validate and transform, the model of the variables and value domains used in the data artefacts and the model of the transformations.

A third part clarifies some general features of the language (i.e. the core assumptions of the VTL), such as the types of artefacts involved in the transformations, the general rules for the operations on the data sets, the methods for referencing the data sets to be operated on, and the general conventions for the grammar of the language.

A final part highlights some issues related to the governance of VTL developments and to future work, following a number of comments, suggestions and other requirements which were submitted to the task-force in order to enhance the current VTL 1.0 package.

A short annex gives some background information about the BNF (Backus-Naur Form) syntax which has been used for providing a context-free representation of VTL. The Extended BNF (EBNF) representation is part of the VTL 1.0 package available at www.sdmx.org.

General characteristics of the VTL

This section lists and briefly illustrates some general high-level characteristics of the validation and transformation language. They have been discussed and shared as requirements for the language in the VTL working group since the beginning of the work and have been taken into consideration for the design of the language.

User orientation

- ⇨ The language is designed for users without information technology (IT) skills, who should be able to define calculations and validations independently, without the intervention of IT personnel;
  - The language is based on a "user" perspective and a "user" information model (IM) and not on possible IT perspectives (and IMs)
  - As much as possible, the language is able to manipulate statistical data at an abstract/conceptual level, independently of the IT representation used to store or exchange the data observations (e.g. files, tables, xml tags), so operating on abstract (from IT) model artefacts to produce other abstract (from IT) model artefacts
  - It references IM objects and does not use direct references to IT objects
- ⇨ The language is intuitive and friendly (users should be able to define and understand validations and transformations as easily as possible), so the syntax is:
  - Designed according to mathematics, which is a universal knowledge;
  - Expressed in English to be shareable in all countries;
  - As simple, intuitive and self-explanatory as possible;
  - Based on common mathematical expressions, which involve "operands" operated on by "operators" to obtain a certain result;
  - Designed with minimal redundancies (e.g. possibly avoiding operators specifying the same operation in different ways without concrete reasons).
- ⇨ The language is oriented to statistics, and therefore it is capable of operating on statistical objects and envisages the operators needed in the statistical processes and in particular in the data validation phases, for example:
  - Operators for data validations and edit;
  - Operators for aggregation, including according to hierarchies;
  - Operators for dimensional processing (e.g. projection, filter);
  - At a later stage, operators for time series processing (e.g. time shift, change of periodicity, moving average, seasonal adjustment, correlation) operators for statistics (e.g. aggregation, mean, median, percentiles, variance, indexes, correlation, sampling, inference, estimation);

Integrated approach

- ➪ The language is independent of the statistical domain of the data to be processed;
  - VTL has no dependencies on the subject matter (the data content);
  - VTL is able to manipulate statistical data in relation to their structure.
- ➪ The language is suitable for the various typologies of data of a statistical environment (for example dimensional data, survey data, registers data, micro and macro, quantitative and qualitative) and is supported by an information model (IM) which covers these typologies;
  - The IM allows the representation of the various typologies of data of a statistical environment at a conceptual/logical level (in a way abstract from IT and from the physical storage);
  - The various typologies of data are described as much as possible in an integrated way, by means of common IM artefacts for their common aspects;
  - The principle of the Occam's razor is applied as an heuristic principle in designing the conceptual IM, so keeping everything as simple as possible or, in other words, unifying the model of apparently different things as much as possible.
- ➪ The language (and its IM) is independent of the phases of the statistical process and usable in any one of them;
  - Operators are designed to be independent of the phases of the process, their syntax does not change in different phases and is not bound to some characteristic restricted to a specific phase (operators' syntax is not aware of the phase of the process);
  - In principle, all operators are allowed in any phase of the process (e.g. it is possible to use the operators for data validation not only in the data collection but also, for example, in data compilation for validating the result of a compilation process; similarly it is possible to use the operators for data calculation, like the aggregation, not only in data compilation but also in data validation processes);
  - Both collected and calculated data are equally permitted as inputs of a calculation, without changes in the syntax of the operators/expression;
  - Collected and calculated data are represented (in the IM) in a homogeneous way with regards to the metadata needed for calculations.
- ➪ The language is designed to be applied not only to SDMX but also to other standards;
  - VTL, like any consistent language, relies on a specific information model, as it operates on the VTL IM artefacts to produce other VTL IM artefacts. In principle, a language cannot be applied as-is to another information model (e.g. SDMX, DDI, GSIM); this possibility exists only if there is a unambiguous correspondence between the artefacts of those information models and the VTL IM (that is if their artefacts correspond to the same mathematical notion);
  - The goal of applying the language to more models/standards is achieved by using a very simple, generic and conceptual Information Model (the VTL IM),

  - and mapping this IM to the models of the different standards (SDMX, DDI, GSIM, …); to the extent that the mapping is straightforward and unambiguous, the language can be inherited by other standards (with the proper adjustments);
  - To achieve an unambiguous mapping, the VTL IM is deeply inspired by the GSIM IM and uses the same artefacts when possible[1]; in fact, GSIM is designed to provide a formal description of data at business level against which other information models can be mapped; moreover, loose mappings between GSIM and SDMX and between GSIM and DDI are already available[2]; a very small subset of the GSIM artefacts is used in the VTL IM in order to keep the model and the language as simple as possible (Occam's razor principle); these are the artefacts strictly needed for describing the data involved in Transformations, their structure and the variables and value domains;
  - GSIM artefacts are supplemented when needed, with other artefacts that are necessary for describing calculations; in particular, the SDMX model for Transformations is used;
  - As mentioned above, the definition of the VTL IM artefacts is based on mathematics and is expressed at an abstract user level.

Active role for processing

- ⇨ The language is designed to possibly drive in an active way the execution of the calculations (in addition to documenting them)
- ⇨ For the purpose above, it is possible either to implement a calculation engine that interprets the VTL and operates on the data or to rely on already existing IT tools (this second option requires a translation from the VTL to the language of the IT tool to be used for the calculations)
- ⇨ The VTL grammar is being described formally using the universally known Backus Naur Form notation (BNF), because this allows the VTL expressions to be easily defined and processed; the formal description allow the expressions:
  - To be automatically parsed (against the rules of the formal grammar); on the IT level, this requires the implementation of a parser that compiles the expressions and checks their correctness;
  - To be automatically translated from the VTL to the language of the IT tool to be used for the calculation; on the IT level, this requires the implementation of a proper translator;
  - To be automatically translated from one VTL version to another, e.g. following an upgrade of the VTL syntax; on the IT level, this requires the implementation of a proper translator also.

[1] See the next section (VTL Information Model) and the section "Relations with the GSIM Information model"

[2] See at: http://www1.unece.org/stat/platform/display/gsim/GSIM+and+standards;

- ➪ The inputs and the outputs of the calculations and the calculations themselves are artefacts of the IM
  - This is a basic property of any robust language because it allows calculated data to be operands of further calculations;
  - If the artefacts are persistently stored, their definition is persistent as well; if the artefacts are non-persistently stored (used only during the calculation process like input from other systems, intermediate results, external outputs) their definition can be non-persistent;
  - Because the definition of a calculation needs the data structure definition of its input artefacts, the latter must be available when the calculation is defined;
  - The VTL is designed to make the data structure of the output of a calculation deducible from the calculation algorithm and from the data structure of the operands (this feature ensures that the calculated data can be defined according to the IM and can be used as operands of further calculations);
  - In the IT implementation, it is advisable to automate (as much as possible) the structural definition of the output of a calculation, in order to enforce the consistency of the definitions and avoid unnecessary overheads for the definers.
- ➪ The VTL and its information model make it possible to check automatically the overall consistency of the definition of the calculations, including with respect to the artefact of the IM, and in particular to check:
  - the correctness of the expressions with respect to the syntax of the language
  - the integrity of the expressions with respect to their input and output artefacts and the corresponding structures and properties (for example, the input artefacts must exist, their structure components referenced in the expression must exist, qualitative data cannot be manipulated through quantitative operators, and so on)
  - the consistency of the overall graph of the calculations (for example, there should not be cycles in the sequence of calculations in order to avoid that the result of a calculation goes as input to the same calculation, so producing unpredictable and erroneous results);

Independence of IT implementation

- ➪ According to the “user orientation” above, the language is designed so that users are not required to be aware of the IT solution;
  - To use the language, the users need to know only the abstract view of the data and calculations and do not need to know the aspects of the IT implementation, like the storage structures, the calculation tools and so on.
- ➪ The language is not oriented to a specific IT implementation and permits many possible different implementations (this property is particularly important in order to allow different institutions to rely on different IT environments and solutions);

  - On the technical level, the connection between the user layer and the IT layer is left to the specific IT implementations;
  - The VTL approach favours effective IT implementations that decouple the user layer and the IT layer.
- ➩ The language does not require the awareness of the physical data structure; the operations on the data are specified according to the conceptual/logical structure, and so are independent of the physical structure; this ensures that the physical structure may change without necessarily affecting the conceptual structure and the user expressions;
  - Data having the same conceptual/logical structure may be accessed using the same statements, even if they have different IT structures;
  - The VTL provides for commands for data store and retrieve at a conceptual/logical level; the mapping and the conversion between the conceptual and the physical structures of the data is left to the IT implementation (and users need not be aware of it);
  - By mapping the user and the IT data structures, the IT implementations can make it possible to store/retrieve data in/from different IT data stores (e.g. relational databases, dimensional databases, xml files, spread-sheets, traditional files);
- ➩ The language does not require the awareness of the IT tools used for the calculations (e.g. routines in a programming language, statistical packages like R, SAS, Mathlab, relational databases (SQL), dimensional databases (MDX), XML tools,…);
  - The syntax of the VTL is independent of existing IT calculation tools;
  - On the IT level, this may require a translation from the VTL to the language of the IT tool to be used for the calculation;
  - By implementing the proper translations at the IT level, institutions can use different IT tools to execute the same algorithms; moreover, it is possible for the same institution to use different IT tools within an integrated solution (e.g. to exploit different abilities of different tools);
  - VTL instructions do not change if the IT solution changes (for example following the adoption of another IT tool), so avoiding impacts on users as much as possible;

Extensibility, customizability

- ➩ It is possible to build and extend the language gradually, enriching the available operators according to the evolution of the business needs, so progressively making the language more powerful;
- ➩ In addition, it is possible to call external routines of other languages/tools, provided that they are compatible with the IM; this requisite is aimed to fulfil specific calculation needs without modifying the operators of the language, so exploiting the power of the other languages/tools if necessary for specific purposes

- The external routines should be compatible with, and relate back to, the conceptual IM of the calculations as for its inputs and outputs, so that the integrity of the definitions is ensured
- The external routines are not part of the language, so their use might be subject to some limitations (e.g. it might be impossible to parse them as if they were operators of the language)
- The use of external routines has some drawbacks, because it may obviously compromise the IT implementation independence, the abstraction and the user orientation; therefore external routines should be used only for specific needs and in limited cases, whereas widespread and generic needs should be fulfilled through the operators of the language;

⇨ Nothing can prevent the Organizations adopting the VTL from extending it by defining customized parts, on their own total responsibility and charge, in order to improve the standard language for their specific purposes (e.g. for supporting possible algorithms not permitted by the standard part); also the customized parts must be compliant with the VTL IM and the VTL core assumptions (adopting Organizations are totally in charge of any possible maintenance activity deriving from VTL modifications); such extensions however are not recommended because they can compromise the exchange of validation rules and the use of common tools.

Language effectiveness

⇨ The language is oriented to give full support to the various typologies of data of a statistical environment (for example dimensional data, survey data, registers data, micro and macro, quantitative and qualitative, …) described as much as possible in a coherent way, by means of common IM artefacts for their common aspects, and relying on mathematical notions, as mentioned above. The various types of statistical data are considered as mathematical functions, having independent variables (Identifiers) and dependent variables (Measures, Attributes[3]), whose extensions can be thought as logical tables (DataSets) made of rows (Data Points) and columns (Identifiers, Measures, Attributes).

⇨ The language supports operations on the Data Sets (i.e. mathematical functions) in order to calculate new Data Sets from the existing ones, on the structure components of the Data Sets (Identifiers, Measures, Attributes), on the Data Points.

⇨ The algorithms are specified by means of mathematical expressions which compose the operands (Data Sets, Components …) by means of operators (e.g. +,-,*,/,>,<) to obtain a certain result (Data Sets, Components …);

⇨ The validation is considered as a kind of calculation having as an operand the Data Set to be validated and producing a Data Set containing the outcome of the validation (typically having values “true” and “false” in the measure, respectively for successful and unsuccessful validation); being a Data Set, the result of the validation can be further processed (it can be input of further calculations);

[3] The Measures bear information about the real world and the Attributes about the Data Set or some part of it.

- ➪ Calculations on multiple measures are supported, as well as calculations on the attributes of the Data Sets and calculations involving missing values;
- ➪ The operations are intended to be consistent with the historical changes of the artefacts (e.g. of the code lists, of the hierarchies …), so allowing a proper behaviour for each reference period; the support to this aspect is left to the standards adopting the VTL (e.g. SDMX, DDI …) because different standards may represent historical changes in different ways;
- ➪ The language is ready to allow different algorithms for different reference times (feature to be implemented at a later stage);
- ➪ the VTL operators are generally "modular", meaning that it is possible to compose multiple operators in a single expression; in other words, an operator can have an expression as operand, so obtaining a new expression, and this can be made recursively;
- ➪ The final and the intermediate results of a calculation can be permanently stored (or not) according to the needs;
- ➪ Multiple results may be calculated by means of multiple expressions.

VTL Information Model

Generic Model for Data and their structures

This Section provides a formal model for the structure of data as operated on by the Validation and Transformation Language (VTL).

