SDMX Technical Working Group

VTL Task Force

VTL - version 2.1

(Validation & Transformation Language)

Part 1 - User Manual

July 2024

Foreword

The Task force for the Validation and Transformation Language (VTL), created in 2012-2013 under the initiative of the SDMX Secretariat, is pleased to present the version 2.1 of VTL.

The SDMX Secretariat launched the VTL work at the end of 2012, moving on from the consideration that SDMX already had a package for transformations and expressions in its information model, while a specific implementation language was missing. To make this framework operational, a standard language for defining validation and transformation rules (operators, their syntax and semantics) has been adopted.

The VTL task force was set up early in 2013, composed of members of SDMX, DDI and GSIM communities and the work started in summer 2013. The intention was to provide a language usable by statisticians to express logical validation rules and transformations on data, described as either dimensional tables or unit-record data. The assumption is that this logical formalization of validation and transformation rules could be converted into specific programming languages for execution (SAS, R, Java, SQL, etc.), and would provide at the same time, a "neutral" business-level expression 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 kind of 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 (VTL IM), taking the common parts of GSIM, SDMX and DDI, mainly using artefacts from GSIM, somewhat simplified and with some additional detail. In this way, existing standards (GSIM, SDMX, DDI, others) would be allowed to adopt VTL by mapping their information model against the VTL IM. Therefore, although a work-product of SDMX, the VTL language in itself is independent of SDMX and will be usable with other standards as well. Thanks to the possibility of being mapped with the basic part of the IM of other standards, the VTL IM also makes it possible to collect and manage the basic definitions of data represented in different standards.

For the reason described above, the VTL specifications are designed at logical level, independently of any other standard, including SDMX. The VTL specifications, therefore, are self-standing and can be implemented either on their own or by other standards (including SDMX).

The first public consultation on VTL (version 1.0) was held in 2014. Many comments were incorporated in the VTL 1.0 version, published in March 2015. Other suggestions for improving the language, received afterwards, fed the discussion for building the draft version 1.1, which contained many new features, was completed in the second half of 2016 and provided for public consultation until the beginning of 2017.

The high number and wide impact of comments and suggestions induced a high workload on the VTL TF, which agreed to proceed in two steps for the publication of the final documentation. The first step has been dedicated to fixing some high-priority features and making them as much stable as possible; given the high number of changes, it was decided that the new version should be considered as a major one and thus named VTL 2.0.

The second step, taking also into consideration that some VTL implementation initiatives are already in place, is aimed at acknowledging and fixing other features considered of minor impact and priority, without affecting the features already published or the possible relevant implementations.

In parallel with the work for designing the new VTL version, the task force has been involved in the SDMX implementation of VTL, aiming at defining formats for exchanging rules and developing web services to retrieve them; the new features have been included in the SDMX 3.0 package.

The present VTL 2.1 package contains the general VTL specifications, independently of the possible implementations of other standards; it includes:

a) The User Manual, highlighting the main characteristics of VTL, its core assumptions and the information model the language is based on;
b) The Reference manual, containing the full library of operators ordered by category, including examples;
c) eBNF notation (extended Backus-Naur Form) which is the technical notation to be used as a test bed for all the examples;
d) A Technical Notes document, containing some guidelines for VTL implementation.

The latest version of VTL is freely available online at https://sdmx.org/?page_id=5096

Acknowledgements

The VTL specifications have been prepared thanks to the collective input of experts from Bank of Italy, Bank for International Settlements (BIS), European Central Bank (ECB), Eurostat, ILO, INEGI-Mexico, INSEE-France, 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 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, Franck Cotton, Vincenzo Del Vecchio, Fabio Di Giovanni, Jens Dossé, Heinrich Ehrmann, Bryan Fitzpatrick, Tjalling Gelsema, Luca Gramaglia, Arofan Gregory, Gyorgy Gyomai, Edgardo Greising, Dragan Ivanovic, Angelo Linardi, Juan Munoz, Chris Nelson, Stratos Nikoloutsos, Antonio Olleros, Stefano Pambianco, Marco Pellegrino, Michele Romanelli, Juan Alberto Sanchez, Roberto Sannino, Angel Simon Delgado, Daniel Suranyi, Olav ten Bosch, Laura Vignola, Fernando Wagener and Nikolaos Zisimos.

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

Table of contents

Introduction

This document presents the Validation and Transformation Language (also known as 'VTL') version 2.1.

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

The first development of VTL aims at enabling, as a priority, the formalisation of data validation algorithms rather than tackling more complex algorithms for data compilation. In fact, the assessment of business cases showed that the majority of the institutions ascribes (prescribes) 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 a second iteration of the first phase, and therefore still presents a version of VTL primarily oriented to support the data validation. However, as the features needed for validation also include simple calculations, this version of VTL can support basic compilation needs as well. In general, validation is considered as a particular case of transformation; therefore, the term "Transformation" is meant to be more general, including validation as well. The actual operators included in this version of VTL are described in the Reference Manual.

Although VTL is developed under the umbrella of the SDMX governance, 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 Official Statistics (HLG-MOS) expressed their wish of adopting VTL as a unique language, usable in SDMX, DDI and GSIM.

Accordingly, the task-force working for the VTL development agreed on the objective of adopting a common language, in the hope of avoiding the risk of having diverging variants.

Consequently, 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 can inherit the VTL language by unequivocally mapping their artefacts to those of the VTL IM.

Structure of the document

The following main sections of the document describe the following topics:

The general characteristics of the VTL, which are also the main requirements that the VTL is aimed to fulfil.

The changes of VTL 2.x in respect to VTL 1.0 and a section for changes for version 2.1.

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

The Data Types that the VTL manipulates, i.e. types of artefacts i.e. types of artefacts that can be passed in input to or are returned in output from the VTL operators.

The general rules for defining the Transformations, which are the algorithms that describe how the operands are transformed into the results.

The characteristics, the invocation and the behaviour of the VTL Operators, taking into account the perspective of users that need to learn how to use them.

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 that were submitted to the task force in order to enhance the VTL package.

A short annex gives some background information about the BNF (Backus-Naur Form) syntax used for providing a context-free representation of VTL.

The Extended BNF (EBNF) representation of the VTL 2.1 package is available at https://sdmx.org/?page_id=5096.

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, even according to hierarchies;
    - Operators for dimensional processing (e.g. projection, filter);
    - Operators for statistics (e.g. aggregation, mean, median, variance …).

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 regard 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 an 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; 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 make it possible to 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 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 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 translated from/to other languages if needed (through the implementation of a proper translator).

[1] See the section "Relationships between VTL and GSIM"

- ⇨ The inputs and the outputs of the calculations and the calculations themselves are artefacts of the IM
    - o This is a basic property of any robust language because it allows calculated data to be operands of further calculations;
    - o 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;
    - o Because the definition of the algorithms of the calculations is based on the definition of their input artefacts (in particular on the data structure of the input data), the latter must be available when the calculation is defined;
    - o 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);
    - o 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 definitions of the calculations, including with respect to the artefact of the IM, and in particular to check:
    - o the correctness of the expressions with respect to the syntax of the language
    - o 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)
    - o the consistency of the overall graph of the calculations (for example, in order to avoid that the result of a calculation goes as input to the same calculation, there should not be cycles in the sequence of calculations, thus eliminating the risk of 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;
    - o 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);
  - The VTL provides only for a logical/conceptual layer for defining the data transformations, which applies on a logical/conceptual layer of data definitions
  - The VTL does not prescribe any technical/physical tool or solution, so that it is possible to implement the VTL by using many different IT tools
  - The link between the logical/conceptual layer of the VTL definitions and the IT implementation layer is out of the scope of the VTL;
- ➪ The language does not require to the users the awareness of the storage data structure; the operations on the data are specified according to the conceptual/logical structure, and so are independent of the storage structure; this ensures that the storage 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 commands for data storage and retrieval at a conceptual/logical level; the mapping and the conversion between the conceptual and the storage structures of the data is left to the IT implementation (and users need not be aware of it);
  - By mapping the logical and the storage 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 is not strictly connected with some specific IT tool to perform the calculations (e.g. SQL, statistical packages, other languages, 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, different 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

- ➪ The language is made of few "core" constructs, which are the fundamental building blocks into which any operation can be decomposed, and a "standard library", which contains a number of standard operators built from the core constructs; these are the standard parts of the language, which can be extended gradually by the VTL maintenance body, enriching the available operators according to the evolution of the business needs, so progressively making the language more powerful;
- ➪ Other organizations can define additional operators having a customized behaviour and a functional syntax, so extending their own library by means of custom-designed operators. As obvious, these additional operators are not part of the standard VTL library. To exchange VTL definitions with other institutions, the possible custom libraries need to be pre-emptively shared.
- ➪ 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. In this case:
    - o 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
    - o The external routines are not part of the language, so their use is subject to some limitations (e.g. it is impossible to parse them as if they were operators of the language)
    - o The use of external routines compromises 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;
- ➪ Whilst an Organisation adopting VTL can extend it by defining customized parts, on its own total responsibility, in order to improve the standard language for specific purposes (e.g. for supporting possible algorithms not permitted by the standard part), it is important that the customized parts remain compliant with the VTL IM and the VTL fundamentals. Adopting Organizations are totally in charge of any activity for maintaining and sharing their customized parts. Adopting Organizations are also totally in charge of any possible maintenance activity to maintain the compliance between their customized parts and the possible VTL future versions.

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[2]), 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 their structure components (Identifiers, Measures, Attributes), on their 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 Sets to be validated and producing a Data Set containing information about the result of the validation;
- ⇨ Calculations on multiple measures are supported by most operators, 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 real world historical changes which induce changes of the artefacts (e.g. of the code lists, of the hierarchies …); however, because different standards may represent historical changes in different ways, the implementation of this aspect is left to the standards (e.g. SDMX, DDI …), to the institutions and to the implementers adopting the VTL and therefore the VTL specifications does not prescribe any particular methodology for representing the historical changes of the artefacts (e.g. versioning, qualification of time validity);
- ⇨ Almost all the VTL operators can be nested, meaning that in the invocation of an operator any operand can be the result of the invocation of other operators which calculate it;
- ⇨ The results of the calculations can be permanently stored or not, according to the needs.

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

Evolution of VTL 2.0 in respect to VTL 1.0

Important contributions gave origin to the work that brought to the VTL 2.0 and now to this VTL 2.1 version.

Firstly, it was not possible to acknowledge immediately - in VTL 1.0 - all of the remarks received during the version 1.0 public review. Secondly, the publication of VTL version 1.0 triggered the launch of other reviews and proofs of concepts, by several institutions and organizations, aimed at assessing the ability of VTL of supporting properly their real use cases.

The suggestions coming from these activities had a fundamental role in designing the new version of the language.

The main improvements are described below.

The Information Model

The VTL Information Model describes the artefacts that VTL manipulates (i.e. it provides generic models for defining Data and their structures, Variables, Value Domains and so on) and how the VTL is used to define validations and transformations (i.e. a generic model for Transformations).

In VTL 2.0 some mistakes of VTL 1.0 have been corrected and new kinds of artefacts have been introduced in order to make the representation more complete and to facilitate the mapping with the artefacts of other standards (e.g. SDMX, DDI …).

As already said, VTL is intended to operate at logical/conceptual level and independently of the implementation, actually allowing different implementations. For this reason, VTL-IM provides only for a core abstract view of data and calculations and leaves out the implementation aspects.

Some other aspects, even if logically related to the representation of data and calculations, are intentionally left out because they can depend on the actual implementation too. Some of them are mentioned hereinafter (for example the representation of real-world historical changes that impact model artefacts).

The operational metadata needed for supporting real processing systems are also out of VTL scope.

The implementation of the VTL-IM abstract model artefacts needs to take into account the specificities of the standards (like SDMX, DDI …) and the information systems for which it is used.