The purpose is to provide a formal description of data at business level against which other information models (IMs) can be mapped, to facilitate the implementation of VTL with other standards like SDMX, DDI and possibly others. This is the same purpose as the Generic Statistical Information Model (GSIM) and, consequently, this formal model uses the GSIM artefacts as much as possible (GSIM 1.1 version)[4]. Besides, GSIM already provides a first mapping with SDMX and DDI that can be used for the technical implementation[5]. Note that the description of the GSIM 1.1 classes and relevant definitions can be consulted in the "Clickable GSIM" of the UNECE site[6].

Some slight differences between this model and GSIM are due to the fact that in the VTL IM both unit and dimensional data are considered as mathematical functions having independent and dependent variables and are treated in the same way.

For each Unit (e.g. a person) or Group of Units of a Population (e.g. groups of persons of a certain age and civil status), identified by means of the values of the independent variables (e.g. either the "person id" or the age and the civil status), a mathematical function provides for the values of the dependent variables, which are the properties to be known (e.g. the revenue, the expenses …).

A mathematical function can be seen as a **logical table made of rows and columns**. Each column holds the values of a variable (either independent or dependent); each row holds the association between the values of the independent variables and the values of the dependent variables (in other words, each row is a single "point" of the function).

This way, the manipulation of any kind of data (unit and dimensional) is brought back to the manipulation of very simple and well-known objects, which can be easily understood and managed by users. According to these assumptions, there would be no more need to distinguish between unit and dimensional data; nevertheless such a distinction is maintained here in order to make it easier to map the VTL IM to the GSIM IM and, through GSIM, to the DDI and SDMX models.

Starting from this assumption, each mathematical function (logical table) may be defined as a GSIM Data Set and its structure as a GSIM Data Structure, having Identifier, Measure and

---

[4] See also the section "Relations with the GSIM Information model"

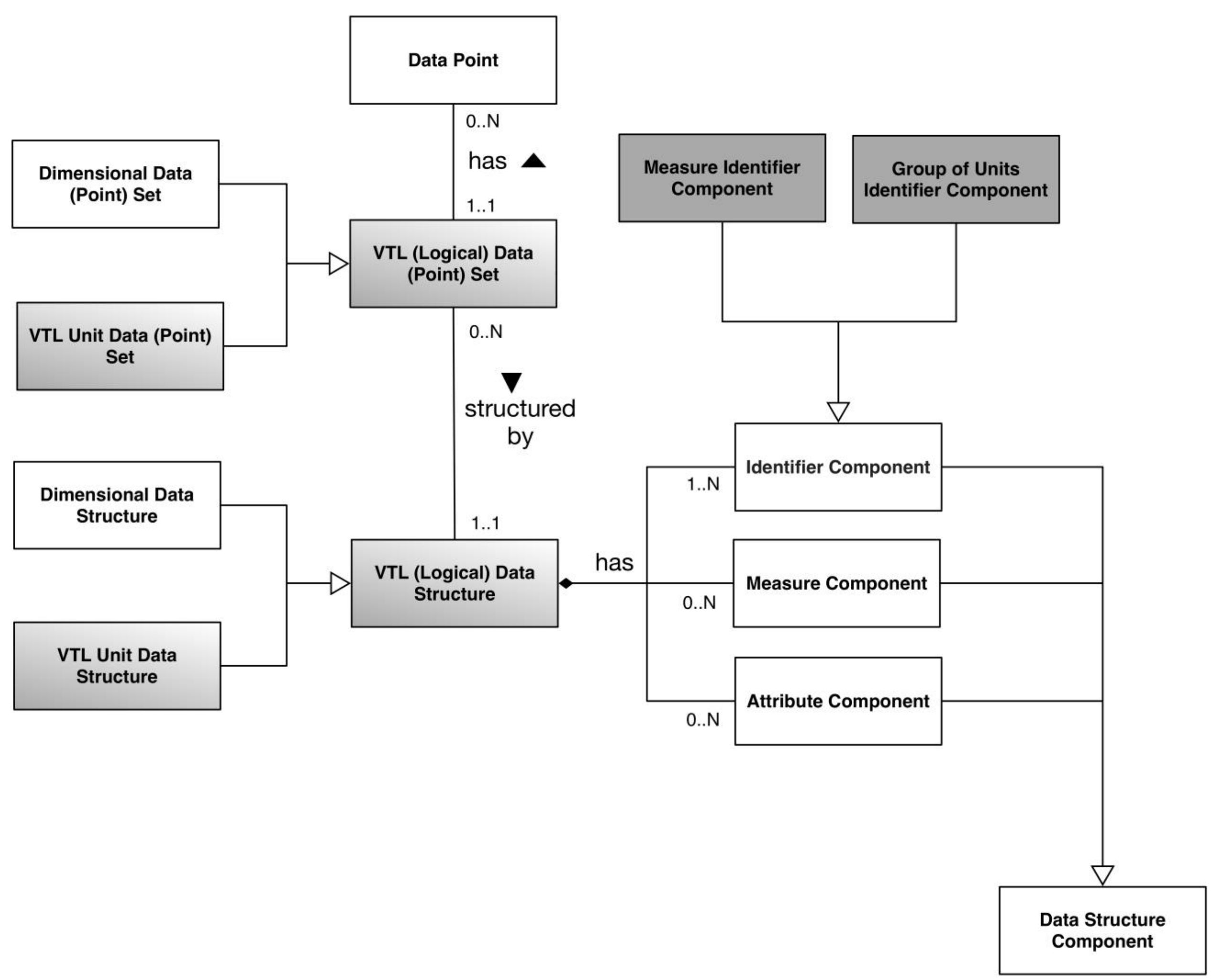
[5] For the GSIM – DDI and GSIM – SDMX mappings, see also the relationships between GSIM and other standards at the UNECE site http://www1.unece.org/stat/platform/display/gsim/GSIM+and+standards. About the mapping with SDMX, however, note that here it is assumed that the SDMX artefacts Data Set and Data Structure Definition may represent both dimensional and unit data (not only dimensional data) and may be mapped respectively to the VTL artefacts Data Set and Data Structure.

[6] Hyperlink "http://www1.unece.org/stat/platform/display/GSIMclick/Clickable+GSIM"

Attribute Components. The Identifier components are the independent variables of the function, the Measures and Attribute Components are the dependent variables. Obviously the GSIM artefacts “Data Set” and “Data Set Structure” have to be strictly interpreted as **logical artefacts** on a mathematical level, not necessarily corresponding to physical data sets and physical data structures.

As earlier pointed out, in respect to GSIM this assumption leads to a representation that is identical for the dimensional data and very similar for the unit data, as described below. The same names as in GSIM are used for the Artefacts, the “VTL” prefix is applied to the Artefact that are very similar to the GSIM ones but not exactly corresponding.

ER diagram - Data model



White box: same artefact as in GSIM 1.1
Light grey box: similar to GSIM 1.1
Dark grey box: additional detail (in respect to GSIM 1.1)

Explanation of the Diagram

VTL (Logical) Data (Point) Set: a mathematical function (logical table) that describes some properties of some groups of units of a population. In general, the groups of units may be composed of one or more units. For unit data, each group is composed of a single unit. For dimensional data, each group may be composed of any number of units. A VTL Data Set is considered as a logical set of observations (Data Points) having the same structure and the same general meaning, independently of the possible physical representation or storage. This artefact is similar to the "Data Set" in GSIM. In particular, the GSIM Data Set may be a GSIM Dimensional Data Set or a GSIM Unit Data, while the VTL Data Set may be:

Dimensional Data (Point) Set: a kind of (Logical) Data Set describing groups of units of a population that may be composed of many units. This artefact is the same as the GSIM Dimensional Data Set.

VTL Unit Data (Point) Set: a kind of (Logical) Data Set describing single units of a population. This is similar to GSIM because the VTL Unit Data Set is the same as the Unit Data Record in GSIM, which has its own structure and can be thought of as a mathematical function. The difference is that the VTL Unit Data Set takes the place of the GSIM Unit Data Set, which is omitted because it cannot be considered as a mathematical function: in fact it can have many GSIM Unit Data Records with different structures.

Data Point: a single value of the function, i.e. a single association between the values of the independent variables and the values of the dependent variables. A Data Point corresponds to a row of the table that describes the function. This artefact is the same as the GSIM Data Point.

VTL (Logical) Data Structure: the structure of a mathematical function, having independent and dependent variables. The independent variables are called "Identifier components", the dependent variables are called either "Measure Components" or "Attribute Components". The distinction between Measure and Attribute components is based on their meaning: the Measure Components give information about the real world, while the Attribute components give information about the function itself. This artefact is similar to the Data Structure in GSIM. In particular, the GSIM Data Structure may be a Dimensional Data Structure or a Unit Data Structure, while the VTL Data Structure may be:

Dimensional Data Structure: the structure of (0..n) Dimensional Data Sets. This artefact is the same as in GSIM.

VTL Unit Data Structure: the structure of (0..n) Unit Data Sets. This is similar to GSIM because the VTL Unit Data Structure is the same as the Logical Record in GSIM, which corresponds to a single structure. The difference is that the VTL Unit Data Structure takes the place of the GSIM Unit Data Structure, which is omitted because it cannot be considered as the structure of a mathematical function: in fact it can have many Logical Records with different structures.

Data Structure Component: any component of the data structure, which can be either an Identifier, or a Measure, or an Attribute Component. This artefact is the same as in GSIM.

Identifier Component (or simply Identifier): a component of the data structure that is an independent variable of the function. This artefact is the same as in GSIM. On the other hand, the following distinction is a detail that does not exist in GSIM, needed to distinguish proper Identifier Components and possible Identifiers Components used in some cases to identify the measures:

Group of Units Identifier Component: a "proper" Identifier Component that contributes to identify the groups of units (composed of either single or many units) that the function describes.

Measure Identifier Component: an Identifier Component that contributes to identify the measures of the function when more measures are conveyed through the same Measure Component. This artefact corresponds to the SDMX Measure Dimension.

Measure Component (or simply Measure): a component of the data structure that is a dependent variable of the function and gives information about the real world. This artefact is the same as in GSIM.

Attribute Component (or simply Attribute): a component of the data structure that is a dependent variable of the function and gives information about the function itself. This artefact is the same as in GSIM.

Examples

As a first simple example, let us consider the following table:

Production of the American Countries

Ref.Date	***Country***	***Meas.Name***	***Meas.Value***	***Status***
2013	Canada	Population	50	Final
2013	Canada	GNP	600	Final
2013	USA	Population	250	Temporary
2013	USA	GNP	2400	Final
…	…	…	…	…
2014	Canada	Population	51	Unavailable
2014	Canada	GNP	620	Temporary
…	…	…	…	…

The whole table is equivalent to a proper mathematical function, in fact its rows have the same structure (in term of columns). The Table can be defined as a Data Set, whose name can be "Production of the American Countries". Each row of the table is a Data Point belonging to the Data Set. The Data Structure of this Data Set has five Data Structure Components:

- Reference Date (Identifier Component)
- Country (Identifier Component)
- Measure Name (Measure Identifier Component)
- Measure Value (Measure Component)
- Status (Attribute Component)

As a second example, let us consider the following physical table, in which the symbol "###" denotes cells that are not allowed to contain a value.

Institutional Unit Data

Row Type	*I.U. ID*	*Ref.Date*	*I.U. Name*	*I.U. Sector*	*Assets*	*Liabilities*
I	A	###	AAAAA	Private	###	###
II	A	2013	###	###	1000	800
II	A	2014	###	###	1050	750
I	B	###	BBBBB	Public	###	###
II	B	2013	###	###	1200	900
II	B	2014	###	###	1300	950
I	C	###	CCCCC	Private	###	###
II	C	2013	###	###	750	900
II	C	2014	###	###	800	850
…	…	…	…	…	…	…

This table is not equivalent as a whole to a proper mathematical function because its rows (i.e. the Data Points) have different structures (in term of allowed columns). However it is easy to recognize that there exist two possible structures (corresponding to the Row Types I and II), so that the original table can be split in the following ones:

Row Type I - Institutional Unit register

I.U. ID	*I.U. Name*	*I.U. Sector*
A	AAAAA	Private
B	BBBBB	Public
C	CCCCC	Private
…	…	…

Row Type II - Institutional Unit Assets and Liabilities

I.U. ID	*Ref.Date*	*Assets*	*Liabilities*
A	2013	1000	800
A	2014	1050	750
B	2013	1200	900
B	2014	1300	950
C	2013	750	900
C	2014	800	850
…	…	…	…

Each one of these two tables corresponds to a mathematical function and can be represented like in the first example above.

In correspondence to one physical table (the former) there are two logical tables (the latter), so that the definitions will be the following ones:

Data Set 1: *Record type I - Institutional Units register*

Data Structure 1:

- I.U. ID (Identifier Component)
- I.U. Name (Measure Component)
- I.U. Sector (Measure Component)

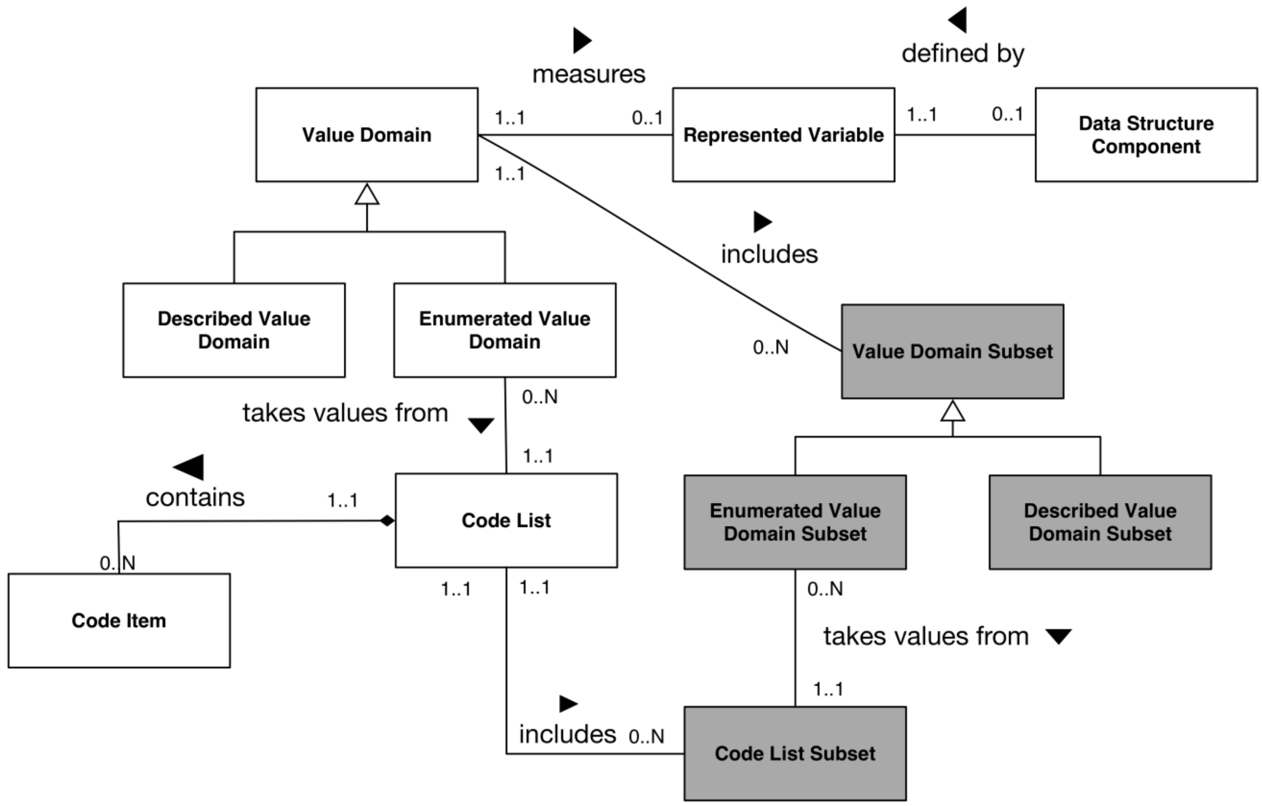
Data Set 2: *Record type II - Institutional Units Assets and Liabilities*

Data Structure 2:

- I.U. ID (Identifier Component)
- Reference Date (Identifier Component)
- Assets (Measure Component)
- Liabilities (Measure Component)

Generic Model for Variables and Value Domains

ER diagram – Variable and Value domain model



Whitebox: same as in GSIM 1.1
Dark grey box: additional detail (in respect to GSIM 1.1)

Explanation of the Diagram

Data Structure Component: see the explanation already given above, in the data model section.

Represented Variable: a characteristic of a statistical population (e.g. the country of birth) represented in a specific way (e.g. through the ISO code). This artefact is the same as in GSIM.

Value Domain: the domain of the allowed values for a variable. This artefact is the same as in GSIM. An important characteristic of the Value Domain is the data type (e.g. String, Numeric, Integer, Boolean, Date), which is the type that any Value of the Value Domain must correspond to.

Described Value Domain: a Value Domain defined by a criterion (e.g. the domain of the positive integers). This artefact is the same as in GSIM.

Enumerated Value Domain: a Value Domain defined by enumeration of the allowed values (e.g. domain of ISO codes of the countries). This artefact is the same as in GSIM.

Code List: a list of allowed codes (values) of an Enumerated Value Domain, with associated categories (e.g. the list of the ISO codes of the countries, each one associated with the name of the country). This artefact is the same as in GSIM.

The following artefacts are aimed to represent possible subsets of the GSIM Value Domains and Code Lists. This is needed for validation purposes, because very often not all the values of the Value Domain are allowed, but only a subset of them (e.g. not all the countries but only the European countries). Although this detail does not exist in GSIM, these artefacts are fully compliant with the GSIM artefacts described above, representing Domains and Code Lists:

> **Value Domain Subset**: a subset of the domain of the allowed values for a variable. This artefact does not exist in GSIM, however it is compliant with the GSIM Value Domain. A Value Domain Subset has the same data type as its Value Domain.
>
> > **Described Value Domain Subset**: a described (defined by a criterion) subset of a Value Domain (e.g. the countries having more than 100 million inhabitants, the integers between 1 and 100). This artefact does not exist in GSIM, however it is compliant with the GSIM Described Value Domain.
> >
> > **Enumerated Value Domain Subset**: an enumerated subset of a Value Domain (e.g. the enumeration of the European countries). This artefact does not exist in GSIM, however it is compliant with the GSIM Enumerated Value Domain.
>
> **Code List Subset**: the list of the codes of an Enumerated Value Domain Subset (e.g. the list of the ISO codes of the European countries). This artefact does not exist in GSIM, however is consistent with the GSIM Code List. The Code List Subset enumerates only the codes and does not associate the categories (e.g. the names of the countries), because the latter are already maintained in the Code List artefact (which contains all the possible codes with the associated categories).

Generic Model for Transformations

The purpose of this section is to provide a formal model for describing the validation and transformation of the data.

A transformation is assumed to be an algorithm to produce a new model artefact (typically a Data Set) starting from existing ones. It is also assumed that the data validation is a particular case of transformation, therefore the term “transformation” is meant to be more general and to include the validation case as well.

This model is essentially derived from the SDMX IM[7], as DDI and GSIM do not have an explicit transformation model at the moment[8]. In its turn, the SDMX model for Transformations is similar in scope and content to the Expression metamodel that is part of the Common Warehouse Metamodel (CWM)[9] developed by the Object Management Group (OMG).

The model represents the user logical view of the definition of algorithms by means of expressions. In comparison to the SDMX and CWM models, some more technical details are omitted for the sake of simplicity, including the way expressions can be decomposed in a tree of nodes in order to be executed (if needed, this detail can be found in the SDMX and CWM specifications).

The basic brick of this model is the notion of a Transformation.

A Transformation specifies the algorithm to obtain a certain artefact of the VTL information model, which is the result of the Transformation, starting from other existing artefacts, which are its operands.

Normally the artefact produced through a Transformation is a Data Set (as usual considered at a logical level as a mathematical function). Therefore, a Transformation is mainly an algorithm for obtaining a derived Data Set starting from already existing ones.

The general form of a Transformation is the following:

```
variable parameter := expression
```

“:=” is the assignment operator, meaning that the result of the evaluation of *expression* in the right-hand side is assigned to the *variable parameter* in the left-hand side, which is the a-priori unknown output of *expression* (typically a Data Set).

In turn, the *expression* in the right-hand side composes some operands (e.g. some input Data Sets) by means of some operators (e.g. sum, product …) to produce the desired results (e.g. the validation outcome, the calculated data).

For example: $D_r := D_1 + D_2$ (D_r, D_1, D_2 are assumed to be Data Sets)

---

[7] The SDMX specification can be found at http://sdmx.org/wp-content/uploads/2011/08/SDMX_2-1-1_SECTION_2_InformationModel_201108.pdf (see package 13 - “Transformations and Expressions”).

[8] The Transformation model described here is not a model of the processes, like the ones that both SDMX and GSIM have. The mapping between the VTL Transformation and the Process models is not covered by the present document, and will be addressed in a separate work task with contributions from several standards experts.

[9] This specification can be found at http://www.omg.org/cwm.

In this example the measure values of the Data Set D_r is calculated as the sum of the measure values of the Data Sets D_1 and D_2.

A validation is intended to be a kind of Transformation. For example, the simple validation that $D_1 = D_2$ can be made through an "If" operator, with an expression of the type:

D_r := If ($D_1 = D_2$, then "true", else "false")

In this case, the Data Set D_r would have a Boolean measure containing the value "true" if the validation is successful and "false" if it is unsuccessful.

These are only fictitious examples for explanation purposes. The general rules for the composition of Data Sets (e.g. rules for matching their Data Points, for composing their measures …) are described in the sections below, while the actual Operators of the VTL are described in the Part 2.

The *expression* in the right-hand side of a Transformation must be written according to a formal language, which specifies the list of allowed operators (e.g. sum, product …), their syntax and semantics, and the rules for composing the expression (e.g. the default order of execution of the operators, the use of parenthesis to enforce a certain order …). The Operators of the language have Parameters[10], which are the a-priori unknown inputs and output of the operation, characterized by a given role (e.g. dividend, divisor or quotient in a division).

Note that this generic model does not specify the language to be used. As a matter of fact, not only the VTL but also other languages might be compliant with this specification, provided that they manipulate and produce artefacts of the information model described above. However the VTL has been agreed as the standard language to define and exchange validation and transformation rules among different organizations.

Also note that this generic model does not actually specify the operators to be used in the language. Therefore, the VTL may evolve and may be enriched and extended.

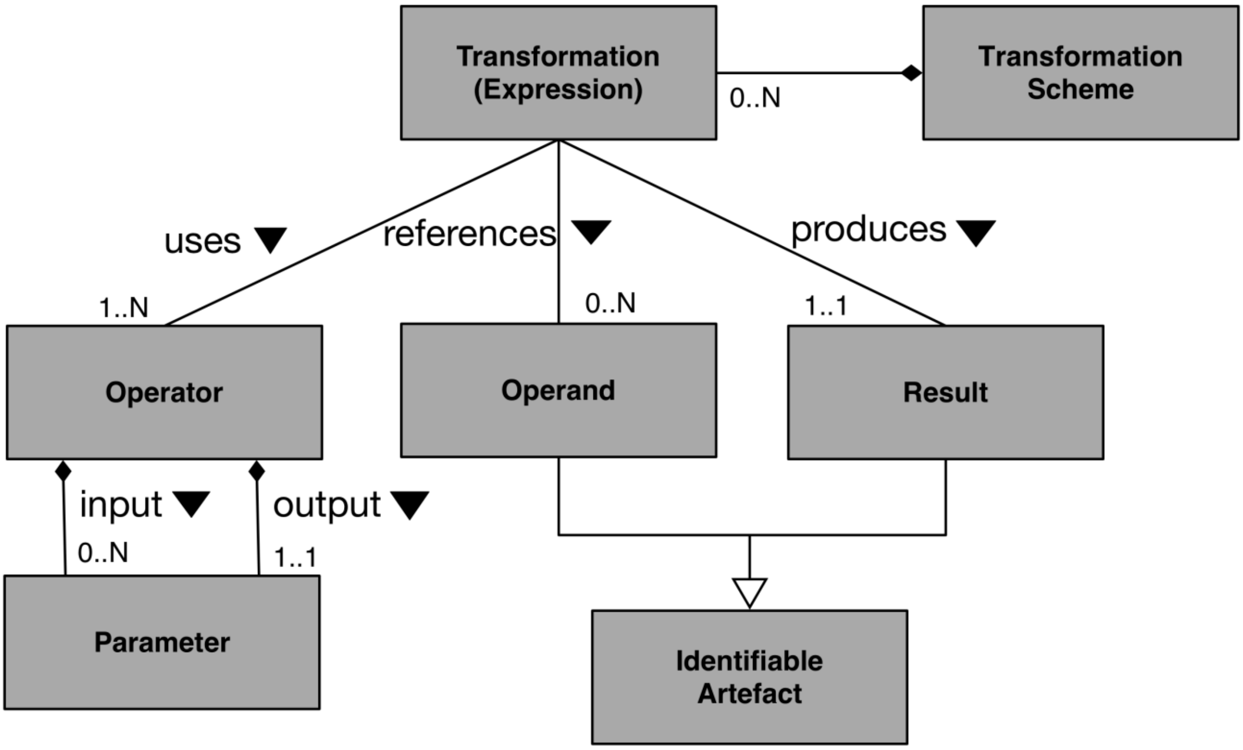
In the practical use of the language, Transformations can be composed one with another to obtain the desired outcomes. In particular, the result of a Transformation can be an operand of other Transformations, in order to define a sequence of calculations as complex as needed.

Moreover, the Transformations can be grouped into Transformations Schemes, which are sets of transformations meaningful to the users. For example a Transformation Scheme can be the set of transformations needed to obtain some specific meaningful results, like the validations of one or more Data Sets.

A set of Transformations takes the structure of a graph, whose nodes are the model artefacts (usually Data Sets) and whose arcs are the links between the operands and the results of the single Transformations. This graph is directed because the links are directed from the operands to the results and is acyclic because it should not contain cycles (like in the spreadsheets), otherwise the result of the Transformations might become unpredictable.

[10] The term is used with the same meaning of "argument", like usual in computer science.

ER Diagram - Transformations



White box: same as in GSIM 1.1
Dark grey box: additional detail (in respect to GSIM 1.1)

(All these artefacts match the SDMX artefact having the same name; however the identifiable artefacts are intended to be the ones of the VTL model)

Explanation of the diagram

Transformation: the basic element of the calculations, which consists in a statement which assigns the outcome of the evaluation of an Expression to an Identifiable Artefact of the Information model; the Transformation artefact is the same as in SDMX;

Expression: a finite combination of symbols that is well-formed according to the syntactical rules of the language; the goal of an Expression is to compose some Operands in a certain order by means of the Operators of the language in order to obtain the desired result; therefore the symbols of the Expression designate Operators, Operands and the order of application of the Operators (e.g. the parenthesis); an expression is defined as a string and is a property of a Transformation, as in SDMX;

Transformation Scheme: a set of Transformations aimed to obtain some meaningful results for the user (like the validation of one or more Data Sets); the Transformation Scheme may be also considered as a VTL program; this artefact is the same as in SDMX;

Operator: the specification of an operation to be performed on some Operands (e.g. +, -, *, /); this artefact is the same as in SDMX;

Parameter: a-priori unknown input or output of an Operator, having a definite role in the operation (e.g. dividend, divisor or quotient for the division) and corresponding to a certain type of artefact (e.g. a "Data Set", a "Data Structure Component" …), the Parameter artefact is the same as in SDMX;

Operand: a specific Identifiable Artefact referenced in the expression as an input (e.g. a specific input Data Set); the distinction between Operand and Result is not explicit in SDMX;

Result: a specific Identifiable Artefact to which the result of the expression is assigned (e.g. the calculated Data Set); the distinction between Operand and Result is not explicit in SDMX;

Identifiable Artefact: an Identifiable Artefact of the VTL information model (e.g. a Data Set, a Data Structure Component); this artefact is the same as in SDMX;

Note that with regards to the SDMX Transformation and Expression Model, some artefacts are intentionally not shown here, essentially to avoid more technical details (i.e. the decomposition of the operations in the Expression, described in SDMX by means of the ExpressionNode and its sub-types ReferenceNode, ConstantNode, OperatorNode). For this reason, in the diagram above, the Transformation references Operators and Artefacts (through its Expression). On the technical implementation perspective, however, the model would be the same as the SDMX one (except some details that are specific to the SDMX context).