Structural artefacts and reusable rules

The structural artefacts of the VTL IM (e.g. a set of code items) as well as the artefacts of other existing standards (like SDMX, DDI, or others) are intrinsically reusable. These so-called "structural" artefacts can be referenced as many times as needed.

In order to empower the capability of reusing definitions, a main requirement for VTL 2.0 has been the introduction of reusable rules (for example, validation or aggregation rules defined once and applicable to different cases).

The reusable rules are defined through the VTL definition language and applied through the VTL manipulation language.

The core language and the standard library

VTL 1.0 contains a flat list of operators, in principle not related to one another. A main suggestion for VTL 2.0 was to identify a core set of primitive operators able to express all of the other operators present in the language. This was done in order to specify the semantics of available operators more formally, avoiding possible ambiguities about their behaviour and fostering coherent implementations. The distinction between 'core' and 'standard' library is not important to the VTL users but is largely of interest of the VTL technical implementers.

The suggestion above has been acknowledged, so VTL 2.0 manipulation language consists of a core set of primitive operators and a standard library of derived operators, definable in term of the primitive ones. The standard library contains essentially the VTL 1 operators (possibly enhanced) and the new operators introduced with VTL 2.0 (see below).

In particular, the VTL core includes an operator called "join" which allows extending the common scalar operations to the Data Sets.

The user defined operators

VTL 1.0 does not allow defining new operators from existing ones, and thus the possible operators are predetermined. Besides, thanks to the core operators and the standard library, VTL 2.0 allows to define new operators (also called "user-defined operators") starting from existing ones. This is achieved by means of a specific statement of the VTL-DL (the "define operator" statement, see the Reference Manual).

This a main mechanism to enforce the requirements of having an extensible and customizable language and to introduce custom operators (not existing in the standard library) for specific purposes.

As obvious, because the user-defined operators are not part of the standard library, they are not standard VTL operators and are applicable only in the context in which they have been defined. In particular, if there is the need of applying user-defined operators in other contexts, their definitions need to be pre-emptively shared.

The VTL Definition Language

VTL 1.0 contains only a manipulation language (VTL-ML), which allows specifying the transformations of the VTL artefacts by means of expressions.

A VTL Definition Language (VTL-DL) has been introduced in version 2.0.

In fact, VTL 2.0 allows reusable rules and user-defined operators, which do not exist in VTL 1.0 and need to be defined beforehand in order to be invoked in the expressions of the VTL manipulation language. The VTL-DL provides for their definition.

Second, VTL 1.0 was initially intended to work on top of an existing standard, such as SDMX, DDI or other, and therefore the definition of the artefacts to be manipulated (Data and their structures, Variables, Value Domains and so on) was assumed to be made using the implementing standards and not VTL itself.

During the work for the VTL 1.1 draft version, it was proposed to make the VTL definition language able to define also those VTL-IM artefacts that have to be manipulated. A draft version of a possible artefacts definition language was included in VTL 1.1 public consultation, held until the beginning of 2017. The comments received and the following analysis evidenced that the artefact definition language cannot include the aspects that are left out of the IM (for example the representation of the historical changes of the real world impacting the model artefacts) yet are:

i. needed in the implementations;
ii. influenced by other implementation-specific aspects;
iii. in real applications, bound to be extended by means of other context-related metadata and adapted to the specific environment.

In conclusion, the artefact definition language has been excluded from this VTL version and the opportunity of introducing it will be further explored in the near future.

In respect to VTL 1.0, VTL 2.0 definition language (VTL-DL) is completely new (there is no definition language in VTL 1.0).

The functional paradigm

In the VTL Information Model, the various types of statistical data are considered as mathematical functions, having independent variables (Identifiers) and dependent variables (Measures, Attributes), whose extensions can be thought of as logical tables (Data Sets) made of rows (Data Points) and columns (Identifiers, Measures, Attributes). Therefore, the main artefacts to be manipulated using VTL are the logical Data Sets, i.e. first-order mathematical functions[3].

Accordingly, VTL uses a functional programming paradigm, meaning a paradigm that treats computations as the evaluation of higher-order mathematical functions[4], which manipulate the first-order ones (i.e., the logical Data Sets), also termed “operators” or “functionals”. The functional paradigm avoids changing-state and mutable data and makes use of expressions for defining calculations.

It was observed, however, that the functional paradigm was not sufficiently achieved in VTL 1.0 because in some particular cases a few operators could have produced non- functional

[3] A first-order function is a function that does not take other functions as arguments and does not provide another function as result.

[4] A higher-order function is a function that takes one or more other functions as arguments and/or provides another function as result.

results. In effects, even if this regarded only temporary results (not persistent), in specific cases, this behaviour could have led to unexpected results in the subsequent calculation chain.

Accordingly, some VTL 1.0 operators have been revised in order to enforce their functional behaviour.

The operators

The VTL 2.0 manipulation language (VTL-ML) has been upgraded in respect to the VTL 1.0. In fact VTL 2.0 introduces a number of new powerful operators, like the analytical and the aggregate functions, the data points and hierarchy checks, various clauses and so on, and improve many existing operators, first of all the "join", which substitutes the "merge" of the VTL 1.0. The complete list of the VTL 2.0 operators is in the reference manual.

Some rationalisations have brought to the elimination of some operators whose behaviour can be easily reproduced using other operators. Some examples are the "*attrcalc*" operator which is now simply substituted by the already existing "*calc*" and the "query syntax" that was allowed for accessing a subset of Data Points of a Data Set, which on one side was not coherent with the rest of the VTL syntax conventions and on the other side can be easily substituted by the "filter" operator.

Even in respect to the draft VTL 1.1 many rationalisations have been applied, also following the very numerous comments received during the relevant public consultation.

Changes for version 2.1

The VTL 2.1 version is a minor one and contains the following changes in respect to 2.0:

i. typos and errors in the text and/or in the examples have been fixed;
ii. new operators have been defined: time operators (datediff, dateadd, year/month/quarter/dayofmonth/dayofyear, daystoyear, daystomonth, durationtodays), case operator (simple extension of if-then-else), random operator (generating a random decimal number >= 0 and < 1)
iii. some changes have been introduced: the cast operator will have only explicit or implicit mask (no optional mask not allowed), some assumptions have been taken in the ordering for some use cases, the default window clause for analytic operators has been changed to be compliant with the SQL standard behaviour.

A new document (Technical Notes) has been added to the documentation to support VTL implementation.

VTL Information Model

Introduction

The VTL Information Model (IM) is a generic model able to describe the artefacts that VTL can manipulate, i.e. to give the definition of the artefact structure and relationships with other artefacts.

The knowledge of the artefacts definition is essential for parsing VTL expressions and performing VTL operations correctly. Therefore, it is assumed that the referenced artefacts are defined before or at the same time the VTL expressions are defined.

The results of VTL expressions must be defined as well, because it must always be possible to take these results as operands of further expressions to build a chain of transformations as complex as needed. In other words, VTL is meant to be "closed", meaning that operands and results of the VTL expressions are always artefacts of the VTL IM. As already mentioned, the VTL is designed to make it possible to deduce the data structure of the result from the calculation algorithm and the data structure of the operands.

VTL can manage persistent or temporary artefacts, the former stored persistently in the information system, the latter only used temporarily. The definition of the persistent artefact must be persistent as well, while the definition of temporary artefacts can be temporary[5].

The VTL IM provides a formal description at business level of the artefacts that VTL can manipulate, which is the same purpose as the Generic Statistical Information Model (GSIM) with a broader scope. In fact, the VTL Information Model uses GSIM artefacts as much as possible (GSIM 2.0 version)[6]. Note that the description of the GSIM 2.0 classes and relevant definitions can be consulted in the GSIM section of the UNECE site[7]. However, the detailed mapping between the VTL IM and the IMs of the other standards is out of the scope of this document and is left to the competent bodies of the other standards[8].

The VTL IM provides for model at a logical/conceptual level, which is independent of the implementation and allows different possible implementations.

The VTL IM provides for an abstract view of the core artefacts used in the VTL calculations and intentionally leaves out implementation aspects. Some other aspects, even if logically related to the representation of data and calculations, are also left out because they can depend on the actual implementation too (for example, the textual descriptions of the VTL artefacts, the representation of the historical changes of the real world).

[5] The definition of a temporary artefact can be also persistent, if needed.

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

[7] https://unece.org/statistics/modernstats/gsim

[8] Some initiatives have been started by UNECE High-Level Group for the Modernisation of Official Statistics (HLG-MOS); see for example https://unece.org/statistics/documents/2023/11/working-documents/hlg2023-ssg-sdmxvtlddi-implement-gsim.

The operational metadata needed for supporting real processing systems are also left out from the VTL scope (for example the specification of the way data are managed, i.e. collected, stored, validated, calculated/estimated, disseminated ...).

Therefore, the VTL IM cannot autonomously support real processing systems, and for this purpose needs to be properly integrated and adapted, also adding more metadata (e.g., other classes of artefacts, properties of the artefacts, relationships among artefacts …).

Even the possible VTL implementations in other standards (like SDMX and DDI) would require proper adjustments and improvements of the IM described here.

The VTL IM is inspired to the modelling approach that consists in using more modelling levels, in which a model of a certain level models the level below and is an instance of a model of the level above.

For example, assuming conventionally that the level 0 is the level of the real world to be modelled and ignoring possible levels higher than the one of the VTL IM, the VTL modelling levels could be described as follows:

Level 0 – the real world

Level 1 – the extensions of the data that model some aspect of the real world. For example, the content of the data set *"population from United Nations"*:

Year	*Country*	*Population*
2016	*China*	*1,403,500,365*
2016	*India*	*1,324,171,354*
2016	*USA*	*322,179,605*
…		
2017	*China*	*1,409,517,397*
2017	*India*	*1,339,180.127*
2017	*USA*	*324,459,463*
…		

Level 2 – the definitions of specific data structures (and relevant transformations) which are the model of the level 1. An example: *the data structure of the data set "population from United Nations" has one measure component called "population" and two identifier components called Year and Country*.

Level 3 – the VTL Information Model, i.e. the generic model to which the specific data structures (and relevant transformations) must conform. An example of IM rule about the data structure: *a Data Set may be structured by just one Data Structure, a Data Structure may structure any number of Data Sets*.

A similar approach is very largely used, in particular in the information technology and for example by the Object Management Group[9], even if the terminology and the enumeration of the levels is different. The main correspondences are:

VTL Level 1 (extensions)	–	OMG M0 (instances)
VTL Level 2 (definitions)	–	OMG M1 (models)

[9] For example in the Common Warehouse Metamodel and Meta-Object Facility specifications

VTL Level 3 (information model) – OMG M2 (metamodels)

Often the level 1 is seen as the level of the data, the level 2 of the metadata and the level 3 of the meta-metadata, even if the term metadata is too generic and somewhat ambiguous. In fact, “metadata” is any data describing another data, while “definition” is a particular metadata which is the model of another data. For example, referring to the example above, a possible other data set which describes how the population figures are obtained is certainly a metadata, because it gives information about another data (the population data set), but it is not at all its definition, because it does not describe the information structure of the population data set.

The VTL IM is illustrated in the following sections.

The first section describes the generic model for defining the statistical data and their structures, which are the fundamental artefacts to be transformed. In fact, the ultimate goal of the VTL is to act on statistical data to produce other statistical data.

In turn, data items are characterized in terms of variables, value domains, code items and similar artefacts. These are the basic bricks that compose the data structures, fundamental to understand the meaning of the data, ensuring harmonization of different data when needed, validating and processing them. The second section presents the generic model for these kinds of artefacts.

Finally, the VTL transformations, written in the form of mathematical expressions, apply the operators of the language to proper operands in order to obtain the needed results. The third section depicts the generic model of the transformations.

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).

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 …).