Example

Imagine that D_1, D_2 and D_3 are Data Sets containing information on some goods, specifically: D_1 the stocks of the previous date, D_2 the flows in the last period, D_3 the current stocks. Assume that it is desired to check the consistency of the Data Sets using the following statement:

D_r := If $((D_1 + D_2) = D_3$, then "true", else "false")

In this case:

The Transformation may be called "Consistency check between stocks and flows" and is formally defined through the statement above.

- D_r — is the Result
- D_1, D_2 *and* D_3 — are the Operands
- If $((D_1 + D_2) = D_3$, then "true", else "false") — is the Expression
- ":=", "If", "+" , "=" — are the Operators

Each operator has some predefined parameters, for example in this case:

- input parameters of "+": two numeric Data Sets (to be summed)
- output parameters of "+": a numeric Data Sets (resulting from the sum)
- input parameters of "=": two Data Sets (to be compared)
- output parameter of "=": a Data Set (resulting from the comparison)
- input parameters of "If": an Expression defining a condition, i.e. $(D_1+D_2)=D_3$
- output parameter of "If": a Data Set (as resulting from the "then", "else" clauses)

Persistency and Identification of the artefacts of the model

The artefacts of the model can be either persistent or non-persistent. An artefact is persistent if it is permanently stored, and vice-versa.

A persistent artefact exists externally independently of a VTL program, while a non-persistent artefact exists only within a VTL program.

The VTL grammar provides for the identification of the non-persistent artefacts (see the section about the conventions for the grammar of the language) and leaves the accurate definition of the identification mechanism of the persistent artefacts to the standards adopting the VTL (e.g. SDMX, DDI …)[11].

However, the VTL aims at promoting international sharing of rules, which should have a clear identification. Therefore, VTL just gives some minimum requirements about the structure of this universal identifier, assuming that the standards adopting the VTL will ensure that the identifier of a persistent artefact is unique.

In practice, the VTL considers that many definers need to operate independently and simultaneously (e.g. many organizations, units,…), so that they should be made independent as much as possible in assigning names to the artefacts, making sure that nevertheless the resulting names are unique.

Therefore, VTL foresees:

- the **Name** of the artefact (a generic string), which is unique in the environment of the definer;
- an optional **Namespace** (generic string beginning with an alphabetic character) which is a supplementary qualifier that identifies the environment in which the artefact Name is assumed to be unique, to avoid name conflicts.

The Name of the artefact may be composite. For example, in case of versioned artefacts, the Name is assumed to contain the version as well. It is the responsibility of the definer to ensure that the artefact Names are unique in the environment.

The Namespace may be composite as well. For example, a composite structure may be useful to make reference to environments and sub-environments. Notice that VTL does not provide for a general mechanism to ensure that a Namespace is universally unique, which is left to the standards implementing the VTL.

When the context is clear, as typically happens in validation, the Namespace can be omitted. In other words, the Name of the artefact is always mandatory, while the Namespace is required only for the operands that belong to a different Namespace than the Transformation.

As intuitive, the Namespace may begin with the name of the institution ("maintenance agency" in SDMX terms). Assuming the dot (".") as separator character between environments and sub-environments, examples of possible Namespaces are:

- ESCB.analyis&insight
- EuropeanStatisticalSystem.validation
- OECD.Stat

[11] Different standards may have different identification mechanisms.

- Unesco
- Bancaditalia.dissemination.public

The artefact identifier as a whole is also a string, composed of the concatenation of the Namespace – if needed – and the artefact Name, where the slash ("/") symbol is a typical and recommended choice (e.g. “NAMESPACE/NAME” for explicit Namespace definition or simply “NAME” for referencing the default Namespace).

VTL core assumptions

The Validation and Transformation Language is based on two parts: the **core assumptions** and the **standard library of Operators**. The former specifies the general behaviour of the language, and is by default stable. The latter contains the standard set of Operators of the language, and can be gradually enriched following the evolution of the user needs. Possible new operators must obviously comply with the core assumptions.

The core assumptions include:

- The types of Operands and Results
- The operations on the Data Sets
- Storage and retrieval of the Data Sets
- The conventions for the grammar of the language

The core assumptions are explained in the following sections. The standard library of operators is described in the Part 2.

The Types of Operands and Results

The Data types of the VTL

The VTL assumes that operands and results belong to a data type, which influences the operations that can be applied on the data.

The instances of the various data types (i.e. the real objects of those types) are called *literals*.

The ***basic data types*** of the language are five: *String*, *Numeric*, *Integer*, *Boolean* and *Date.* They are described in the following table.

Basic data types	
`String`	A sequence of one or more characters enclosed in double quotes (“). Examples of allowed literals for this data type are: “hello”, “test”, “x”, “this is a string”. Note that in the VTL syntax the double quotes are intended to be the standard ones ("), i.e. the same character to open and close the string, even if in this document and in the Part 2 the styled double quotes may be shown.
`Numeric`	Fixed and floating point numbers, up to 38 digits of precision. At least the following numbers should be representable as numeric in implementations: - Positive numbers in the range 1×10^{-130} to $9.99...9 \times 10^{125}$ with up to 38 significant digits. - Negative numbers from -1×10^{-130} to $9.99...99 \times 10^{125}$ with up to 38 significant digits. - Zero (0). - Positive (+Inf) and negative infinity (-Inf). The point (.) is used as the decimal separator and must be present in the literal. Examples of allowed literals for this type are: 1.0, 234.56, 456.45; also the scientific notation is allowed: 12.23E+12, 35.2E-150, -2E10+3, 0.0.

`Integer`	The basic signed integer type. At least 16 bit in size, although the actual size may vary by implementation. Examples of allowed literals for this type are: 2, 5, 7, 24, -14, 0.
`Boolean`	The Boolean data type. The allowed literals are *true* and *false*.
`Date`	A point-in-time value. The type stores the year, the month, the day, the hours the minutes and the seconds (after midnight). Date are in 24-hours format: YYYY-MM-DD HH24:MI:SS While the YYYY-MM-DD is mandatory, HH24:MI:SS is optional and, if omitted, 00:00:00 is implied. Examples of allowed literal values are: 2012-09-30, 2013-10-02, 2014-01-01 12:23:35. The format for Date literals is customizable, in the sense that specific supplementary formats may be used in implementations in addition to this one, if properly configured in the system. Alternate literals may also include the ones adopted by commercial systems for compatibility reasons, for example: date'2012-09-30'.

With reference to the VTL information model, the data type is a characteristic of the Value Domain. In turn, the data type of the Value Domain is inherited by its Values and its Subsets.

A Represented Variable has the same data type of its Value Domain.

A Structure Component has the same data type of the corresponding Represented Variable (i.e. the data type of its Value Domain).

Also the Data Set has a data type, which is a "composite" one and corresponds to the set of the data types of its Structure Components.

A Transformation (Expression) has the data type of its result.

In conclusion, a data type can be assigned to any artefact of the VTL model (either a basic or a composite data type).

The Parameters of the VTL Operators

As already mentioned, a Parameter is a generic input or output of an Operator and has a definite role in the operation (e.g. dividend, divisor or quotient for the division).

A Parameter corresponds either to a certain type of artefacts of the information model or to some kind of constant value (for the sake of simplicity, the constants have not been described in the IM).

The parameters corresponding to a type of artefacts of the IM are called variable parameters, because their values are not known beforehand (i.e. when the Expression is written and compiled) and can be considered as the "language variables". The types of variable parameter are the Data Set type, the Structure Component type (hereinafter simply Component), the Value Domain Subset type and, possibly, other IM artefacts.

The parameters corresponding to constant values are called constant parameters, because their values are known beforehand (they are written directly in the expressions).

The instances of the various types of Parameters (i.e. the real objects of those types, both variable and constants) are named *literals* (like the instances of the simple data types above).

The following table contains the main types of variable parameters.

Types of variable Parameters	
`Dataset<T>`	A Data Set, having the composite data type T, which corresponds to the set of the data types of its Structure Components. A Data Set may be persistent or not. A persistent Data Set is permanently stored, i.e. maintained in a storage media and therefore exists also independently of a VTL program. A temporary Data Set is not stored and exists only within a VTL program. These sub-types of Datasets are specified by writing: • `PersistentDataset<T>` • `TemporaryDataset<T>`
`Component<T>`	A Structure Component having the data type T. A Structure component has the role of Identifier, Measure or Attribute Component, this role can be specified by writing: • `IdentifierComponent<T>` • `MeasureComponent<T>` • `AttributeComponent<T>` Structure Components can be classified according to their Data Type as: • `String: Component<String>` • `Numeric: Component<Numeric>` • `Integer: Component<Integer>` • `Boolean: Component<Boolean>` • `Date: Component<Date>` Allowed literals are the names of the Structure Components of the Data Sets, as defined in the IM. The membership (#) operator allows referencing specific Components within a Data Set. The syntax is: *dataset_name#component_name* (for a better description see the corresponding section in the Part 2). For the *dataset name* an alias can be used.
`ValueDomainSubset<T>`	A Value Domain Subset of data type T. Value Domain Subsets can be classified according to their Data Type as: • `String: ValueDomainSubset<String>` • `Numeric: ValueDomainSubset<Numeric>` • `Integer: ValueDomainSubset<Integer>` • `Boolean: ValueDomainSubset<Boolean>` • `Date: ValueDomainSubset<Date>)`

In addition to the IM artefacts, the Operators can also use constant values of the following types (they have not been described in the IM for the sake of simplicity):

- Simple Constants (meaning scalar constants belonging to one of the basic data types)
- Sets of Constants (meaning unordered sets of constants having a common data type)
- Lists of Constants (meaning ordered sets of constants having a common data type)

The following table contains the main types of constant parameters.

Types of constant Parameters	
`Constant<T>`	A constant value of data type <T>. Constants can be classified according to their Data Type as: • `String:` `Constant<String>` • `Numeric:` `Constant<Numeric>` • `Integer:` `Constant<Integer>` • `Boolean:` `Constant<Boolean>` • `Date:` `Constant<Date>)`
`ConstantSet<T>`	An unordered collection, without duplicates, of Constants of the same type T. The round brackets "( )" denote that the order is not significant. Examples of allowed literals: ("a","b","c","d"), (1,2,3,4), (1.2, 3.4, 0.0).
`ConstantList<T>`	An ordered collection of Constants of the same type T, enclosed in square brackets, which denotes that the order is significant. Examples of allowed literals: ["a","b","c","d"], [1,2,3,4], [1.2, 3.4, 0.0].

Type management and checking

The language does not have explicit operators for converting the type (typecasting).

It is envisaged that there will be "implicit upcasting" between the Integer and the Numeric data types and between the corresponding types of Parameters. This means that wherever in the language it is possible to use a Constant<Numeric>, a Constant<Integer> is allowed as well. Similarly, wherever it is possible to use a Component<Numeric>, a Component<Integer> is allowed as well. Obviously, the opposite is not allowed. In these cases, in the description of the single Operators in the Part 2, the Numeric type is indicated, provided that there are no particular constraints on using Integers.

The VTL is strongly typed, in the sense that any Parameter belongs to one of the possible types.

The various Operators have specific constraints in terms of number and types of Parameters (see the corresponding sections in the Part 2).

Also a VTL Expression is assumed to correspond to a Parameter type, which is the type of its output Parameter. The type of an Expression can be calculated at compile time.

An Expression can be input of an Operator, provided that the Parameter type of the (result of the) Expression is compliant with the Operator constraints.

The Operators constraints in terms of number and types of Parameters are statically checked (at compile time) so that type errors are not possible at runtime. Moreover, only type-safe upcast conversion for Integers into Numerics is performed.

Type errors result in **compile time exceptions** preventing the Transformations from being used (exchanged, executed …).

The operations on the Data Sets

General rules

As already mentioned, normally the model artefact produced through a Transformation is a Data Set (considered at a logical level as a mathematical function). Therefore a *Transformation* is mainly an algorithm for obtaining a derived Data Set starting from already existing ones. As a matter of fact, the Data Set at the moment is the only type of Parameter that is possible to store permanently through a command of the language (see the Put section in the Part 2).

Let us call Data Set Operator a generic VTL Operator which produces a Data Set. If we assume that F is a Data Set Operator, D_r is its result Data Set and $D_{i\ (i=1,\dots n)}$ are its input Data Sets, the general form of a Transformation based on F can be written as follows:

$$D_r := F\ (D_1, D_2, \dots, Dn)$$

Operator F composes the Data Points of $D_{i\ (i=1,\dots n)}$ to obtain the Data Points of D_r.

For making this operation, F follows a number of default behaviours described here.

In general the Data Sets $D_{i\ (i=1,\dots n)}$ and consequently their Data Points may have any number of Identifier, Measure and Attribute Components, nevertheless the VTL Data Set Operators may require specific constraints on the Data Structure Components of their input Data Sets[12].

The Data Structure Components of the result Data Set D_r will be determined as a function of the Data Structure Components of the input Data Sets and the semantics of the Operator F.

There can exist different cases of application of the Data Set Operators, having specific default behaviours and constraints.

In particular, as for the number of operands, a **Data Set Operator** is called “**unary**” if it allows only one Data Set as input operand (e.g. minimum, maximum, absolute value …) and “**n-ary**” if it requires more than one Data Set as input operand (e.g. sum, product, merge …). The **n-ary** Operators require a preliminary matching between the Data Points of the various input Data Sets.

The **Data Sets** may be also usefully categorized with reference to the number of their Measure Components. A Data Set is called “**mono-measure**” if it has just one Measure Component and “**multi-measure**” if it has two or more Measure Components. For the multi-measure Data Sets it may be necessary to specify which measures should be considered in the operation.

Other cases originate from the possible existence of missing data and Attribute Components. If there are missing values in the input Data Sets, the operation may generate meaningless outcomes, so inducing missing values in the result according to certain rules. On the other hand, there can be the need of producing the values for the Attribute Components of the result starting from the values of the Attributes of the operands.

[12] To adhere to the needed constraints, the identification structure of the Data Sets can be manipulated by means of appropriate VTL Operators, also described in this document.

The Identifier Components and the Data Points default matching

By default, the unary Data Set Operators leave the Identifier Components unchanged, so that the result has the same identifier components as the operand. The operation applies only on the Measures and no matching between Data Points is needed.

The "n-ary" VTL Data Set Operators compose more than one input Data Sets. A simple example is: $D_r := D_1 + D_2$

These Operators (i.e. the +) require a preliminary match between the Data Points of the input Data Sets (i.e. D_1 and D_2) in order to compose their measures (e.g. summing them) and obtain the Data Points of the result (i.e. D_r).

For example, let us assume that D_1 and D_2 contain the population and the gross product of the United States and the European Union respectively and that they have the same Structure Components, namely the Reference Date and the Measure Name as Identifier Components, and the Measure Value as Measure Component:

D_1 = United States Data

Ref.Date	***Meas.Name***	***Meas.Value***
2013	Population	200
2013	Gross Prod.	800
2014	Population	250
2014	Gross Prod.	1000

D_2 = European Union Data

Ref.Date	***Meas.Name***	***Meas.Value***
2013	Population	300
2013	Gross Prod.	900
2014	Population	350
2014	Gross Prod.	1000

The desired result of the sum is the following:

D_r = United States + European Union

Ref.Date	***Meas.Name***	***Meas.Value***
2013	Population	500
2013	Gross Prod.	1700
2014	Population	600
2014	Gross Prod.	2000

In this operation, the Data Points having the same values for the Identifier Components are matched, then their Measure Components are combined according to the semantics of the specific Operator (in the example the values are summed).

The operation above is assumed to happen under a **strict constraint**: the input Data Sets must have the same Identifier Components. The result will also have the same Identifier Components as the operands.

Some Data Set operations (including the sum) may be possible also under a more **relaxed constraint**, that is if the Identifier Components of one Data Set are a superset of those of the other Data Set.

For example, let us assume that D_1 contains the population of the European countries (by reference date and country) and D_2 contains the population of the whole Europe (by reference date):

D_1 = European Countries

Ref.Date	***Country***	***Population***
2012	U.K.	60
2012	Germany	80
2013	U.K.	62
2013	Germany	81

D_2 = Europe

Ref.Date	***Population***
2012	480
2013	500

In order to calculate the percentage of the population of each single country on the total of Europe, the Transformation will be:

$$D_r := D_1 \ / \ D_2 \ * 100$$

The Data Points will be matched according to the Identifier Componen*ts* common to D_1 and D_2 (in this case only the Ref.Date), then the operation will take place.

The result Data Set will have the Identifier Components of both the operands:

D_r = European Countries / Europe * 100

Ref.Date	***Country***	***Population***
2013	U.K.	12.5
2013	Germany	16.7
2014	U.K.	12.4
2014	Germany	16.2

In the Part 2, dedicated to the description of the library of Operators, it is specified whether the Operators require the **strict** or the **relaxed** constraint (see the “Constraints” subsections).

More formally, let F be a generic n-ary VTL Data Set Operator, D_r the result Data Set and D_i $_{(i=1,\dots n)}$ the input Data Sets, so that: $D_r := F(D_1, D_2, \dots, D_n)$

The "strict" constraint requires that the Identifier Components of the D_i $_{(i=1,\dots n)}$ are the same. The result D_r will also have the same Identifier components.

The "relaxed" constraint requires that at least one input Data Set D_k exists such that for each D_i $_{(i=1,\dots n)}$ the Identifier Components of D_i are a (possibly improper) subset of those of D_k. The output Data Set D_r will have the same Identifier Components of D_k.

The n-ary Operator F will produce the Data Points of the result by matching the Data Points of the operands that share the same values for the common Identifier Components and by operating on the values of their Measure Components according to its semantics.

Behaviour for Measure Components

As already mentioned, given $D_r := F(D_1, D_2, \dots, D_n)$, the input Data Sets D_i $_{(i=1,\dots n)}$ may have any number of Measure Components. Therefore to enforce the desired behaviour it is necessary to understand which Measures the Operator is applied to. This Section shows the general VTL assumptions about how Measure Components are handled, while the behaviour of the single operators is described in the Part 2.

The most simple case is the **application of unary Operators to mono-measure Data Sets**, which does not generate ambiguity; in fact the Operator is intended to be applied to the only Measure of the input Data Set. The result Data Set will have the same Measure, whose values are the result of the operation.

For example, let us assume that D_1 contains the salary of the employees (the only Identifier is the Employee ID and the only Measure is the Salary):

D_1 = Salary of Employees

Employee ID	*Salary*
A	1000
B	1200
C	800
D	900

The Transformation $D_r := D_1 * 1.10$ applies to the only Measure (the salary) and calculates a new value increased by 10%, so the result will be:

D_r = Increased Salary of Employees

Employee ID	*Salary*
A	1100
B	1320
C	880
D	990

In case of **unary Operators applied to a multi-measure Data Set**, the Operator *F* is by default intended to be applied separately to all its Measures, unless differently specified. The result Data Set will have the same Measures as the operand.

For example, given the import and export by reference date:

D_1 = Import & Export

Ref.Date	***Import***	***Export***
2011	1000	1200
2012	1300	1100
2013	1200	1300

The Transformation $D_r := D_1 * 0.80$ applies to all the Measures (e.g. to both the Import and the Export) and calculates their 80%:

D_r = 80% of Import & Export

Ref.Date	***Import***	***Export***
2011	800	960
2012	1040	880
2013	960	1040

If there is the need to **apply an Operator only to specific Measures**, the membership (#) operator can be used, which allows referencing specific Components within a Data Set. The syntax is: *dataset_name#component_name* (for a better description see the corresponding section in the Part 2).

For example, in the Transformation D_r := D_1#Import * 0.80

the operation applies only to the Import (and calculates its 80%):

D_r = 80% of the Import, 100% of the Export

Ref.Date	***Import***	***Export***
2011	800	1200
2012	1040	1100
2013	960	1300

Note that in the example above, the Import is kept and left unchanged. In fact by default all the Measures are kept in the result, even the ones that are not operated on. If there is the need to keep only some Measures, the “keep” clause can be used (see the Part 2).

In case of **n-ary Operators**, by default **the operation is applied on the Measures of the input Data Sets having the same names**, unless differently specified. To avoid ambiguities and possible errors, the input Data Sets are constrained to have the same Measures and the result will have the same Measures too.

For example, let us assume that D_1 and D_2 contain the births and the deaths of the United States and the European Union respectively.

D_1 = Births & Deaths of the United States

Ref.Date	***Births***	***Deaths***
2011	1000	1200
2012	1300	1100
2013	1200	1300

D_2 = Birth & Deaths of the European Union

Ref.Date	***Births***	***Deaths***
2011	1100	1000
2012	1200	900
2013	1050	1100

The Transformation $D_r := D_1 + D_2$ will produce:

D_r = Births & Deaths of United States + European Union

Ref.Date	***Births***	***Deaths***
2011	2100	2200
2012	2500	2000
2013	2250	2400

The Births of the first Data Set have been summed with the Births of the second to calculate the Births of the result (and the same for the Deaths).

If there is the need to **apply an Operator on Measures having different names**, the "rename" clause can be used to make their names equal (for a complete description of the clause see the corresponding section in the Part 2).

For example, given these two Data Sets:

D_1 (Residents in the United States)

Ref.Date	***Residents***
2011	1000
2012	1300
2013	1200

D_2 (Inhabitants of the European Union)

Ref.Date	***Inhabitants***
2011	1100
2012	1200
2013	1050

A Transformation for calculating the population of United States + European Union is:

D_r := D_1[rename Residents as Population] + D_2[rename Inhabitants as Population]

The result will be:

D_r (Population of United States + European Union)

Ref.Date	***Population***
2011	2100
2012	2500
2013	1250

Note that the number and the names of the Measure Components of the input Data Sets are assumed to match (following their renaming if needed), otherwise the Expression is considered in error.

In case the Measure Components of the input Data Sets match only partially, the Measure structure must be properly adapted through the features for structure manipulation (e.g. the *keep* and the *calc* clauses, see below and in the relevant sections in the Part 2).

If there is the need to **apply an Operator only to specific Measures**, the membership (#) operator can be used as in the case of unary Operators. Even in this case, by default all the Measures are kept in the result, even the ones that are not operated on; if there is the need to keep only some Measures, the "keep" clause can be used (see the Part 2).

Finally, it may be needed to **apply different Operators on different Measures**. This is possible through the *merge* Operator in combination with the *keep* and *calc* clauses (this offers a wide variety of possibilities, see the specific sections in the Part 2).

Roughly speaking, *merge* allows the production of a Data Set having the union of the Components of the input Data Sets (in a similar way to the SQL join), *keep* selects the Components to keep in the result, *calc* defines specific operations for specific Components.

As a first example, let D_1 and D_2 be two multi-measure Data Sets, both having I as the common Identifier Component and M_1 and M_2 as Measures. Suppose that we want to calculate D_r having the Measures M_3 and M_4, where the former is the sum of the M_1 of the input Data Sets and the latter is the difference of the M_2. This can be obtained as:

```
Dr :=
merge(D1, D2, on(D1#I = D2#I), return(D1#I as I,
```

```
D1#M1 as M11, D2#M1 as M12, D1#M2 as M21, D2#M2 as M22) )
[calc M11 + M12 as M3, M21 – M22 as M4][keep I, M3, M4]
```

The *merge* operator joins *D1* and *D2*, applying the general key matching behaviour on the Identifier Component *I* (the resulting rows). The *return* keyword, which is part of the *merge* operator (see the Part 2), specifies which columns to return in the result, which will have *I* as Identifier Component and four Measure Components, obtained from *D1* and *D2* (two from each). The *calc* clause calculates the sum and the difference between the right pairs of measures. Finally, *keep* maintains only the desired Components.

As another example, assume that D_1 and D_2 are two mono-measure Data Sets, both having *I* as Identifier Component and M_1 as Measure Component. Suppose that we want to calculate D_r having two Measures, M_2 obtained as the sum of the M_1 of the input Data Sets and M_3 obtained as their difference. This can be achieved as:

```
Dr :=
merge(D1, D2, on(D1#I = D2#I),
return(D1#I as I, D1#M1 as M11, D2#M1 as M12) )
[calc M11 + M12 as M2, M11 – M12 as M3][keep I, M2, M3]
```

The *merge* operator joins D_1 with D_2, the *return* keyword produces a temporary multi-measure Data Set where M_{11} and M_{12} have been copied from D_1 and D_2 respectively. Those Measure are in turn summed (into M_2) and subtracted (into M_3). The *keep* maintains only the desired Components.

Finally, note that **each Operator may be applied on Measures of certain data types**, corresponding to its semantics. For example *abs* and *round* will require the Measures to be numeric, while *substr* will require them to be a string. Expressions which violate this constraint are obviously considered in error.

For example consider the Transformation: $D_r := abs(D_1)$

As already described, this expression is assumed to apply the *abs* Operator (i.e. absolute value) to all the Measures Components of D_1. If all these Measures are quantitative the expression is considered correct, on the contrary, if at least one Measure is of an incompatible data type, the expression is considered in error. The general description of the VTL data types is given above while the description of the data types on which each operator can be applied is given in the Part 2.

Order of execution

VTL allows the application of many Operators in a single expression. For example:

$$Dr := D_1 + D_2 \; / \; (D_3 - D_4 \, / \, D_5)$$

When the order of execution of the Operators is not explicitly defined (through the use of parenthesis), a default order of execution applies.

In the case above, according to the VTL precedence rules, the order will be:

I. D_4 / D_5 (default precedence order)
II. $D_3 - I$ (explicitly defined order)
III. D_2 / II (default precedence order)
IV. $D_1 + III$ (default precedence order)

The default order of execution depends on the precedence and associativity order of the VTL Operators and is described in detail in the Part 2.

Missing Data

The awareness of missing data is very important for correct VTL operations, because the knowledge of the Data Points of the result depends on the knowledge of the Data Points of the operands. For example, assume $D_r := D_1 + D_2$ and suppose that some Data Points of D_2 are unknown, it follows that the corresponding Data Points of D_r cannot be calculated and are unknown too.

Missing data can take up two basic forms.

In the first form, **the lack of information is explicitly represented**. This is the case of Data Points that show a "missing" value for some Measure or Attribute Components, which denotes the absence of a true value for a Component. The "missing" value is not allowed for the Identifier Components, in order to ensure that the Data Points are always identifiable.

In the second form, **the lack of information remains implicit**. This is the case of Data Points that are not present at all in the Data Set. For example, given a Data Set containing the reports to an international organization relevant to different countries and different dates, and having as Identifier Components the Country and the Reference Date, this Data Set may lack the Data Points relevant to some dates (for example the future dates) or some countries (for example the countries that didn't send their data) or some combination of dates and countries.

The interpretation of the Data Points that are not present in the Data Set may be different in different cases. There are situations in which it is not correct to assume that such Data Points are "unknown". As a matter of fact, there exist significant cases in which the "known" Data Points having a prefixed value (e.g. the "zero" value) are intentionally omitted, so that:

- It is not possible to conclude that the missing Data Points are unknown;
- it may be required to consider the missing Data Points as known and having such a prefixed value.