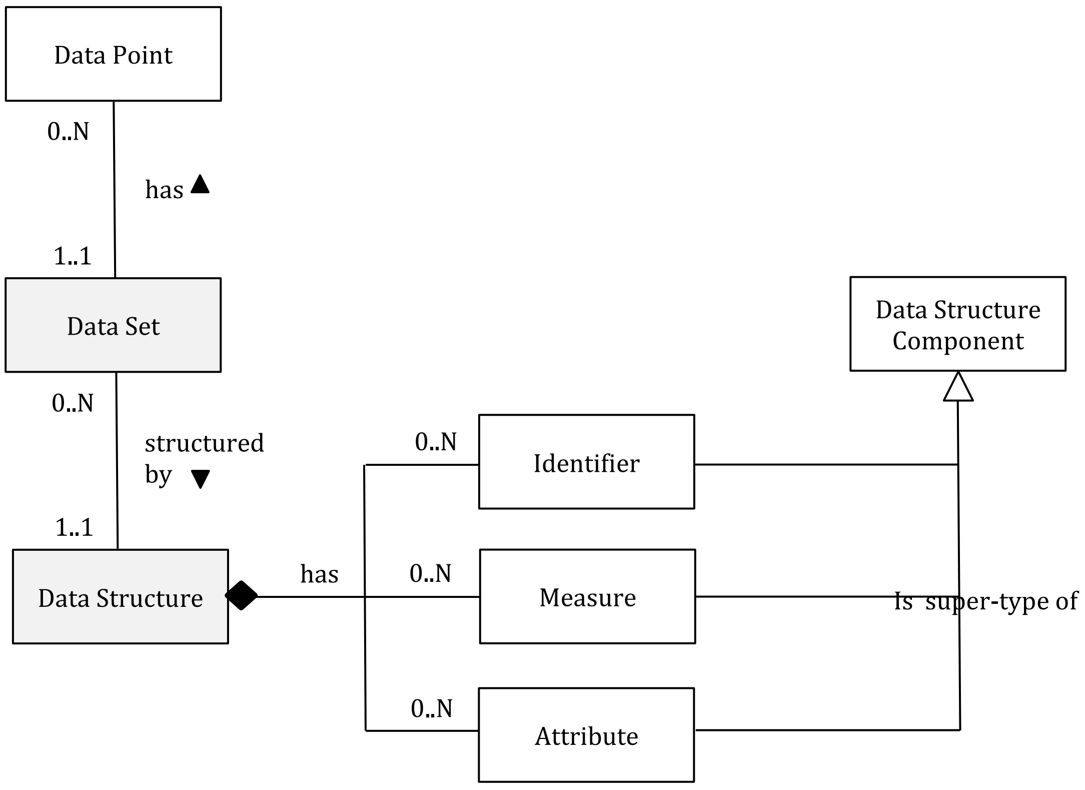
A mathematical function can be seen as a **logical table made of rows and columns**. Each column holds the values of a variable (either independent or dependent); each row holds the association between the values of the independent variables and the values of the dependent variables (in other words,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

In 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 no longer be the need of distinguishing between unit and dimensional data, and in fact VTL does not introduces such a distinction at all. Nevertheless, even if such a distinction is not part of the VTL IM, this aspect is illustrated below in this document in order to make it easier to map the VTL IM to the GSIM IM and the DDI IM, which have such a distinction.

Starting from this assumption, each mathematical function (logical table) may be defined having Identifier, Measure and Attribute Components. The Identifier components are the independent variables of the function, the Measures and Attribute Components are the dependent variables. Obviously, the 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.

In order to avoid any possible misunderstanding with respect to SDMX, also take note that the VTL Data Set in general does not correspond to the SDMX Dataset. In fact, a SDMX dataset is a physical set of data (the data exchanged in a single interaction), while the VTL Data Set is a logical set of data, in principle independent of its possible physical representation and handling (like the exchange of part of it). The right mapping is between the VTL Data Set and the SDMX Dataflow.

Data model diagram



White box: same artefact as in GSIM 2.0
Light grey box: similar to GSIM 2.0

Explanation of the Diagram

Data 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 logical structure and the same general meaning, independently of the possible physical representation or storage. Between the VTL Data Sets and the physical datasets there can be relationships of any cardinality: for example, a VTL Data Set may be stored either in one or in many physical data sets, as well as many VTL Data Sets may be stored in the same physical datasets (or database tables). The mapping between the VTL logical artefacts and the physical artefacts is left to the VTL implementations and is out of scope of this document.

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 logical table that describes the function; therefore, the extension of the function (Data Set) is a set of Data Points. Some Data Points of the function can be unknown (i.e. missing or null), for example, the possible ones relevant to future dates. The single Data Points do not need to be individually defined, because their definition is the definition of the function (i.e. the Data Set definition).

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 conventional and essentially based on their meaning: the Measure Components give information about the real world, while the Attribute components give information about the function itself.

Data Structure Component: any component of the data structure, which can be either an Identifier, or a Measure, or an Attribute Component.

> **Identifier Component** (or simply Identifier): a component of the data structure that is an independent variable of the function.
>
> **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.
>
> **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. In case the automatic propagation of the Attributes is supported (see the section "Behaviour for Attribute Components"), the Attributes can be further classified in normal Attributes (not automatically propagated) and Viral Attributes (automatically propagated).

There can be from 0 to N Identifiers in a Data Structure. A Data Set having no identifiers can contain just one Data Point, whose independent variables are not explicitly represented.

There can be from 0 to N Measures in a Data Structure. A Data Set without Measures is allowed because the Identifiers can be considered as functional dependent from themselves (so having also the role of Measure). In an equivalent way, the combinations of values of the Identifiers can be considered as "true" (i.e. existing), therefore it can be thought that there is an implicit Boolean measure having value "TRUE" for all the Data Points[10].

[10] For example, this is the case of a relationship that does not have properties: imagine a Data Set containing the relationship between the students and the courses that they have followed, without any other information: the corresponding Data Set would have StudentId and CourseId as Identifiers and would not have any explicit measure

The extreme case of a Data Set having no Identifiers, Measures and Attributes is allowed. A Data Set of this kind contains just one scalar Value whose meaning is specified only through the Data Set name. As for the VTL operations, these Data Sets are managed like the scalar Values.

Note that the VTL may manage Measure and Attribute Components in different ways, as explained in the section "The general behaviour of operations on datasets" below, therefore the distinction between Measures and Attributes may be significant for the VTL.

Represented Variable: a characteristic of a statistical population (e.g. the country of birth) represented in a specific way (e.g. through the ISO numeric country code). A represented variable may contribute to define any number of Data Structure Components.

Functional Integrity

The VTL data model requires a functional dependency between the Identifier Components and all the other Components of a Data Set. It follows that a Data Set can also be seen as a tabular structure with a finite number of columns (which correspond to its Components) and rows (which correspond to its individual Data Points), in fact for each combination of values of the Identifier Components' columns (which identify an individual Data Point), there is just one value for each Measure and Attribute (contained in the corresponding columns).

The functional dependency translates into the following *functional integrity* requirements:

- Each Component has a distinct name in the Data Structure of the Data Set and contains one scalar value for each Data Point.
- All the Identifier Components of the Data Set must contain a significant value for all the Data Points (i.e. such value cannot be unknown ("`NULL")`).
- In a Data Set there cannot exist two or more Data Points having the same values for all the Identifier Components (i.e. the same Data Point key).
- When a Measure or Attribute Component has no significant value (i.e. "`NULL")` for a Data Point, it is considered unknown for that Data Point.
- When a Data Point is missing (i.e. a possible combination of values of the independent variables is missing), all its Measure and Attribute Components are by default considered unknown (unless otherwise specified).

The VTL expects the input Data Sets to be functionally integral and is designed to ensure that the resulting Data Set are functionally integral too.

Examples

As a first simple example of Data Sets seen as mathematical functions, 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
…	…	…	…	…

This table is equivalent to a proper mathematical function: in fact, it fulfils the functional integrity requirements above. 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 (Identifier Component - Measure Identifier)
- 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 or contain the "`NULL`" 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 does not fulfil the functional integrity requirements above because its rows (i.e. the Data Points) either have different structures (in term of allowed columns) or have null values in the Identifiers. However, it is easy to recognize that there exist two possible functional 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. Therefore, these would be two distinct logical Data Sets according to the VTL IM, even if stored in the same physical table.

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

VTL 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)

VTL 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)

These examples clarify the meaning of "logical" table or Data Set in VTL, that is a set of data which can be considered as the extensional form of a mathematical function, whichever technical format is used, regardless it is stored or not and, in case, wherever it is stored.

In the example above, one physical data set corresponds to more than one logical VTL Data Sets, with a 1 to many correspondence. In the general case, between physical and logical data sets there can be any correspondence (1 to 1, 1 to many, many to 1, many to many).

The data artefacts

The list of the VTL artefacts related to the manipulation of the data is given here, together with the information that the VTL may need to know about them[11].

For the sake of simplicity, the names of the artefacts can be abbreviated in the VTL manuals (in particular the parts of the names shown between parentheses can be omitted).

As already mentioned, this list provides an abstract view of the core metadata needed for the manipulation of the data structures but leaves out implementation and operational aspects. For example, textual descriptions of the artefacts are left out, as well as any specification of temporal validity of the artefacts, procedural metadata (specification of the way data are processed, i.e., collected, stored, validated, calculated/estimated, disseminated ...) and so on. In order to support real systems, the implementers can conveniently adjust this model to their environments and integrate it by adding additional metadata (e.g. other properties of the artefacts, other classes of artefacts, other relationships among artefacts ...).

Data Set

Data Set name	*name of the Data Set*
Data Structure name	*reference to the data structure of the Data Set*

Data Structure

Data Structure name	*name of the Data Structure (the Structure Components are specified in the following artefact)*

(Data) Structure Component

Data Structure name	*the data structure, which the Data Structure Component belongs to*
Component name	*the name of the Component*
Component Role	*IDENTIFIER or MEASURE or ATTRIBUTE (or also VIRAL ATTRIBUTE if the automatic propagation is supported)*

[11] For example, for ensuring correct operations, the knowledge of the Data Structure of the input Data Sets is essential at parsing time, in order to check the correctness of the VTL expression and determine the Data Structure of the result, and at execution time to perform the calculations