The most common case of this kind is the "zero" value for quantitative data. According to a common practice, in fact, in high volume sparse data (i.e. when most of the Data Points have the value "zero"), the Data Points equal to "zero" are intentionally omitted, because it would be highly cumbersome or even unbearable to represent them explicitly. In these cases it may be correct to assume that the missing Data Points are "known" and have the value "zero". This situation will be called hereinafter "implicit zero".

On the contrary, if the Data Points assuming the value "zero" are explicitly represented, it is correct to assume that the missing Data Points are "unknown". This situation is called "explicit zero".

For some quantitative Operators, the current version of VTL allows both implicit and explicit zero operations. In the former case, if a calculation finds missing Data Points for an operand, the corresponding result is regularly calculated assuming for them the value "zero". In the latter case, on the contrary, the result is considered "unknown".

For the sake of clarity, the VTL introduces distinct operators for the two cases. For example, the VTL algebraic operators *(+, -, *, /)* operate in implicit zero mode, while there are other corresponding operators (*++, --, **, //*) which perform the same operation in explicit zero mode.

In practice, considering the case $D_r = F(D_1, D_2)$, if a Data Point P_1 of D_1 does not match with any Data Point P_2 of D_2 (i.e. there does not exist a P_2 of D_2 having the same value for the Identifier Components as P_1 of D_1), both the kinds of operators assume a fictitious matching Data Point P_{2F}, whose Measure Components are assigned the value "zero" by the former kind of operators *(+, -, *, /)* and the value "unknown" (NULL) by the latter (*++, --, **, //*).

Coming back to the case of the **explicit representation of the "missing" values**, there can exist more missing values having different meanings. For example, possible meanings are "non-reported data" (the value should have been reported but it is absent), "nil data" (the data is negligible or zero), "not applicable data" (data is missing as expected) and so on. At the moment there is no standardization of the missing values and different organizations may use different sets of missing values (the goal of standardizing the missing values is out of the context of this work). Moreover, the needed missing values may change.

A common practice to deal with missing values is to use just one value for the Measure Components having the generic meaning of "unknown" (the NULL literal) and introducing dedicated Attribute Components to better qualify the meaning as "non-reported", "nil", "not applicable" and so on.

The VTL supports this practice through the NULL literal and the propagation rules of the Attribute Components, which are described below.

The general properties of the NULL are the following ones:

- **Data type:** NULL is type-less; this means that it is an allowed value for a Component of any data type (e.g. Numeric, String, Boolean …)
- **Testing**. A specific Operator (**isnull**) allows to test if a value is NULL returning a Boolean value (TRUE or FALSE).
- **Comparisons**. Whenever a NULL value is involved in a comparison (>, <, >=, <=, in, not in, between) the result of the comparison is NULL.
- **Mathematical operations**. Whenever a NULL value is involved in a mathematical operation (+, -, *, /, …), the result is NULL.
- **String operations**. In operations on Strings, NULL is considered an empty String ("").
- **Boolean operations**. VTL adopts 3VL (three-value logic). Therefore the following deduction rules are applied:

TRUE	*or*	NULL	→	TRUE
FALSE	*or*	NULL	→	NULL
TRUE	*and*	NULL	→	NULL
FALSE	*and*	NULL	→	FALSE

- **Conditional operations**. The NULL is considered equivalent to FALSE; for example in the control structures of the type (*if (p) -then -else*), the action specified in *–then* is executed if the predicate *p* is TRUE, while the action *-else* is executed if the *p* is FALSE or NULL;
- **Filter clauses**. The NULL is considered equivalent to FALSE; for example in the filter clause [*filter p*], the Data Points for which the predicate *p* is TRUE are selected and returned in the output, while the Data Points for which *p* is FALSE or NULL are discarded.
- **Aggregations**. The aggregations (like *sum*, *avg* and so on) return one Data Point in correspondence to a set of Data Points of the input. In these operations the input Data Points having a NULL value are in general not considered. In the average, for example,

they are not considered both in the numerator (the sum) and in the denominator (the count). Specific cases for specific operators are described in the respective sections.

- **Implicit zero**. Arithmetic operators assuming implicit zeros (+,-,*,/) may generate NULL values for the Identifier Components in particular cases (superset-subset relation between the set of the involved Identifier Components). Because NULL values are in general forbidden in the Identifiers, the final outcome of an expression must not contain Identifiers having NULL values. As a momentary exception needed to allow some kinds of calculations, Identifiers having NULL values are accepted in the partial results. To avoid runtime error, possible NULL values of the Identifiers have to be fully eliminated in the final outcome of the expression (through a selection, or other operators), so that the operation of "assignment" (:=) does not encounter them.

If a different behaviour is desired for NULL values, it is possible to **override** them. This can be achieved with the combination of the *calc* and *isnull* operators.

For example, suppose that in a specific case the NULL values of the Measure Component M_1 of the Data Set D_1 have to be considered equivalent to the number 1, the following Transformation can be used to multiply the Data Sets D_1 and D_2, preliminarily converting NULL values of $D_1\#M_1$ into the number 1. For detailed explanations of *calc* and *isnull* refer to the specific sections in the Part 2.

D_r `:=` D_1 `[calc if(ISNULL(M1) then 1 else M1) as M1] *` D_2

The Attribute Components

Given as usual $D_r := F(D_1, D_2, \dots, D_n)$ and considering that the input Data Sets $D_{i\ (i=1,\dots n)}$ may have any number of Attribute Components, there can be the need of calculating the desired Attribute Components of D_r. This Section describes the general VTL assumptions about how Attributes are handled (specific cases are dealt with in description of the single operators in the Part 2).

It should be noted that the Attribute Components of a Data Set are dependent variables of the corresponding mathematical function, just like the Measures. In fact, the difference between Attribute and Measure Components lies only in their meaning: it is intended that the Measures give information about the real world and the Attributes about the Data Set itself (or some part of it, for example about one of its measures).

The VTL has a different default behaviour for Attributes and for Measures.

As specified above, Measures are kept in the result by default, whereas Attributes are assigned a characteristic called "**virality**", which determines if the Attribute is kept in the result by default or not: a "**viral**" Attribute is kept while a "**non-viral**" Attribute is not kept (the default behaviour is applied when no explicit indication about the keeping of the Attribute is provided in the expression).

A second aspect is the "virality" of the Attribute in the result. By default, a viral Attribute is considered viral also in the result.

A third aspect is the operation performed on an Attribute. By default, **the operations which apply to the Measures are not applied to the Attributes**, so that the operations on the Attributes need a dedicated specification. If no operations are explicitly defined on an Attribute, a default calculation algorithm is applied in order to determine the Attribute's values in the result.

As already mentioned, when the default behaviour is not desired, a different behaviour can be specified by means of the proper use of the *keep*, *calc* and *attrcalc* clauses. In particular, through these clauses, it is possible to override the virality (to keep a *non-viral* Attribute or not to keep a *viral* one), to alter the virality of the Attributes in the result (from *viral* to *non-viral* or vice-versa) and to define a specific calculation algorithm for an Attribute (see the detailed description of these clauses in the Part 2).[13]

Hence, the **default Attribute propagation rule** behaves as follows:

- the non-viral Attributes are not kept in the result and their values are not considered;
- the viral Attributes of the operand are kept and are considered viral also in the result; in other words, if an operand has a viral Attribute V, the result will have V as viral Attribute also;
- The Attributes, like the Measures, are combined according to their names, e.g. the Attributes having the same names in multiple Operands are combined, while the Attributes having different names are considered as different Attributes;
- the values of the Attributes which exist and are viral in only one operand are simply copied (obviously, in the case of unary Operators this applies always);
- the Attributes which exist and are viral in multiple operands (i.e. Attributes having the same names) are combined in one Attribute of the result (having the same name also), whose values are calculated according to the default calculation algorithm explained below;

Extending an example already given for unary Operators, let us assume that D_1 contains the salary of the employees of a multinational enterprise (the only Identifier is the Employee ID, the only Measure is the Salary, and there are two other Components defined as viral Attributes, namely the Currency and the Scale of the Salary):

D_1 = Salary of Employees

Employee ID	*Salary*	*Currency*	*Scale*
A	1000	U.S. $	Unit
B	1200	€	Unit
C	800	yen	Thousands
D	900	U.K. Pound	Unit

The Transformation $D_r := D_1 * 1.10$ applies only to the Measure (the salary) and calculates a new value increased by 10%, the viral Attributes are kept and left unchanged, so the result will be:

[13] In particular the *keep* clause allows the specification of whether or not an attribute is kept in the result while the *calc* and the *attrcalc* clauses make it possible to define calculation formulas for specific attributes. The *calc* can be used both for Measures and for Attributes and is a unary Operator, e.g. it may operate on Components of just one Data Set to obtain new Measures / Attributes, while the *attrcalc* is dedicated to the calculation of the Attributes in the N-ary case

D_r = Increased Salary of Employees

Employee ID	*Salary*	*Currency*	*Scale*
A	1100	U.S. $	Unit
B	1320	€	Unit
C	880	yen	Thousands
D	990	U.K. Pound	Unit

The Currency and the Scale of D_r will be considered viral too and therefore would be kept also in case D_r becomes operand of other Transformations.

For n-ary operations, the VTL **default Attribute calculation algorithm** produces the values of the Attributes of the result Data Set from those of its operands and is applied by default if no operations on the Attributes are explicitly defined. This algorithm is independent of the Operator applied on the Measures and works as follows:

- Whenever in the evaluation of a VTL expression, two data points P_i and P_j are combined as for their Measures, the Attributes having the same name, if viral, are combined as well (non-viral Attributes are ignored)
- It is assumed that each possible value of an Attribute is associated to a **default weight** (in the IM, this is a type of property of the Value Domain which contains the possible values of the Attribute);
- the result of the combination is **the value having the highest weight**;
- if multiple values have the same weight, the result of the combination is the first in lexicographical order.

Note that the default weight for each possible value of an Attribute can be overridden, if desired. However this is out of the scope of the language: the specific implementations will provide configuration mechanisms (e.g. a user modifiable text file) to alter such values.

For example, let us assume that D_1 and D_2 contain the births and the deaths of the United States and the Europe respectively, plus a viral Attribute that qualifies if the Value is estimated (having values True or False).

D_1 = Births & Deaths of the United States

Ref.Date	*Births*	*Deaths*	*Estimate*
2011	1000	1200	False
2012	1300	1100	False
2013	1200	1300	True

D_2 = Birth & Deaths of the European Union

Ref.Date	*Births*	*Deaths*	*Estimate*
2011	1100	1000	False
2012	1200	900	True
2013	1050	1100	False

Assuming the weights 1 for “false” and 2 for “true”, the Transformation $D_r := D_1 + D_2$ will produce:

D_r = Births & Deaths of United States + European Union

Ref.Date	***Births***	***Deaths***	***Estimate***
2011	2100	2200	False
2012	2500	2000	True
2013	2250	2400	True

Note also that:

- if the attribute *Estimate* was non-viral in both the input Data Sets, it would not be kept in the result
- if the attribute *Estimate* was viral only in one Data Set, it would be kept in the result with the same values as in the viral Data Set

The VTL default Attribute propagation rule (here called A) ensures the following properties (in respect to the application of a generic VTL operator “§” on the measures):

Commutative law (1)

$A(D_1 \S D_2) = A(D_2 \S D_1)$

The application of *A* produces the same result (in term of Attributes) independently of the ordering of the operands. For example, $A(D_1 + D_2) = A(D_2 + D_1)$. This may seem quite intuitive for “sum”, but it is important to point out that it holds for every operator, also for non-commutative operations like difference, division, logarithm and so on; for example $A(D_1 / D_2) = A(D_2 / D_1)$

Associative law (2)

$A(D_1 \S A(D_2 \S D_3) = A(A(D_1 \S D_2) \S D_3)$

Within one operator, the result of *A* (in term of Attributes) is independent of the sequence of processing.

Reflexive law (3)

$A(\S(D_1)) = A(D_1)$

The application of *A* to an Operator having a single operand gives the same result (in term of Attributes) that its direct application to the operand (in fact the propagation rule keeps the viral attributes unchanged).

Having these properties in place, it is always possible to avoid ambiguities and circular dependencies in the determination of the Attributes’ values of the result. Moreover, it is sufficient without loss of generality to consider only the case of binary operators (i.e. having two Data Sets as operands), as more complex cases can be easily inferred by applying the VTL Attribute propagation rule recursively (following the order of execution of the operations in the VTL expression).

With regard to this last aspect, the VTL assumes that the **order of execution** of the operations in an expression is determined by the precedence and associativity rules of the Operators applied on the Measures, as already explained in the relevant section. The operations on the

Attributes are performed in the same order, independently of the application of the default Attribute propagation rule or user defined operations.

For example, recalling the example already given:

$$D_r := D_1 + D_2 \ / \ (D_3 - D_4 \ / \ D_5)$$

The evaluation of the Attributes will follow the order of composition of the Measures:

I. $A(D_4 \ / \ D_5)$ (default precedence order)
II. $A(D_3 \ \text{-} \ I)$ (explicitly defined order)
III. $A(D_2 \ / \ II)$ (default precedence order)
IV. $A(D_1 + III)$ (default precedence order)

Storage and retrieval of the Data Sets

The Storage

As mentioned, the general form of Transformation can be written as follows:

$$D_r := F \ (D_1, D_2, \ldots, Dn)$$

In practice, the right-hand side is a mathematical expression like the one described above:

$$D_r := D_1 + D_2 \ / \ (D_3 - D_4 \ / \ D_5)$$

As already shown, this expression implies the calculation of many Data Sets in different steps:

I. $(D_4 \ / \ D_5)$
II. $(D_3 \ \text{-} \ I)$
III. $(D_2 \ / \ II)$
IV. $(D_1 + III)$

Calculated Data Sets are assumed to be non-persistent (temporary), as well as D_r , to which is assigned the final result of the expression (step IV).