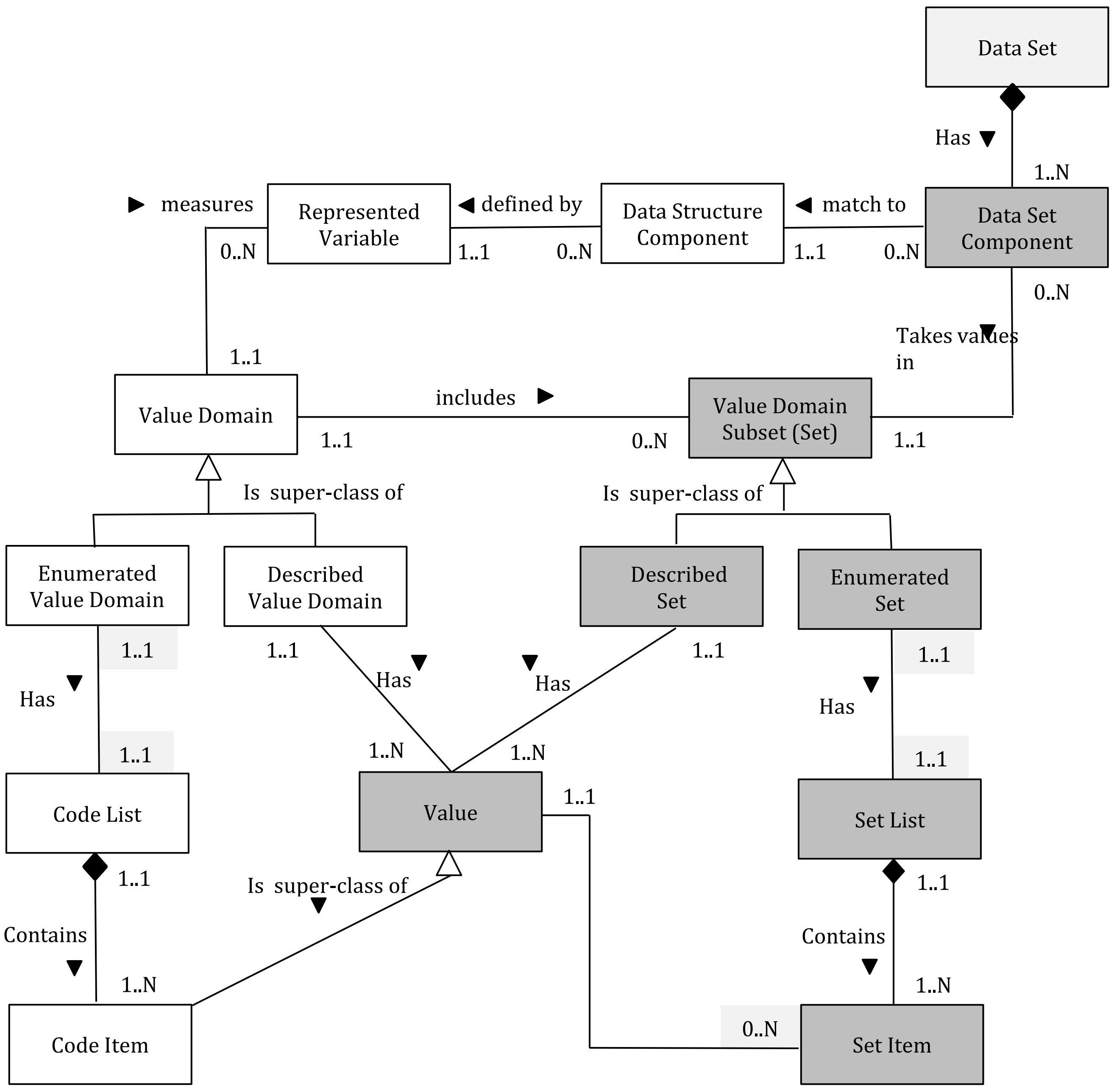
Represented Variable	*the Represented Variable which defines the Component (see also below)*

The Data Points have the same information structure of the Data Sets they belong to; in fact they form the extensions of the relevant Data Sets; VTL does not require defining them explicitly.

Generic Model for Variables and Value Domains

This Section provides a formal model for the Variables, the Value Domains, their Values and the possible (Sub)Sets of Values. These artefacts can be referenced in the definition of the VTL Data Structures and as parameters of some VTL Operators.

Variable and Value Domain model diagram



White box:	same as in GSIM 2.0
Light grey:	similar to GSIM 2.0
Dark grey	additional detail (in respect to GSIM 2.0)

Explanation of the Diagram

The VTL IM distinguishes explicitly between Value Domains and their (Sub)Sets in order to allow different Data Set Components relevant to the same aspect of the reality (e.g. the geographic area) to share the same Value Domain and, at the same time, to take values in different Subsets of it. This is essential for VTL for several operations and in particular for validation purposes. For example, it may happen that the same Represented Variable, say the "place of birth", in a Data Set takes values in the Set of the European Countries, in another one takes values in the set of the African countries, and so on, even at different levels of details (e.g. the regions, the cities). The definition of the exact Set of Values that a Data Set Component can take may be very important for VTL, in particular for validation purposes. The specification of the Set of Values that the Data Set Components may assume is equivalent, on the mathematical plane, to the specification of the domain and the co-domain of the mathematical function corresponding to the Data Set.

Data Set: see the explanation given in the previous section (Generic Model for Data and their structures).

Data Set Component: a component of the Data Set, which matches with just one Data Structure Component of the Data Structure of such a Data Set and takes values in a (sub)set of the corresponding Value Domain[12]; this (sub)set of allowed values may either coincide with the set of all the values belonging to the Value Domain or be a proper subset of it. In respect to a Data Structure Component, a Data Set Component bears the important additional information of the set of allowed values of the Component, which can be different Data Set by Data Set even if their data structure is the same.

Data Structure: a Data Structure; see the explanation already given in the previous section (Generic Model for Data and their structures)

Data Structure Component: a component of a Data Structure; see the explanation already given in the previous section (Generic Model for Data and their structures). A Data Structure Component is defined by a Represented Variable.

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). A represented variable may take value in (or may be measured by) just one Value Domain.

Value Domain: the domain of allowed values for one or more represented variables. Because of the distinction between Value Domain and its Value Domain Subsets, a Value Domain is the wider set of values that can be of interest for representing a certain aspect of the reality like the time, the geographical area, the economic sector and so on. As for the mathematical meaning, a Value Domain is meant to be the representation of a "space of events" with the meaning of the probability theory[13]. Therefore, a single Value of a Value Domain is a representation of a single "event" belonging to this space of events.

[12] This is the Value Domain which measures the Represented Variable, which defines the Data Structure Component, which the Data Set Component matches to

[13] According to the probability theory, a random experiment is a procedure that returns a result belonging a predefined set of possible results (for example, the determination of the "geographic location" may be considered as a random experiment that returns a point of the Earth surface as a result). The "space of results" is the space of

Described Value Domain: a Value Domain defined by a criterion (e.g. the domain of the positive integers).

Enumerated Value Domain: a Value Domain defined by enumeration of the allowed values (e.g. domain of ISO codes of the countries).

Code List: the list of all the Code Items belonging to an enumerated Value Domain, each one representing a single "event" with the meaning of the probability theory. As for its mathematical meaning, this list is unique for a Value Domain, cannot contain repetitions (each Code Item can be present just once) and cannot contain ambiguities (each Code Item must have a univocal meaning, i.e., must represent a single event of the space of the events). The multiplicity of the relationship with the Enumerated Value Domain is 1:1 because, as it happens for the Data Set, the VTL considers the Code List as an artefact at a logical level, corresponding to its mathematical meaning. A logical VTL Code List, however, may be obtained as the composition of more physical lists of codes if needed: the mapping between the logical and the physical lists is out of scope of this document and is left to the implementations, provided that the basic conceptual properties of the VTL Code List are ensured (unicity, no repetitions, no ambiguities). In practice, as for the VTL IM, the Code List artefact matches 1:1 with the Enumerated Value Domain artefact, therefore they can be considered as the same artefact.

Code Item: an allowed Value of an enumerated Value Domain. A Code Item is the association of a Value with the relevant meaning. An example of Code Item is a single country ISO code (the Value) associated to the country it represents (the category). As for the mathematical meaning, a Code Item is the representation of an "event" of a space of events (i.e. the relevant Value Domain), according to the notions of "event" and "space of events" of the probability theory (see the note above).

Value: an allowed value of a Value Domain. Please note that on a logical / mathematical level, both the Described and the Enumerated Value Domains contain Values, the only difference is that the Values of the Enumerated Value Domains are explicitly represented by enumeration, while the Values of the Described Value Domains are implicitly represented through a criterion.

The following artefacts are aimed at representing possible subsets of the Value Domains. This is needed for validation purposes, because very often not all the values of the Value Domain are allowed in a Data Structure Component, but only a subset of them (e.g. not all the countries but only the European countries). This is needed also for transformation purposes, for example to filter the Data Points according to a subset of Values of a certain Data Structure Component (e.g. extract only the European Countries from some data relevant to the World Countries).

Value Domain Subset (or simply **Set**): a subset of Values of a Value Domain. Hereinafter a Value Domain Subset is simply called **Set**, because it can be any set of Values belonging to the Value Domain (even the set of all the values of the Value Domain).

all the possible results. Instead an "event" is a set of results (going back to the example of the geographic location, the event "Europe" is the set of points of the European territory and more in general an "event" corresponds to a "geographical area"). The "space of events" is the space of all the possible "events" (in the example, the space of the geographical areas).

Described Value Domain Subset (or simply **Described Set**): a described (defined by a criterion) subset of Values of a Value Domain (e.g. the countries having more than 100 million inhabitants, the integers between 1 and 100).

Enumerated Value Domain Subset (or simply **Enumerated Set**): an enumerated subset of a Value Domain (e.g. the enumeration of the European countries).

Set List: the list of all the Values belonging to an Enumerated Set (e.g. the list of the ISO codes of the European countries), without repetitions (each Value is present just once). As obvious, these Values must belong to the Value Domain of which the Set is a subset. The Set List enumerates the Values contained in the Set (e.g. the European country codes), without the associated categories (e.g. the names of the countries), because the latter are already maintained in the Code List / Code Items of the relevant Value Domain (which enumerates all the possible Values with the associated categories). In practice, as for the VTL IM, the Set List artefact coincides 1:1 with the Enumerated Set artefact, therefore they can be considered as the same artefact.

Set Item: an allowed Value of an enumerated Set. The Value must belong to the same Value Domain the Set belongs to. Each Set Item refers to just one Value and just one Set. A Value can belong to any number of Sets. A Set can contain any number of Values.

Relations and operations between Code Items

The VTL allows the representation of logical relations between Code Items, considered as events of the probability theory and belonging to the same enumerated Value Domain (space of events). The VTL artefact that allows expressing the Code Item Relations is the Hierarchical Ruleset, which is described in the reference manual.

As already explained, each Code Item is the representation of an event, according to the notions of "event" and "space of events" of the probability theory. The relations between Code Items aim at expressing the logical implications between the events of a space of events (i.e. in a Value Domain). The occurrence of an event, in fact, may imply the occurrence or the non-occurrence of other events. For example:

- The event UnitedKingdom implies the event Europe (e.g. if a person lives in UK he/she also lives in Europe), meaning that the occurrence of the former implies the occurrence of the latter. In other words, the geo-area of UK is included in the geo-area of the Europe.
- The events Belgium, Luxembourg, Netherlands are mutually exclusive (e.g. if a person lives in one of these countries he/she does not live in the other ones), meaning that the occurrence of one of them implies the non-occurrence of the other ones (Belgium AND Luxembourg = impossible event; Belgium AND Netherlands = impossible event; Luxembourg and Netherlands = impossible event). In other words, these three geo-areas do not overlap.
- The occurrence of one of the events Belgium, Netherlands or Luxembourg (i.e. Belgium OR Netherlands OR Luxembourg) implies the occurrence of the event Benelux (e.g. if a person lives in one of these countries he/she also lives in Benelux) and vice-versa (e.g. if a person lives in Benelux, he/she lives in one of these countries). In other words, the union of these three geo-areas coincides with the geo-area of the Benelux.

The logical relationships between Code Items are very useful for validation and transformation purposes. Considering for example some positive and additive data, like for example the population, from the relationships above it can be deduced that:

- The population of United Kingdom should be lower than the population of Europe.
- There is no overlapping between the populations of Belgium, Netherlands and Luxembourg, so that these populations can be added in order to obtain aggregates.
- The sum of the populations of Belgium, Netherlands and Luxembourg gives the population of Benelux.

A **Code Item Relation** is composed by two members, a 1st (left) and a 2nd (right) member. The envisaged types of relations are: "is equal to" (=), "implies" (<), "implies or is equal to" (<=), "is implied by" (>), and "is implied by or is equal to" (>=). "Is equal to" means also "implies and is implied". For example:

UnitedKingdom < Europe means (UnitedKingdom implies Europe)

In other words, this means that if a point of space belongs to United Kingdom it also belongs to Europe.

The left members of a Relation is a single Code Item. The right member can be either a single Code Item, like in the example above, or a logical composition of Code Items: these are the **Code Item Relation Operands**. The logical composition can be defined by means of Operators, whose goal is to compose some Code Items (events) in order to obtain another Code Item (event) as a result. In this simple algebra, two operators are envisaged:

- the logical OR of mutually exclusive Code Items, denoted "+", for example:

  Benelux = Belgium + Luxembourg + Netherlands

  This means that if a point of space belongs to Belgium OR Luxembourg OR Netherlands then it also belongs to Benelux and that if a point of space belongs to Benelux then it also belongs either to Belgium OR to Luxembourg OR to Netherlands (disjunction). In other words, the statement above says that territories of Belgium, Netherland and Luxembourg are non-overlapping and their union is the territory of Benelux. Consequently, as for the additive measures (and being equal the other possible Identifiers), the sum of the measure values referred to Belgium, Luxembourg and Netherlands is equal to the measure value of Benelux.