A temporary result within the expression can be only input of other operators in the same expression.

Parameter D_r , which the result of the whole expression is assigned to, can be directly referenced as operand by other Transformations of the same VTL program (a VTL program is a set of Transformations, that is a Transformation Scheme, aimed to obtain some meaningful results for the users, supposed to be executed in the same run).

The *Put* command is used to specify that a result must be persistent. Any step of the calculation can be made persistent (including all the steps).

The *Put* has two parameters, the first is the (partial) result of the calculation that has to be made persistent (a non-persistent parameter of *Dataset* type), the second is the reference to the persistent Data Set, for example:

$$D_r := Put(D_1 + D_2 \ / \ (D_3 - D_4 \ / \ D_5), \text{“PDS1”})$$

means that the overall result of the expression is stored in the persistent Data Set having name PDS1. The expression:

$$D_r := Put(D_1 + D_2 \ / \ Put((D_3 - D_4 \ / \ D_5), \text{“PDS1”}), \text{“PDS2”})$$

Specifies that $(D_3 - D_4 \ / \ D_5)$ is stored in *PDS1* and the overall result in *PDS2*.

The Retrieval

Considering again the general form of Transformation:

$$D_r := F(D_1, D_2, \ldots, Dn)$$

the "n" Data Sets $D_{i\ (i=1,\ldots\ n)}$ are the operands of the Expression and their values have to be retrieved.

The generic D_i may be retrieved either as the temporary result of another Transformation (of the same VTL program) or from a persistent data source. In the former case D_i is the name of the left-hand parameter (D_r) of the other Transformation. In the latter, D_i is the reference to a persistent Data Set (see the following sections).

A specific Operator (Get) ensures powerful features for accessing persistent data (see the detail in the Part 2). A direct reference to a persistent Data Set is equivalent to the application of the Get command.

The Operators Get and Put are also called "commands" because they allow the interaction with the persistent storage.

The references to persistent Data Sets

In defining the Transformations, persistent Data Sets can be retrieved or stored by means of the Get and Put commands respectively.

As described in the VTL IM, the Data Set is considered as an artefact at a logical level, equivalent to a mathematical function having independent variables (Identifiers) and dependent variables (Measures and Attributes). A Data Set is a set of Data Points, which are the occurrences of the function. Each Data Point is an association between a combination of values of the independent variables and the corresponding values of the dependent variables.

Therefore, the VTL references the conceptual/logical Data Sets and does not reference the physical objects where the Data Points are stored. The link between the Data Set at a logical level and the corresponding physical objects is out of the scope of the VTL and left to the implementations.

Also the versioning of the artefacts of the information model, including the Data Sets, is out of the scope of the VTL and left to the implementations.

The VTL allows reference through commands (Get and Put) to any persistent Data Set defined and identified according the VTL IM. For correct operation, knowledge of the Data Structure of the input Data Sets is essential, in order to check the correctness of the expression and determine the Data Structure of the result. For this reason, the VTL requires that at compilation time the Data Structures of the referenced Data Sets are available.

In addition, to simplify some kind of operations, the VTL makes it possible to reference also Cartesian subsets of the already defined Data Sets (i.e. sub Data Sets specified as Cartesian products of Value Domain Subsets of some Identifier Components).

This is consistent with the IM, because any subset of the Data Points of a Data Set may be considered in its turn a Data Set, and with correct VTL operations, because the Data Structure of a sub Data Set is deducible from the Data Structure of the original Data Set, once that the specification of the subset is given.

Note however that it is not possible to reference directly a non-Cartesian sub Data Set (i.e. a sub Data Set that cannot be obtained as a Cartesian product of Value Domain Subsets). As any other kind of Data Set, however, non-Cartesian subsets can be obtained through an Expression, as partial or final results.

For example, in case of unit data, given the Data Set "Legal Entity" having as Identifiers of the Country, the IssuerOrganization, and the LegalEntityNumber, the VTL allows direct reference to either the whole Data Set or a sub-Data Set obtained specifying some countries, and/or issuers, and/or numbers. By specifying a single value for each identifier it is possible to reference even a single Legal Entity (i.e. a single Data Point).

In case of Dimensional Data Sets, assuming that the Country and the Date are the Identifiers, it is possible to reference the sub Data Sets corresponding to one or some countries, to one or some dates, and to a combination of them. If the dates are periodical, the sub Data Set corresponding to one country is a time-series. The sub Data Set corresponding to a certain date is a cross-section. The sub Data Set corresponding to one country and one date is a single Data Point. Therefore the VTL allows direct reference to dimensional data, time-series, cross-sections, and single observations.

In conclusion, a VTL reference to a persistent (sub)Data Set is composed of two parts:

- The identification of the Data Set (mandatory)
- The specification of a subset of it (optional)

The Identification of a persistent Data Set

The identification of the persistent Data Sets to read from (Get) or to store into (Put) follows the general rules of identification of the persistent artefact (see the corresponding section above).

Therefore, the Data Set identifier is the **Data Set Name**, which is unique in the environment. As different environments can use the same Data Set Names for their artefacts, the Data Set Name can optionally be qualified by a **Namespace**, to avoid name conflicts.

In case the Data Set identifier has a Namespace, a separator character can be chosen (and configured in the system) among the non-alphanumeric ones. A typical, and recommended, choice is the slash ("/") symbol. If the Data Set identifier does not have a Namespace, the same namespace as the respective Transformation is assumed.

Examples of good references to Data Sets are:

"NAMESPACE/DS_NAME" (explicit Namespace definition)

"DS_NAME" (the Namespace of the Transformation is assumed)

The specification of a subset of a persistent Data Set

The VTL allows the retrieval or storage of a subset of a predefined Data Set by filtering the values of its Identifier Components.

Two basic options are allowed in the grammar of this specification:

- A **full notation (query string), specifying both the Identifiers and the values** to be filtered (e.g. Date= 2014, Country=USA, Sector=Public ...); in this case the filtering condition is preceded by the "?" symbol.
- A **short notation (ordered concatenation), specifying only the values** to be filtered (e.g. 2014.USA.Public); in this case the filtering condition is preceded by the "/" symbol; the values have to be specified following a predefined order of the Identifiers.

The **query string** is a postfix syntax specifying the filter in case the order of the identifiers is not defined beforehand or not known.

The filter is specified by concatenating the filtering conditions on the Identifiers, expressed in any order and separated by "&". If a filtering condition is not specified for an Identifier, the latter is not constrained and all the available values are taken. For example:

I. `DS_NAME?DATE=2014&COUNTRY=USA&SECTOR=PUBLIC`

In the example above, **single values** are specified for each filtering condition.

In the same way, it is also possible to specify **multiple values** for some filtering conditions, separating the values by the "**+**" keyword (list). For example, to take the years 2013 and 2014 and the countries USA and Canada:

II. `DS_NAME?DATE=2013+2014&COUNTRY=USA+CANADA&SECTOR=PUBLIC`

Finally, where the Values have an order like the one for the "Date" data type, it is possible to specify ranges of values for some filtering conditions, separating the first and last values of the range by the "**-**" keyword (range). For example, to take all the years from 2008 to 2014:

III. `DS_NAME?DATE=2008-2014&COUNTRY=USA+CANADA&SECTOR=PUBLIC`

The **ordered concatenation** is a simplified syntax to specify the filter in case the order of the identifiers is defined beforehand and known.

The filter is specified by concatenating the filtering conditions in the predefined order of the Identifiers; the filtering conditions do not require the specification of the name of the Identifier, which can be deduced by their predefined order, therefore only the values are specified, separated by "**.**", i.e. a dot. If a value is omitted, the corresponding Identifier is not constrained and all the available values are taken. For example, (assuming that the order on the identifiers is 1-Date, 2-Country, 3-Sector):

I. `DS_NAME/2014.USA.PUBLIC`

   This definition in the query string syntax corresponds to:

   `DS_NAME?DATE=2014&COUNTRY=USA&SECTOR=PUBLIC`

II. `DS_NAME/.USA.PUBLIC`

   This definition filters all the available years for the USA and the public sector, and in the query string syntax corresponds to:

   `DS_NAME?COUNTRY=USA&SECTOR=PUBLIC`

III. `DS_NAME/..PUBLIC`

   This definition filters all the available years and countries for the public sector and in the query string syntax corresponds to:

   `DS_NAME?SECTOR=PUBLIC`

If needed, the list ("+") and/or range ("-") keywords can be used to specify lists or range of values respectively. For example:

IV. `DS_NAME/2008-2014.USA+CANADA.PUBLIC`

   This definition in the query string syntax corresponds to:

   `DS_NAME?DATE=2008-2014&COUNTRY=USA+CANADA&SECTOR=PUBLIC`

Conventions for the grammar of the language

General conventions

A VTL program is a set of Transformations executed in the same run, which is defined as a Transformation Scheme.

Each Transformation consists in a *statement* that is an assignment of the form:

```
variable parameter := expression
```

“:=” is the assignment operator, meaning that the result of the evaluation of the *expression* in the right-hand side is assigned to the *variable parameter* in the left-hand side (which is the output parameter of the assignment).

Examples of assignments are (assuming that ds_i *(i=1…n)* are Data Sets):

- `ds_1 := ds_2`
- `ds_3 := ds_4 + ds_6`

Variable Parameter names

The variable parameters are non-persistent (temporary).

The names of the variable parameters are alphanumeric (starting with an alphabetic character). Also non alphabetic characters (“_”,”-“) are allowed, but not in the first position. Parameter names are case-sensitive.

Examples of allowed names for the parameters are: par1, p_1, VarPar_ABCD, paraMeterXY.

Reserved keywords

Certain words are reserved **keywords** in the language and cannot be used as parameter names, they include:

- all the names of the operators / clauses
- all the symbols used by the language (assignment “:=”, parenthesis “(“,“)”,“[“ ,“]”, ampersand “&”, hash “#” …)
- true
- false
- all
- imbalance
- errorlevel
- condition
- msg_code
- dataset
- script

Expressions

The expression is the right-hand side of an assignment and can be built in a number of alternative options.

In general, an Expression may be the result of the application of an operator to another (sub)-expression. This may be done recursively, as already shown in the examples in the sections above. In other words, an Expression can be an operand of an Operator, resulting in another Expression that in turn may be an operand of an Operator, and so on.

The basic and simplest types of Expressions correspond to the types of Constants and Parameters, for example an Expression can be:

A **Constant**, that is a literal of any data type. Examples of Constant Expressions are:

```
String:       'hello world', 'string'
Numeric:      12.34, 0.0, 23.2E+4
Integer:      2, 0, 45
Boolean:      true, false
Date:         2012-01-31
```

A **Constant Set** of any data type. Examples of Constant Sets Expressions are:

```
String:       ('k', '7', 'l')
Numeric:      (12.34, 0.0, 23.2E+4)
```

A **Constant List** of any data type. Examples of Constant List Expressions are:

```
Numeric:      [12.34, 0.0, 23.2E+4]
Boolean:      [true, false, false]
```

A **Data Set** of any data type. Examples of Data Set Expressions are:

```
Reference to temporary Data Sets:      ds_1, DatasetA, X, Y
Reference to a persistent Data Set:    Namespace/DS_Name
```

A **Component** of any data type. Examples of Component Data Set Expressions are:

```
Component of a temporary Data Sets:    ds_1#date, X#country
Component of a persistent Data Set:    Namespace/DS_Name#sector
In the context of a single Data Set:   date, country, sector
```

A **Value Domain Subset** of any data type. Examples of Value Domain Subset Expressions are:

```
Reference to a persistent V.D.S.:      Namespace/VDS_Name
```

The other types of Expressions correspond to the ways an expression can be built from the basic types. For example an Expression can be:

The **application of an Operator to other Expressions** (as explained above). Some examples are:

```
Functional style:        length(D1),  round(D2, 4)
Non functional style:    D1+D2,    D1 and (D2 or D3),
```

The **application of a Clause to an Expression**. This is the same as above as for the semantic, and it is only different for the syntax, because the clauses are operators that use a postfix style. Some examples are the following ones (the D symbols denote Data Set names and the C symbols the Component names):

```
D1[rename C1 as C2]
D2+D4[keep C1, C2, C3]
D3*(D2+D4)[calc C2*C3 as C5]
```

Comments

VTL allows comments within the statements in order to provide textual explanations of the operations. Whatever is enclosed between /* and */ shall not be processed by VTL parsers, as it shall be considered as comment.

For example:

```
/* Set constant for ‘π’*/
numpi := 3.14
popA := populationDS + 1 /* Assign temp Dataset popA */
```

Constraints and errors

VTL supports a number of errors, which can occur in different situations; errors are divided into three main categories **compile time, runtime, validation.** Each category is divided in turn in subcategories, containing the specific errors.

An error is identified by the string “VTL-“ followed by a four digit code CSEE, where:

- C identifies the category (0: compile time, 1: runtime, 2: validation)
- S identifies the subcategory
- EE identifies the specific error in the subcategory

While the three categories (and subcategories for compile errors) are standardized with codes reported in the remainder of this section, an encoding for specific errors (identified by the last two digits, EE) is not enforced here and can be independently defined by the adopting organization.[14]

A compile time error prevents an expression from being used (exchanged, executed …) and results in an exception reporting the error code (VTL-0XXX) and the wrong expression to the definer.

In contrast, when a runtime error is raised, it can cause:

a) an abnormal termination of the running VTL program, with an exception reporting the error code (VTL-1XXX) and the wrong expression to the user
b) the current expression to be discarded, without generating any exception
c) only the violating Data Point to be discarded, without generating any exception.

The choice between these three behaviours should be dependent on the runtime system and is not part of the language, nor linked to the error codes.

Validation errors are errors resulting from data validation (e.g. *check* operator), which can be stored in Datasets and used for further elaboration. Indeed, validation errors are not VTL errors and do not influence the use of the expression or the normal execution of a VTL program.

Compile Time errors (VTL-0xxx)

The VTL grammar specifies the rules to be followed in writing expressions. The VTL language allows the detection at compile time of the possible violation of the **correct syntax**, the use of

[14] However, notice that in a following version of the language, a standardization is foreseen also for subcategories and specific error codes.

wrong types as parameters for the operators or the **violation of any of the static constraints of the operators** (with respect to the rules described in the Part 2).

A VTL compiler has to be able to detect all the syntax errors, help the user understand the reason and recover. Three subcategories are predetermined (see below). The specific error can be represented by the adopting organization with any code ranging from 00 to 99 (examples are: unclosed literal string; unexpected symbol, etc.)

Syntax errors (VTL-01xx)

A violation of the VTL syntax with respect to the syntax templates of operators in name of operators or number of operands.

Examples of syntactically invalid expressions are:

`R := C1 +` –the second operand is missing

`R := C1 exist_in_all C2` - the correct syntax is “exists_in_all”.

`R := if k1>4 then else K3 + 3` – the “then” operand is missing

Type errors (VTL-02xx)

A violation of of the types of the operands allowed for the operators.

Examples of expressions that are type-invalid are:

`R := C1 + '2'` – if C1 has a measure component that is not <String>

`R := C1 + C2` – if C1 has a MeasureComponent<String> and C2 has a MeasureComponent<Numeric>

`R := C1 / 5` - if C1 has a MeasureComponent<String>.

`R:= if (K1 > 3 and k1 < 5) then 0 else "hello"` – the “then” and the “else” operands must be of the same type

Since the language is strongly typed, all type violations can be reported at compile time.

Static constraint violation errors (VTL-03xx)

Every operator may have additional constraints. They are reported in the respective “Constraints” sections in the Part 2. Some of them are static, in the sense that they can be checked at compile type.

A constraint violation error is the violation of a static VTL constraint .

Examples of expressions that violate static constraints are:

`R := C1 + C2` – if the IdentifierComponents of C1 and C2 are not the same or are not contained in the ones of the other operator.

`R := 3 + 5` – in the plus (+) operator, at least one operand must be a Dataset.

Runtime errors (VTL-1xxx)

These are the errors that can be detected only at runtime, typically because they are generated by the data.

Examples are the classical mathematical constraints on operators arguments (negative or zero logarithm argument, division by zero, etc.).

Particular types of runtime errors are:

- presence of **duplicate** Data Points to be assigned to a Data Set (it is not allowed that two Data Points in a Data Set have the same values for all the Identifier Components because the Data Point identification would be impossible)
- presence of a **NULL value** in an Identifier Component of a Data Point.

These two errors result in a runtime exception only if the inconsistent Data Points are assigned (:=) to a Data Set in the left-hand side of a Transformation or are stored in a persistent Data Set. In other words, if such Data Points are only partial and temporary results inside the expression on the right-hand side, no runtime exceptions will be raised provided that the anomalies (duplications or NULLS) are removed before the execution of the assignment or the Put command.

Examples of expressions generating runtime errors are:

`R := C1 / C2` – where C2 is 0 for any observation

`R := substr(A, 2, 5)` – if A is 1 character long, causing an "out of range"

`R := C1` – if C1 contains NULL values for some IdentifierComponents. Notice that the assignment causes the runtime error; the fact that C1 contains a NULL value for an IdentifierComponent is accepted as partial and temporary result in the right-hand side of the expression.

`R := C1` – if C1 contains duplicates on an IdentifierComponent. Also in this case, notice that the assignment causes the runtime error; the fact that C1 contains a duplicate is accepted as partial and temporary result in the right-hand side of the expression.

A VTL runtime environment will be able to detect a wide number of runtime errors. The specific errors can be divided into subcategories by the adopting organization; moreover, the specific error can be represented with any code ranging from 00 to 99.

Validation errors (VTL-2xxx)

They represent the outcome of a failed user-defined validation. The code can be used for further elaboration or to report discrepancies.

Error codes can be associated with the single validations with the *check* operator, whose last parameter is *errorCode*. This is the code to be used for each Data Point having FALSE for its MeasureComponent.

For example:

```
R := check(C1 >= C2, all, 2601)
```

Checks if C1 is greater or equal than C2 and, if not the case, stores the code 2601 in the *errorCode* attribute.

C1		
K1	K2	M1
1	A	1000
2	B	200

C2			
K1	K2	K3	M1
1	A	X	1000
2	B	Y	350
2	B	Z	150

and produces:

R				
K1	K2	K3	CONDITION	*ERRORCODE*
1	A	X	TRUE	
2	B	Y	FALSE	2601
2	B	Z	TRUE	

A set of VTL validation rules, will be able to detect a wide number of validation errors. The specific errors can be divided into subcategories by the adopting organization; moreover, the specific error can be represented with any code ranging from 00 to 99.

Governance, other requirements and future work

The SDMX Technical Working Group, as mandated by the SDMX Secretariat, is ensuring the technical maintenance of the Validation and Transformation Language through a dedicated VTL task-force. The VTL task-force is open to the participation of experts from other standardisation communities, such as DDI and GSIM.

As the language is designed to be usable within different standards (SDMX, DDI, GSIM), a wider body could in future take on the task of synchronising and coordinating any parallel development. The detailed elements of a wider governance would need to be developed and shared with the other interested communities (e.g. GSIM, DDI, ESS, ESCB,…). Each community has its own governance rules and processes, and attention should be given to creating a system which may ensure a good representation of users' needs together with sound technical governance.

A number of comments, suggestions and other requirements have been submitted to the VTL task force in order to enhance the current VTL 1.0 version. The outcome of a preliminary discussion of these requirements is presented here.

The governance of the extensions

According to the requirements, it is envisaged that the language can be enriched and made more powerful in future versions according to the evolution of the business needs. For example, new operators and clauses can be added, and the language syntax can be upgraded.

The VTL governance body will take care of the evolution process, collecting and prioritising the requirements, planning and designing the improvements, releasing future VTL versions.

The release of new VTL versions is considered as the preferred method of fulfilling the requirements of the user communities. This way, in fact, the possibility of exchanging standard validation and transformation rules would be preserved to the maximum extent possible.

In order to fulfil specific calculation features not yet supported, the VTL provides for a specific operator (Evaluate) whose purpose is to invoke an external calculation function (routine), provided that this is compatible with the VTL IM and data types.

The operator “Evaluate” (also “Eval”) allows defining and making customized calculations (also reusing existing routines) without upgrading or extending the language, because the external calculation function is not considered as an additional operator. The expressions containing Eval are standard VTL expressions and can be parsed through a standard parser. For this reason, when it is not possible or convenient to use other VTL operators, Eval is the recommended method of customizing the language operations.

However, as explained in the section “Extensibility and Customizability” of the “General Characteristics of VTL” above, calling external functions has some drawbacks in respect to the use of the proper VTL operators. The transformation rules would be not understandable unless such external functions are properly documented and shared and could become dependent on the IT implementation, less abstract and less user oriented. Moreover, the external functions cannot be parsed (as if they were built through VTL operators) and this could make the expressions more error-prone. External routines should be used only for specific needs and in limited cases, whereas widespread and generic needs should be fulfilled through the operators of the language.

While the "Eval" operator is part of VTL, the invoked external calculation functions are not. Therefore they are considered as customized parts under the governance, and are responsibility and charge of the organizations which use it.

Another possible form of customization is the extension of VTL by means of non-standard operators/clauses. This kind of extension is deprecated, because it would compromise the possibility of sharing validation rules and using common tools (for example, a standard parser would consider an expression containing non-standard operators as in error).

Organizations possibly extending VTL through non-standard operators/clauses would operate on their own total risk and responsibility, also for any possible maintenance activity deriving from VTL modifications.

Relations with the GSIM Information Model

As explained in the section "VTL Information Model", VTL 1.0 is inspired by GSIM 1.1 as much as possible, in order to provide a formal model at business level against which other information models can be mapped, and to facilitate the implementation of VTL with standards like SDMX, DDI and possibly others.

GSIM faces many aspects that are out of the VTL scope; the latter uses only those GSIM artefacts which are strictly related to the representation of validations and transformations. The referenced GSIM artefacts have been assessed against the requirements for VTL and, in some cases, adapted or improved as necessary, as explained earlier. No assessment was made about those GSIM artefacts which are out of the VTL scope.

In respect to GSIM, VTL considers both unit and dimensional data as mathematical functions having a certain structure in term of independent and dependent variables. This leads to a simplification, as unit and dimensional data can be managed in the same way, but it also introduces some slight differences in data representation. The aim of the VTL Task Force is to propose the adoption of this adjustment for the next GSIM versions.

The VTL IM allows defining the Value Domains (as in GSIM) and their subsets (not explicitly envisaged in GSIM), needed for validation purposes. In order to be compliant, the GSIM artefacts are used for modelling the Value Domains and a similar structure is used for modelling their subsets. Even in this case, the VTL task force will propose the explicit introduction of the Value Domain Subsets in future GSIM versions.

VTL is based on a model for defining mathematical expressions which is called "Transformation model". GSIM does not have a Transformation model, which is however available in the SDMX IM. The VTL IM has been based on the SDMX Transformation model, with the intention of suggesting its introduction in future GSIM versions.

Some misunderstanding may arise from the fact that GSIM, DDI, SDMX and other standards also have a Business Process model. The connection between the Transformation model and the Business Process model has been neither analysed nor modelled in VTL 1.0. One reason is that the business process models available in GSIM, DDI and SDMX are not yet fully compatible and univocally mapped.

It is worth nothing that the Transformation and the Business Process models address different matters. In fact, the former allows defining validation and calculation rules in the form of mathematical expressions (like in a spreadsheet) while the latter allows defining a business process, made of tasks to be executed in a certain order. The two models may coexist and be used together as complementary. For example, a certain task of a business process (say the validation of a data set) may require the execution of a certain set of validation rules, expressed through the Transformation model used in VTL. Further progress in this reconciliation is a task which needs some parallel work in GSIM, SDMX and DDI, and could be reflected in a future VTL version.

Future directions

Structural Validation

We can distinguish two general types of validation according to their goals: “structural validation” and “content validation”, i.e. validation of the information content. The former can be defined as the assurance that data observations are compliant with the desired data structure, the latter that the data give a good representation of the phenomena under investigation.

As both DDI and SDMX provide for structural metadata which allow structural validation, the VTL Task Force discussed whether VTL has to support structural validation or not. The conclusion was affirmative, considering that the use of different kinds of structural metadata is not homogeneous among organizations and among implementing standards and that it could be useful to support all kind of validations using the same method.

It has been acknowledged, however, that this makes it possible to express structural validation rules in two alternative ways: through structural metadata or through VTL rules. Obviously, different choices by different organizations might compromise the possibility of exchanging, understanding and applying validation rules defined by others: the two forms of expressing structural validation rules should be made equivalent, in order to make it possible to transform one into the other, if needed.

This VTL 1.0 version supports structural validation but does not provide yet for an equivalence with and easy conversion of structural metadata. This topic is intended to be covered in future work for a following VTL version.

Reusable rules

A main requirement expressed in the VTL public consultation is to allow generic and reusable rules, in order to apply the same rule in many cases. A typical example is to check that the values of a certain variable belong to a certain set of values.

In VTL 1.0, such rules have to be written for each case. As structural metadata are typically reusable, only the structural validation rules defined through structural metadata are reusable at the moment.

Reusable rules will be supported in a following VTL version, also through the use of “macro” operators (new operators defined by combining the existing ones).

Other operators

In the VTL public consultation, some other kinds of operators have been requested (in addition to the “macro” operators already mentioned). For example, it was highlighted the lack of operators to manipulate dates and times, to convert different units of measure and to deal with time series. Operators of these kinds will be introduced in a following VTL version.

It was also underlined that sometimes it is not easy to understand how to perform some kind of data manipulation. For example the possibility of converting the codes (from a coding system to another) is in some way “hidden” in the hierarchy operator. Cases of this kind may lead to a more explicit documentation or the introduction of more specific operators.

Annex 1 - EBNF

The VTL 1.0 language is also expressed in EBNF (Extended Backus-Naur Form).

EBNF is a standard[15] meta-syntax notation, typically used to describe a Context-Free grammar and represents an extension to BNF (Backus-Naur Form) syntax. Indeed, any language described with BNF notation can be also expressed in EBNF (although expressions are typically lengthier).

Intuitively, an EBNF consists of terminal symbols and non-terminal production rules. Terminal symbols are the alphanumeric characters (but also punctuation marks, whitespace, etc.) that are allowed singularly or in a combined fashion. Production rules are the rules governing how terminal symbols can be combined in order to produce words of the language (i.e. legal sequences).

More details about EBNF notation can be found on:

http://en.wikipedia.org/wiki/Extended_Backus–Naur_Form

Properties of VTL grammar

VTL can be described in terms of a Context-Free grammar[16], with productions of the form $V \rightarrow w$, where V is a single non-terminal symbol and w is a string of terminal and non-terminal symbols.

VTL grammar aims at being unambiguous. An ambiguous Context-Free grammar is such that there exists a string that can be derived with two different paths of production rules, technically with two different leftmost derivations.

In theoretical computer science, the problem of understanding if a grammar is ambiguous is undecidable. In practice, many languages adopt a number of strategies to cope with ambiguities. This is the approach followed in VTL as well. Examples are: the presence of *associativity* and *precedence* rules for infix operators (such as addition and subtraction); the existence of compulsory *else* branch in *if-then-else* operator.

These devices are reasonably good to guarantee the absence of ambiguity in VTL grammar. Indeed, real parser generators (for instance YACC[17]), can effectively exploit them, in particular using the mentioned associativity and precedence constrains as well as the relative ordering of the productions in the grammar itself, which solves ambiguity by default.

---

[15] ISO/IEC 14977

[16] http://en.wikipedia.org/wiki/Context-free_grammar

[17] http://en.wikipedia.org/wiki/Yacc