- the logical complement of an implying Code Item in respect to another Code Item implied by it, denoted "-", for example:

  EUwithoutUK = EuropeanUnion - UnitedKingdom

  In simple words, this means that if a point of space belongs to the European Union and does not belong to the United Kingdom, then it belongs to EUwithoutUK and that if a point of space belongs to EUwithoutUK then it belongs to the European Union and not to the United Kingdom. In other words, the statement above says that territory of the United Kingdom is contained in the territory of the European Union and its complement is the territory of EUwithoutUK. Consequently, considering a positive and additive measure (and being equal the other possible Identifiers), the difference of the measure values referred to EuropeanUnion and UnitedKingdom is equal to the measure value of EUwithoutUK.

Please note that the symbols "+" and "-" do not denote the usual operations of sum and subtraction, but logical operations between Code Items seen as events of the probability theory. In other words, two or more Code Items cannot be summed or subtracted to obtain another Code Item, because they are events (and not numbers), and therefore they can be manipulated only through logical operations like "OR" and "Complement".

Note also that the "+" also acts as a declaration that all the Code Items denoted by "+" are mutually exclusive (i.e. the corresponding events cannot happen at the same time), as well as the "-" acts as a declaration that all the Code Items denoted by "-" are mutually exclusive. Furthermore, the "-" acts also as a declaration that the relevant Code item implies the result of the composition of all the Code Items denoted by the "+".

At intuitive level, the symbol "+" means "*with*" (Benelux = Belgium *with* Luxembourg *with* Netherland) while the symbol "-" means "*without*" (EUwithoutUK = EuropeanUnion *without* UnitedKingdom).

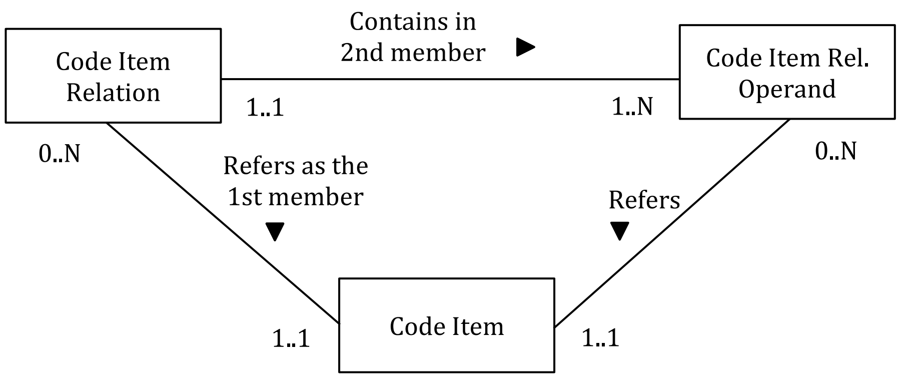
When these relations are applied to additive numeric Measures (e.g. the population relevant to geographical areas), they allow to obtain the Measure Values of the left member Code Items (i.e. the population of Benelux and EUwithoutUK) by summing or subtracting the Measure Values relevant to the component Code Items (i.e. the population of Belgium, Luxembourg and Netherland in the former case, EuropeanUnion and UnitedKingdom in the latter). This is why these logical operations are denoted in VTL through the same symbols as the usual sum and subtraction. Please note also that this is valid whichever the Data Set and the additive Measure are (provided that the possible other Identifiers of the Data Set Structure have the same Values).

These relations occur between Code Items (events) belonging to the same Value Domain (space of events). They are typically aimed at defining aggregation hierarchies, either structured in levels (classifications), or without levels (chains of free aggregations) or a combination of these options. These hierarchies can be recursive, i.e. the aggregated Code Items can in their turn be the components of more aggregated ones, without limitations to the number of recursions.

For example, the following relations are aimed at defining the continents and the whole world in terms of individual countries:

- World = Africa + America + Asia + Europe + Oceania
- Africa = Algeria + … + Zimbabwe
- America = Argentina + … + Venezuela
- Asia = Afghanistan + … + Yemen
- Europe = Albania + … + Vatican City
- Oceania = Australia + … + Vanuatu

A simple model diagram for the Code Item Relations and Code Item Relation Operands is the following:



This diagram tells that a Code Item Relation has a first and a second member. The first member (the left one) refers to just one Code Item, the second member (the right one) may refer to one or more Code Item Relation Operands; each Code Item Relation Operand refers to just one Code Item.

Conditioned Code Item Relations

The Code Items (coded events) of a Code Item Relation can be conditioned by the Values (events) of other Value Domains (spaces of events). Both the Code Items belonging to the first and the second member of the Relation can be conditioned.

A common case is the conditioning relevant to the reference time, which allows expressing the historical validity of a Relation (see also the section about the historical changes below). For example, the European Union (EU) changed its composition in terms of countries many times, therefore the Code Item Relationship between EU and its component countries depends on the reference time, i.e. is conditioned by the Values of the "reference time" Value Domain.

The VTL allows to express the conditionings by means of Boolean expressions which refer to the Values of the conditioning Value Domains (for more details, see the Hierarchical Rulesets in the Reference Manual).

The historical changes

The changes in the real world may induce changes in the artefacts of the VTL-IM and in the relationships between them, so that some definitions may be considered valid only with reference to certain time values. For example, the birth of a new country as well as the split or the merge of existing countries in the real world would induce changes in the Code Items belonging to the Geo Area Value Domain, in the composition of the relevant Sets, in the relationships between the Code Items and so on. The same may obviously happen for other Value Domains.

A correct representation of the historical changes of the artefacts is essential for VTL, because the VTL operations are meant to be consistent with these historical changes, in order to ensure a proper behaviour in relation to each time. With regard to this aspect, VTL must face a complex environment, because it is intended to work also on top of other standards, whose assumptions for representing historical changes may be heterogeneous. Moreover, different institutions may use different conventions in different systems.

Naturally, adopting a common convention for representing the historical changes of the artefacts would be a good practice, because the definitions made by different bodies would be given through the same methodology and therefore would be easily comparable one another. In practice, however, different conventions are already in place and have to be taken into account, because there can also be strong motivations to maintain them. For this reason, the VTL does not impose any definite representation for the historical changes and leaves users free of maintaining their own conventions, which are considered as part of the data content to be processed rather than of the language.

Actually, the VTL-IM intentionally does not include any mechanism for representing historical changes and needs to be properly integrated to this purpose. This aspect is left to the standards and the institutions adopting VTL and the implementers of VTL systems, which can adapt and enrich the VTL-IM as needed.

Even if presented here for association of ideas with the relations between Code Items whose temporal dependency is intuitive, these considerations about the temporal validity of the definitions are valid in general.

Moreover, as already mentioned, the possibility of integrating the VTL-IM with additional metadata is needed also for other purposes, and not only for dealing with the temporal validity.

It is appropriate here to highlight some relationships between the VTL artefacts and some possible temporal conventions, because this can guide VTL implementers in extending the VTL-IM according to their needs.

First, we want to distinguish between two main temporal aspects: the so-called validity time and operational time. Validity time is the time during which a definition is assumed to be true as an abstraction of the real world (for example, Estonia belongs to EU "from 1st May 2004 to current date"). Operational time is the time period during which a definition is available in the processing system and may produce operational effects. The following considerations refers only to the former.

The **assignment of identifiers to the abstractions of the real world** is strictly related to the possible basic temporal assumptions. Two main options can be considered:

a) The same identifier is assigned to the abstraction even if some aspects of such an abstraction change in time. For example, the identifier EU is assigned to the European Union even if the participant countries change. Under this option, a single identifier (e.g. EU) is used to represent the whole history of an abstraction, following the intuitive conceptualization in which abstractions are identified independently of time and maintain the same identity even if they change with time. The variable aspects of an abstraction are therefore described by specifying their validity periods (for example, the participation of Estonia in the EU can be specified through the relevant start and end dates).

b) Different Identifiers are assigned to the abstraction when some aspects of the abstraction change in time. For example, more Identifiers (e.g. EU1, … EU9) represent the European Union, one for each period during which its participant countries remain stable. This option is based on the conceptualization in which the abstractions are identified in connection with the time period in which they do not change, so that a Code Item (e.g. EU1) corresponds to an abstraction (e.g. the European Union) only for the time period in which the abstraction remains stable (e.g. EU1 represents the European Union from when it was created by the founder countries, to the first time it changed composition). An example of adoption of this option b) is the common practice of giving versions to Code Lists or Code Items for representing time changes (e.g. EUv$_1$,... , EUv$_9$ where v=version), being each version assumed as invariable.

Therefore, the general assumptions of VTL for the representation of the historical changes are the following:

- The choice of adopting the options described above is left to the implementations.
- The VTL Identifiers are different depending on the two options above; for example in the option a) there would exist one Identifier for the European Union (e.g. EU) while in the option b) there would exist many different Identifiers, corresponding to the different versions of the European Union (e.g. EU1, … EU9).
- If the Code Items are versioned for managing temporal changes (option b), the version is considered part of the VTL univocal identifier of the Code Item, and therefore different versions are equivalent to different Code Items. As explained above, in fact, the European Union would be represented by many Code Items (e.g. EUv1, … EUv9). The same applies if the Code Items are versioned by means of dates (e.g. start/end dates …) or other conventions instead than version numbers. As obvious, the temporal validity of EUv1 … EUv9, if represented, should not overlap.

The implementers of VTL systems can add the temporal validity (through validity dates or versions) to any class of artefacts or relations of the VTL-IM (as well as any other additional characteristic useful for the implementation, like the textual descriptions of the artefacts or others). If the temporal validity is not added, the occurrences of the class are assumed valid "ever".

The Variables and Value Domains artefacts

The list of the VTL artefacts related to Variables and Value Domains is given here, together with the information that the VTL need to know about them. For the sake of simplicity, the names of some artefacts are often abbreviated in the VTL manuals (in particular the parts of the names shown between parentheses can be omitted).

As already mentioned, this model provides an abstract view of the core metadata supporting the definition of the data structures but leaves out implementation and operational aspects. For example, the textual descriptions of the artefacts are left out, as well as the specification of the temporal validity of the artefacts, the procedural metadata (the specification of the way data are processed i.e. collected, stored, validated, calculated/estimated, disseminated ...) and so on. In order to support real systems, the implementers can conveniently adjust this model and integrate it by adding other metadata (e.g. other properties of the artefacts, other classes of artefacts, other relationships among artefacts …).

(Represented) Variable

Variable name	*name of the Represented Variable*

Value Domain name	*reference to the Value Domain that measures the Variable, i.e. in which the Variable takes values*
(Data Set) ***Component***	
Data Set name	*the Data set which the Component belongs to*
Component name	*the name of the Component*
(Sub) Set name	*reference to the (sub)Set containing the allowed values for the Component*
Value Domain	
Value Domain name	*name of the Value Domain*
Value Domain sub-class	*if it is an Enumerated or Described Value Domain*
Basic Scalar Type	*the basic scalar type of the Values of the Value Domain, for example string, number … and so on (see also the section "VTL data types")*
Value Domain Criterion	*a criterion for restricting the Values of a basic scalar type, for example by specifying a max length of the representation, an upper or/and a lower value, and so on*
Code List	*this artefact is comprised in the previous one, in fact it corresponds one to one to the enumerated Value Domain (see above)*
Value	*this artefact has no explicit representation, because the Values of described Value Domains are not represented by definition, while the Values of the enumerated Value Domains are represented through the Code Item artefact (see below)*
Code Item	*this artefact defines the Code Items of the Enumerated Value Domains*
Value Domain name	*the Value Domain, which the Value belongs to*
Value	*the univocal name of the Value within the Value Domain it belongs to*
*(Value Domain Sub)****Set***	
Value Domain name	*the Value Domain, which the set belongs to*
Set name	*the name of the Set, which must be univocal within the Value Domain*
Set sub-class	*if it is an Enumerated or Described Set*
Set Criterion	*a criterion for identifying the Values belonging to the Set*
Set List	*this artefact is comprised in the previous one, in fact it corresponds one to one to the enumerated Set*

Set Item	*this artefact specifies the Code Items of the Enumerated Sets*
Value Domain name	*reference to the Value Domain which the Set and the Value belongs to*
Set name	*the Set that contains the Value*
Value	*Value element of the Set*

Code Item Relation	
1stMember Domain name	*Value Domain of the first member of the Relation; e.g. Geo_Area*
1stMemberValue	*the first member of the Relation; e.g. Benelux*
1stMemberComposition	*conventional name of the composition method, which distinguishes possible different compositions methods related to the same first member Value. It must be univocal within the 1stMember. Not necessarily, it has to be meaningful; it can be simply a progressive number, e.g. "1"*
Relation Type	*type of relation between the first and the second member, having as possible values =, <, <=, >, >=*

Code Item Relation Operand	
1stMember Domain name	*Value Domain of the first member of the Relation; e.g. Geo_Area*
1stMember Value	*the first member of the Relation; e.g. Benelux*
1stMember Composition	*see the description already given above*
2ndMember Value	*an operand of the Relation; e.g. Belgium*
Operator	*the operator applied on the 2ndMember Value, it can be "+" or "- "; the default is "+"*

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[14], as DDI and GSIM do not have an explicit transformation model at the present time[15]. 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) [16] 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 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 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 derived Data Sets starting from already existing ones.

The general form of a Transformation is the following:

result assignment_operator expression

meaning that the outcome of the evaluation of *expression* in the right-hand side is assigned to the *result of the Transformation* in the left-hand side (typically a Data Set). The assignment operators are two, ":=" and "<-" (for the assignment to a persistent or a non-persistent result, respectively). A very simple example of Transformation is:

D_r <- D_1 (D_r , D_1 are assumed to be Data Sets)

In this Transformation the Data Set D_1 is assigned without changes (i.e. is copied) to D_r, which is persistently stored.

In turn, the *expression* in the right-hand side composes some operands (e.g., some input Data Sets, but also Sets or other artefacts) 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)

In this example, the measure values of the Data Set D_r are calculated as the sum of the measure values of the Data Sets D_1 and D_2, by composing the Data Points having the same Values for the Identifiers. In this case, D_r is not persistently stored.

[14] The SDMX specification can be found at https://sdmx.org/?page_id=5008 (see Section 2 - Information Model, package 13 - "Transformations and Expressions").

[15] The Transformation model described here is not a model of the processes, like the ones that both SDMX and GSIM have, and has a different scope. The mapping between the VTL Transformation and the Process models is not covered by the present document.

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

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 and their behaviours are described in the VTL reference manual.

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[17], 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 formal language to be used. In 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. This is a generic and formal model for defining Transformations of data through mathematical expressions, which in this case is applied to the VTL, 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 without impact on this generic model.

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.

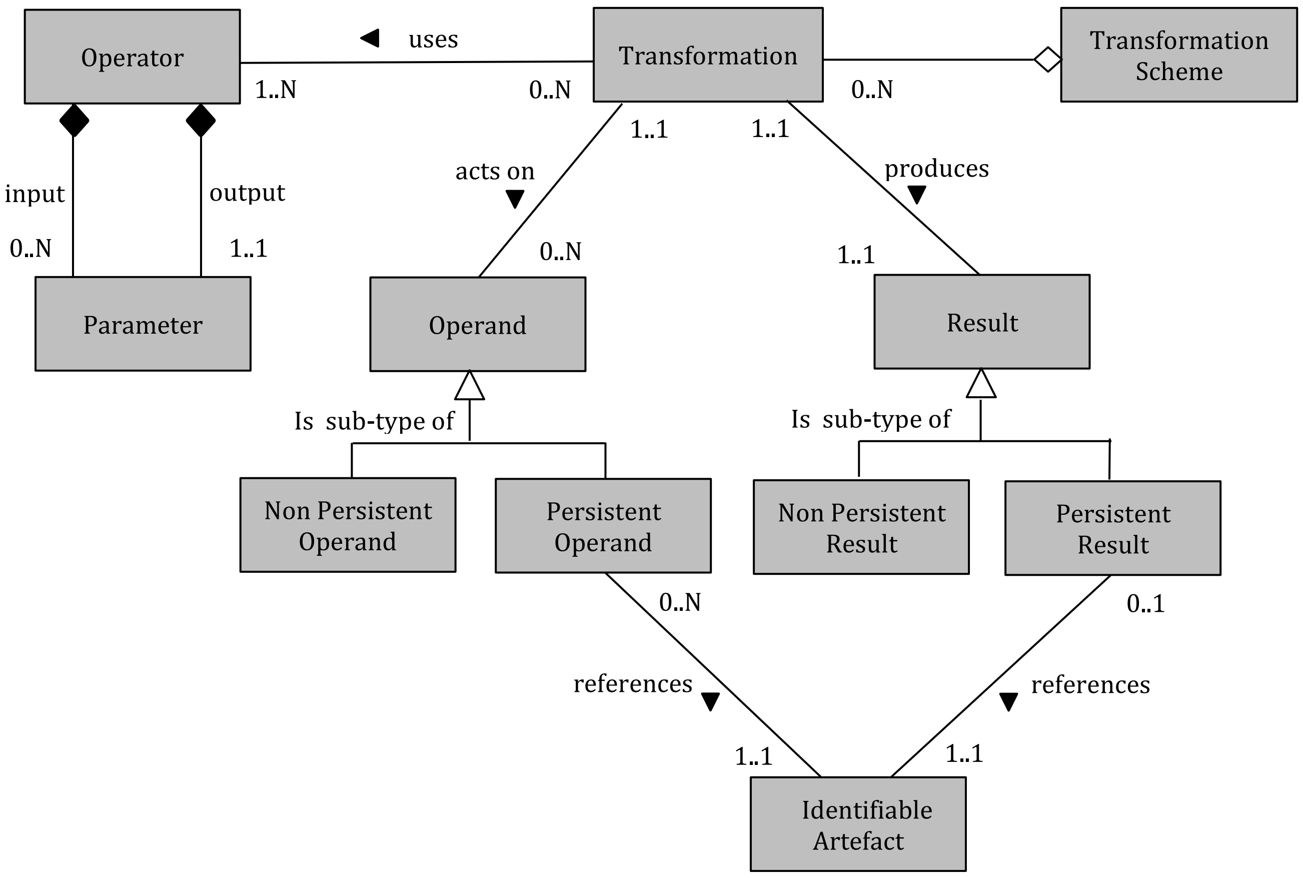
Moreover, the Transformations can be grouped into Transformations Schemes, which are sets of Transformations meaningful to the users. For example, a Transformation Scheme can be the set of Transformations needed to obtain some specific meaningful results, like the validations of one or more Data Sets. A Transformation Scheme is meant to be the smaller set of Transformations to be executed in the same run.

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.

The ability of generating this graph is a main feature of the VTL, because the graph documents the operations performed on the data, just like a spreadsheet documents the operations among its cells.

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

Transformations model diagram



White box:	same as in GSIM 2.0
Dark grey box:	additional detail (in respect to GSIM 2.0)

Explanation of the diagram

Transformation: the basic element of the calculations, which consists of a statement that assigns the outcome of the evaluation of an Expression to an Artefact of the Information Model;

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 text string and is a property of a Transformation;

Transformation Scheme: a set of Transformations aimed at obtaining some meaningful results for the user (like the validation of one or more Data Sets); the Transformation Scheme is meant to be the smaller set of Transformation to be executed in the same run and therefore may also be considered as a VTL program;

Operator: the specification of a type of operation to be performed on some Operands (e.g. sum (+), subtraction (-), multiplication (*), division (/));

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" …), for a deeper explanation see also the Data Type section below. When an Operator is invoked, the actual input passed in correspondence to a certain input Parameter, or the actual output returned by the Operator, is called Argument.

Operand: a specific Artefact referenced in the expression as an input (e.g. a specific input Data Set); a Persistent Operand references a persistent artefact, i.e. an artefact maintained in a persistent storage, while a Non Persistent Operand references a temporary artefact, which is produced by another Transformation and not stored.

Result: a specific Artefact to which the result of the expression is assigned (e.g. the calculated Data Set); a Persistent Result is put away in a persistent storage while a Non Persistent Result is not stored.

Identifiable Artefact: a persistent Identifiable Artefact of the VTL information model (e.g. a persistent Data Set); a persistent artefact can be operand of any number of Transformation but can be the result of no more than one Transformation.

Examples

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 "basic 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 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 Boolean 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)

Functional paradigm

As mentioned, the VTL follows a functional programming paradigm, which treats computations as the evaluation of mathematical functions, so avoiding changing-state and mutable data in the specification of the calculation algorithm. On one side the statistical data are considered as mathematical functions (first order functions), on the other side the VTL operators are considered as functions as well (second order functions), applicable to some data in order to obtain other data.

According to the functional paradigm, the output value of a (second order) function depends only on the input arguments of the function, is calculated in its entirety and once for all by applying the function, and cannot be altered or modified once calculated (immutable) unless the input arguments change.

In fact, the VTL operators, and the expressions built using these operators, specify the algorithm for calculating the results in their entirety, once for all, and never for updating them. When some change in the operands occurs (e.g. the input data change), the VTL assumes that the results are recalculated in their entirety according to the correspondent expressions[18].

Coherently, a VTL artefact can be result of just one Transformation and cannot be updated by other Transformations, a Transformation cannot update either its own operands or the result of other Transformations and the result of a new Transformation is always a new artefact.

Transformation Consistency

The Transformation model requires that the Transformations follow some consistency rules, similar to the ones typical of the spreadsheets; in fact, there is a strict analogy between the generic models of Transformations and spreadsheets.

In this analogy, a VTL artefact corresponds to a non-empty cell of a spreadsheet, a Transformation to the formula defined in a cell (which references other cells as operands), a Result to the content of the cell in which the formula is defined[19].

The model artefacts involved in Transformations can be divided into "collected / primary" or "calculated / derived" ones. The former are original artefacts of the information system, not result of any Transformation, fed from some external source or by the users (they are analogous to the spreadsheet cells that are not calculated). The latter are produced as results of some Transformations (they are analogous to the spreadsheet cells calculated through a formula).

As already said, a Transformation calculates just one result ("derived" model artefact) and a result is calculated by just one Transformation. Both "primary" and "derived" model artefacts can be operands of any number of Transformations. An artefact cannot be operand and result of the same Transformation.

[18] At the implementation level, which is out of the scope of this document, the update operations are obviously possible

[19] The main difference between the two cases is the fact that a cell of a spreadsheet may contain only a scalar value while a VTL artefact may have also a more complex data structure, being typically a Data Set

A Transformation belongs to just one Transformation Scheme, which is analogous to a whole spreadsheet; in fact, it is a set of Transformations executed in the same run and may contain any number of Transformations, in order to produce any number of results.

Because a "derived" model artefact is produced by just one Transformation and a Transformation belongs to just one Transformation Scheme, it follows also that a "derived" model artefact is produced in the context of just one Transformation Scheme.

The operands of a Transformation may come either from the same Transformation Scheme which the Transformation belongs to or from other ones.

Within a Transformation Scheme, it can be built a graph of the Transformations by assuming that each model artefact is a node and each Transformation is a set of arcs, starting from the Operand nodes and ending in the Result node.

This graph must be a directed acyclic graph (DAG): in particular, each arc is oriented from the operand to the result; the absence of cycles makes it possible to calculate unambiguously the "derived" nodes by applying the Transformations by following the topological order of the graph.

Therefore, like in the spreadsheet, not necessarily, the Transformations are performed in the same order as they are written, because the order of execution depends on their input-output relationships (a Transformation that calculates a result, which is operand of other Transformations must be executed first).

In the analogy between VTL and a spreadsheet, the correspondences would be the following:

- VTL model artefact ←→ non-empty cell of a spreadsheet;
- VTL "collected / primary" model artefact ←→ non-empty cell of a spreadsheet whose value is fed from an external source or by the user;
- A "calculated / derived" model artefact ←→ a non-empty cell of a spreadsheet whose value is calculated by a formula;
- A VTL Transformation ←→ A spreadsheet formula assigned to a cell
- a VTL Transformation Scheme ←→ A whole spreadsheet

VTL Data types

The possible operations in VTL depend on the data types of the artefacts. For example, numbers can be multiplied but text strings cannot.

When an Operator is invoked, for each (formal) input Parameter, an actual argument (operand) is passed to the Operator, and for the output Parameter, an actual argument (result) is returned by the Operator. The data type of the argument must comply with the allowed data types of the corresponding Parameter (the allowed data types of each Parameter for each Operator are specified in the Reference Manual).

Every possible argument for a VTL Operator (with special attention to artefacts of the Information Model, e.g., Values, Sets, Data Sets) must be typed and such type deterministically inferable.

In other words, VTL Operators are strongly typed and type compliance is statically checked, i.e., violations result in compile-time errors.

Data types can be related one another, and in particular, a data type can have sub-types and super-types. For example *integer number* is a sub-type of the type *number*, and *number* is in turn a super-type of *integer number*: this means that any integer number is also a number but not the reverse, because there is no guarantee that a generic number is also an integer number. More in general, an object of a certain type is also of the respective super-types, but there is no guarantee that an object of a super-type is of any of its sub-types.

As a consequence, if a Parameter is required to be of certain type, the arguments have either this very type or any of its sub-types; arguments of its super-types are not allowed (e.g. if a Parameter is a *number*, an argument of type *integer* is accepted; vice versa, if it is an *integer*, an argument of type *number* will not be accepted).

The data types depend on two main factors: the kind of values adopted for the representation (e.g. text strings, numbers, dates, Boolean values) and the kind of structure of the data (e.g. elementary scalar values or compound values organized in more complex structures like Sets, Components, Data Sets …).

The data types for scalar values also called "scalar types" (e.g. the scalar 15 is of the scalar type "*number*", while "hello" is of the scalar type "*string*"). The scalar types are elementary because they are not defined in term of other data types. All the other data types are compound.

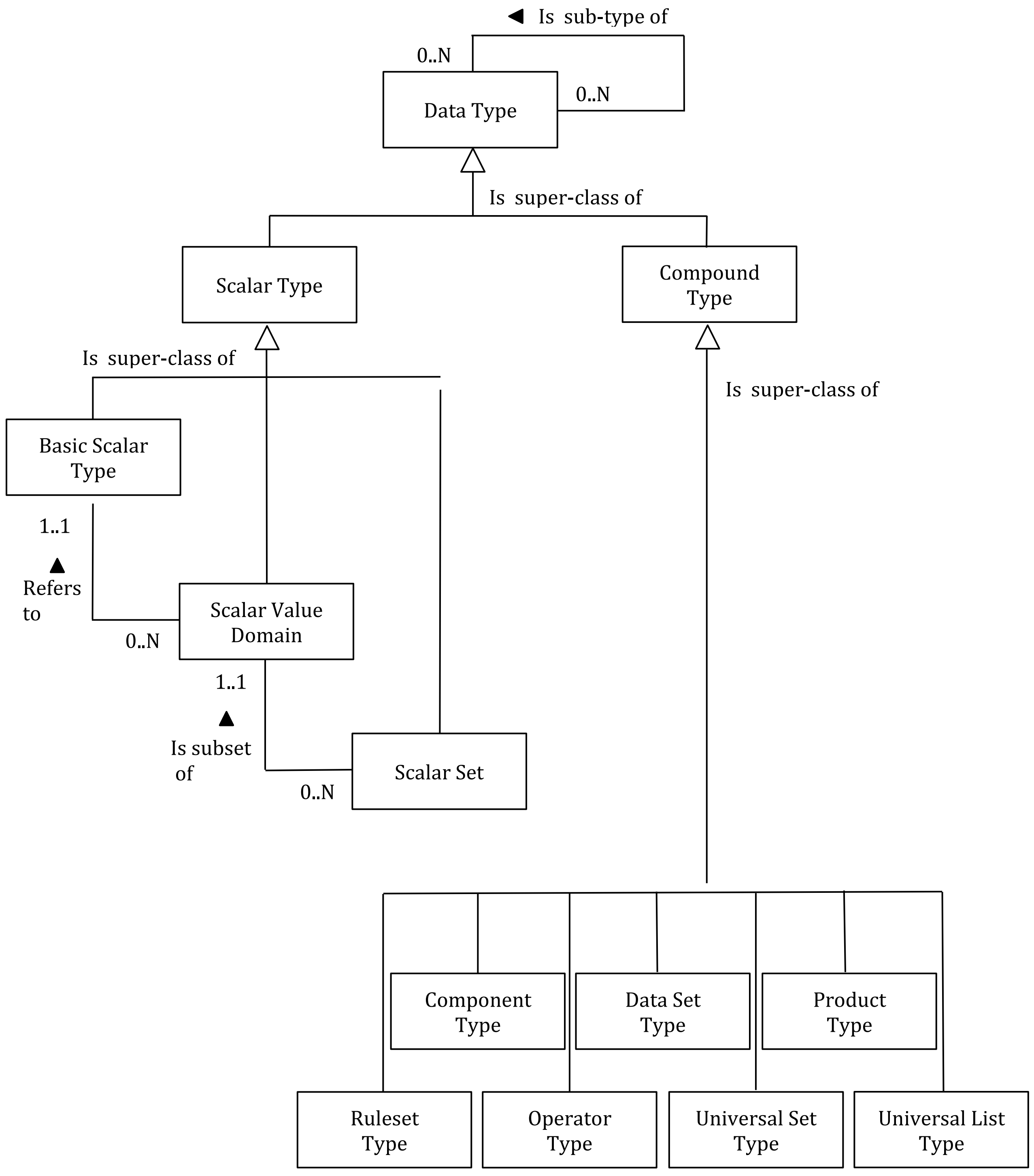
For the sake of simplicity, hereinafter the term "data type" is sometimes abbreviated to "type" and the term "scalar type" to "scalar".

A particular meta-syntax is used to specify the type of the Parameters. For example, the symbol **::** means "is of the type …" or simply "is a …" (e.g. "15 :: *number*" means "15 is of the type *number*").

In the following sections, the classes of the VTL types are illustrated, as well as some relationships between the types and the artefacts of the Information Model.

Data Types overview

Data Types model diagram



Explanation of the diagram

Data Type: this is the class of all the data types manipulated by the VTL. As already said, the actual data type of an object depends on its kind of representation and structure. As for the structure, a Data Type may be a Scalar Data Type or a Compound Data Type.

Scalar Type: the class of all the scalar types, i.e., the possible types of scalar Values. The scalar types are elementary because they are not defined in terms of other types. The Scalar Types can be Basic Scalar Types, Value Domain Scalar Types and Set Scalar Types.

Compound Data Type: the class of the compound types, i.e. the types that are defined in terms of other types.

Basic Scalar Type: the class of the scalar types which exist by default in VTL (namely, *string,number*, *integer, time, date, time_period, duration, boolean)*.

Value Domain Scalar Type: the class of the scalar types corresponding to all the scalar Values belonging to a Value Domain.

Set Scalar Type: the class of the scalar types corresponding to all the scalar Values belonging to a Set (i.e., Value Domain Subset).

Component Type: the class of the types that the Components of the Data Sets belong to, i.e. Represented Variables that assume a certain Role in the Data Set Structure.

Data Set Type: the class of the Data Sets' types, which are the more common input types of the VTL operators.

Operator Type: the class of the Operators' types, i.e., the functions that convert the types of the input operands in the type of the result.

Ruleset Type: the class of the Rulesets' types, i.e. the set of Rules defined by users that specify the behaviour of other operators (like the check and the hierarchy operators).

Product Type: the class of the types that contain Cartesian products of artefacts belonging to other generic types.

Universal Set Type: the class of the types that contain unordered collections of other artefacts that belong to another generic type and do not have repetitions.

Universal List Type: the class of the types that contain ordered collections of other artefacts that belong to another generic type and can have repetitions.

General conventions for describing the types

- The name of the type is written in lower cases and without spaces (for example the Data Set type is named "dataset").
- The double colon **::** means *"is of the type …"* or simply "*is a …*"; for example the declaration

  ```
  operand :: string
  ```

  means that the operand is a *string*.
- The vertical bar **|** indicates mutually exclusive type options, for example

  ```
  operand :: scalar | component | dataset
  ```

means that “operand” can be either *scalar*, or *component*, or *dataset.*

- The angular parenthesis **< type2 >** indicates that type 2 (included in the parenthesis) restricts the specification of the preceding type, for example:

  operand :: component <string>

  means “the operand is a component of *string* basic scalar type”.

  If the angular parenthesis are omitted, it means that the preceding type is already completely specified, for example:

  operand :: component

  means “the operand is a component without other specifications” and therefore it can be of any *scalar* type, just the same as writing operand :: component<scalar> (in fact as already said, “scalar” means “any *scalar* type”).

- The underscore **_** indicates that the preceding type appears just one time, for example:

  measure<string> _

  indicates just one Measure having the scalar type *string*; the underscore also mean that this is a non-predetermined generic element, which therefore can be any (in the example above, the string Measure can be any).

- A specific element_name in place of the underscore denotes a predetermined element of the preceding type, for example:

  measure<string not null> my_text

  means just one Measure Component, which is a not-null *string* type and whose name is “my_text”.

- The symbol **_+** means that the preceding type may appear from 1 to many times, for example:

  measure<string> _+

  means one or more generic Measures having the scalar type *string* (these Measures are not predetermined).

- The symbol **_*** means that the preceding type may appear from 0 to many times, for example:

  measure<string> _*

  means zero or more generic Measures having the scalar type *string* (these Measures are not predetermined).

Scalar Types

Basic Scalar Types

The Basic Scalar Types are the scalar types on which VTL is founded.

The VTL has various basic scalar types (namely, *string, number, integer, time, date, time_period, duration, boolean)*. The super-type of all the scalar types is the type *scalar*, which means “any scalar value”. The type *number* has the sub-type *integer* and the type *time* has two independent sub-types, namely *date* and *time_period.*

The hierarchical tree of the basic scalar types is the following:

```
Scalar
   │
   ├── String
   ├── Number
   │      └── Integer
   ├── Time
   │     │└── Date
   │     └─── Time_period
   ├── Duration
   └── Boolean
```

A scalar Value of type ***string*** is a sequence of alphanumeric characters of any length. On scalar Values of type *string*, the string operations can be allowed, like concatenation of strings, split of strings, extraction of a part of a string (substring) and so on.

A Scalar Value of type ***number*** is a rational number of any magnitude and precision, also used as approximation of a real number. On values of type *number*, the numeric operations are allowed, such as addition, subtraction, multiplication, division, power, square root and so on. The type *integer* (positive and negative integer numbers and zero) is a subtype of the type *number*.

A Scalar Value of type ***time*** denotes time intervals of any duration and expressed with any precision. According to ISO 8601 (ISO standard for the representation of dates and times), a time interval is the intervening time between two time points. This type can allow operations like shift of the time interval, change of the starting/ending times, split of the interval, concatenation of contiguous intervals and so on (not necessarily all these operations are allowed in this VTL version).

The type ***date*** is a subtype of the type *time* which denotes time points expressed at any precision, which are time intervals starting and ending in the same time point (i.e. intervals of zero duration). A value of type *date* includes all the parts needed to identify a time point at the desired precision, like the year, the month, the day, the hour, the minute and so on (for example, 2018-04-05 is the fifth of April 2018, at the precision of the day).

The type ***time_period*** is a subtype of the type *time* as well and denotes non-overlapping time intervals having a regular duration (for example the years, the quarters of years, the months, the weeks and so on). A value of the type *time_period* is composite and must include all the parts needed to identify a regular time period at the desired precision; in particular, the *time-period* type includes the explicit indication of the kind of regular period considered (e.g., “day”, “week”, “month”, “quarter” …). For example, the value 2018M04, assuming that "M" stands for "month", denotes the month n.4 of the 2018 (April 2018). Moreover, 2018Q2, assuming that "Q" stands for "quarter", denotes the second quarter of 2018. In these examples, the letters M and Q are used to denote the kind of period through its duration.

A Scalar Value of type ***duration*** denotes the length of a time interval expressed with any precision and without connection to any particular time point (for example one year, half month, one hour and fifteen minutes). According to ISO 8601, in fact, a duration is the amount of intervening time in a time interval. The *duration is* the scalar type of possible Value Domains and Components representing the period (frequency) of periodical data.

A Scalar Value of type ***boolean*** denotes a logical binary state, meaning either "true" or "false". Boolean Values allow logical operations, such as: logical conjunction (and), disjunction (or), negation (not) and so on.

All the scalar types are assumed by default to contain the conventional value "**NULL**", which means "no value", or "absence of known value" or "missing value" (in other words, the scalar types by default are "nullable"). Note that the "NULL" value, therefore, is the only value of multiple different types (i.e., all the nullable scalar types).

The scalar types have corresponding non-nullable sub-types, which can be declared by adding the suffix "*not null*" to the name of the type. For example, ***string not null*** is a string that cannot be NULL, as well as ***number not null*** is a number that cannot be NULL.

The VTL assumes that a basic scalar type has a unique internal representation and more possible external representations.

The internal representation is the reference representation of a scalar type in a VTL system, used to process the scalar values. The use of a unique internal representation allows to operate on values possibly having different external formats: the values are converted in the reference representation and then processed. Although the unique internal representation can be very important for the operation of a VTL system, not necessarily users need to know it, because it can be hidden in the VTL implementation. The VTL does not prescribe any predefined internal representation for the various scalar types, leaving different VTL systems free to using they preferred or already existing ones. Therefore, the internal representations to be used for the VTL scalar types are left to the VTL implementations.

The external representations are the ones provided by the Value Domains which refer to a certain scalar type (see also the following sections). These are also the representations used for the Values of the Components defined on such Value Domains. As obvious, the users have to know the external representations and formats, because these are used in the Data Point Values. Obviously, the VTL does not prescribe any predefined external representation, leaving different VTL systems free to using they preferred or already existing ones.

Examples of possible different choices for external representations:

- for the *strings*, various character sets can be used;
- for the *numbers*, it is possible to use the dot or the comma as decimal separator, a fixed or a floating point representation; non-decimal or non-positional numeral systems and so on;
- for the *time, date, time_period, duration* it can be used one of the formats suggested by the ISO 8601 standard or other possible personalized formats;

- the "*boolean*" type can use the values TRUE and FALSE, or 0 and 1, or YES and NO or other possible binary options.

It is assumed that a VTL system knows how to convert an external representation in the internal one and vice-versa, provided that the format of the external representation is known.

For example, the external representation of dates can be associated to the internal one provided that the parts that specify year, month and day are recognizable[20].

Value Domain Scalar Types

This is the class of the scalar Types corresponding to the scalar Values belonging to the same Value Domains (see also the section "Generic Model for Variables and Value Domains").

The super-type of all the Value Domain Scalar Types is *valuedomain*, which means any Value Domain Scalar Type. A specific Value Domain Scalar Type is identified by the name of the Value Domain.

As said in the IM section, a Value Domain is the domain of allowed Values for one or more represented variables. In other words, a Value Domain is the space in which the abstractions of a certain category of the reality (population, age, country, economic sector …) are represented.

A Value Domain refers to one of the Basic Scalar Types, which is the basic type of all the Values belonging to the Value Domain. A Value Domain provides an external representation of the corresponding Basic Scalar Type and can also restrict the possible (abstract) values of the latter. Therefore, a Value Domain defines a customized scalar type.

For example, assuming that the "population" is represented by means of numbers from zero to 100 billion, the (possible) "population" Value Domain refers to the "*integer*" basic scalar type, provides a representation for it (e.g., the number is expressed in the positional decimal number system without the decimal point) and allows only the integer numbers from zero up to 100 billion (and not all the possible numbers). Numeric operations are allowed on the population Values.

As another example, assuming that the "classes of population" are represented by means of the characters from A to C (e.g. A for population between 0 and 1 million, B for population greater that 1million until 1 billion, C for population greater than 1 billion), the "classes of population" Value Domain refers to the "string" basic scalar type and allows only the strings "A", "B" or "C". String operations are possible on these values.

As usual, even if many operations are possible from the syntactical point of view, not necessarily they make sense on the semantical plane: as usual, the evaluation of the meaningfulness of the operations remains up to the users. In fact, the same abstractions, in particular if enumerated and coded, can be represented by using different possible Value Domains, also using different scalar types. For example, the *country* can be represented through the ISO 3166-1 numeric codes (type number), or ISO alpha-2 codes (type string), or ISO alpha-3 codes (type string), or

[20] This can be achieved in many ways that depend on the data type and on the adopted internal and external representations. For example, there can exist a default correspondence (e.g., 0 means always False and 1 means always True for Boolean), or the parts of the external representation can be specified through a mask (e.g., for the dates, DD-MM-YYYY or YYYYMMDD specify the position of the digits representing year, month and day).

other coding systems. Even if numeric operations are possible on ISO 3166-1 country numeric codes, as well as string operations are possible on ISO 3166-1 alpha-2 or alpha-3 country codes, not necessarily these operations make sense.

While the Basic Scalar Types are the types on which VTL is founded and cannot be changed, all the Value Domains are user defined, therefore their names and their contents can be assigned by the users.

Some VTL Operators assume that a VTL system have certain kinds of Value Domains which are needed to perform the correspondent operations[21]. In the VTL manuals. Definite names and representations are assigned to such Value Domains for explanatory purposes; however, these names and representations are not mandatory and can be personalised if needed. If VTL rules are exchanged between different VTL systems, the partners of the exchange must be aware of the names and representations adopted by the counterparties.

Set Scalar Types

This is the class of the scalar types corresponding to the scalar Values belonging to the same Sets (see also the section "Generic Model for Variables and Value Domains").

The super-type of all the Set Scalar Types is set, which means any Set Scalar Type. A specific Set Scalar Type is identified by the name of the Set.

A Set is a (proper or improper) subset of the Values belonging to a Value Domain (the Set of all the values of the Value Domain is an improper subset of it). A scalar Set inherits from its Value Domain the Basic Scalar Type and the representation and can restrict the possible Values of its Value Domain (as a matter of fact, except the Set which contains all the values of its Value Domain and can also be assumed to exist by default, the other Sets are defined just to restrict the Values of the Value Domain).

External representations and literals used in the VTL Manuals

The Values of the scalar types, when written directly in the VTL definitions or expressions, are called *literals*.

The literals are written according to the external representations adopted by the specific VTL systems for the VTL basic data types (i.e., the representations of their Value Domains). As already said, the VTL does not prescribe any particular external representation.

In these VTL manuals, anyway, there is the need to write literals of the various data types in order to explain the behaviour of the VTL operators and give proper examples. The representation of these literals are not intended to be mandatory and are not part of the VTL standard specifications, these are only the representations used in the VTL manuals for explanatory purposes and many other representations are possible and legal.

The representations adopted in these manuals are described below.

The string values are written according the Unicode and ISO/IEC 10646 standards.

[21] For example, at least one default Value Domain should exist for each basic scalar type, the Value Domains needed to represent the results of the checks should exist, and so on.

The ***number*** values use the positional numeral system in base 10.

- A fixed-point *number begins* with the integer part, which is a sequence of numeric characters from 0 to 9 (at least one digit) optionally prefixed by plus or minus for the sign (no symbol means plus), a dot is always present in the end of the integer part and separates the (possible) fractional part, which is another sequence of numeric characters.
- A floating point *number*, has a mantissa written like a fixed-point number, followed by the capital letter E (for "Exponent") and by the exponent, written like a fixed-point integer;

For example:

- Fixed point *numbers*: 123.4567 +123.45 -8.901 0.123 -0.123
- Floating point *numbers*: 1.23E2 +123.E-2 -0.89E1 0.123E0

The ***integer*** values are represented like the *number* values with the following differences:

- A fixed-point *integer* is written like a fixed-point *number* but without the dot and the fractional part.
- A floating point *integer* is written like a floating-point *number* but cannot have a negative mantissa.

For example:

- Fixed point integers: 123 +123 -123
- Floating point integers: 123E0 1E3

The ***time*** values are conventionally represented through the initial and final Gregorian dates of the time interval separated by a slash. The accuracy is reduced at the level of the day (therefore omitting the time units shorter than the day like hours, minutes, seconds, decimals of second). The following format is used (this is one of the possible options of the ISO 8601 standard):

YYYY-MM-DD/YYYY-MM-DD

where *YYYY* indicates 4 digits for the year, *MM* indicates two digits for the month, *DD* indicates two digits for the day. For example:

2000-01-01/2000-12-31	the whole year 2000
2000-01-01/2009-12-31	the first decade of the XXI century

The ***date*** values are conventionally represented through one Gregorian date. The accuracy is reduced at the level of the day (therefore omitting the time units shorter than the day like hours, minutes, seconds, decimals of second). The following format is used (this is one of the possible options of the ISO 8601 standard):

YYYY-MM-DD

The meaning of the symbols is the same as above. For example:

2000-12-31	the 31st December of the year 2000
2010-01-01	the first of January of the year 2010

The ***time_period*** values are represented for sake of simplicity with accuracy equal to the day or less (week, month …) and a periodicity not higher than the year. In the VTL manuals, the following format is used (this is a personalized format not compliant with the ISO 8601 standard):

YYYYPppp

where *YYYY* are 4 digits for the year, *P* is one character for specifying which is the duration of the regular period (e.g. D for day, W for week, M for month, Q for quarter, S for semester, Y for the whole year, see the codes of the *duration* data type below), ppp denotes from zero two three digits which contain the progressive number of the period in the year. For example:

2000M12	the month of December of the year 2000
2010Q1	the first quarter of the year 2010
2020Y	the whole year 2010

The ***duration*** values in these manuals are conventionally restricted to very few predefined durations that are codified through just one character as follows:

Code	*Duration*
D	Day
W	Week
M	Month
Q	Quarter
S	Semester
A	Year (Annual)

This is a very simple format not compliant with the ISO 8601 standard, which allows representing durations in a much more complete, even if more complex, way. As mentioned, the real VTL systems may adopt any other external representation.

The ***boolean*** values used in the VTL manuals are *TRUE* and *FALSE* (without quotes).

When a *literal* is written in a VTL expression, its basic scalar type is not explicitly declared and therefore is unknown.

For ensuring the correctness of the VTL operations, it is important to assess the scalar type of the literals when the expression is parsed. For this purpose, there is the need for a mechanism for the disambiguation of the literals types, because often the same literal might itself belong to many types, for example:

- the word "true" may be interpreted as a string or a boolean,
- the symbol "0" may be interpreted as a string, a number or a boolean,
- the word "20171231" may be interpreted as a string, a number or a date.

The VTL does not prescribe any predefined mechanism for the disambiguation of the scalar types of the literals, leaving different VTL systems free to using they preferred or already existing ones. The disambiguation mechanism, in fact, may depend also on the conventions adopted for the external representation of the scalar types in the VTL systems, which can be various.

In these VTL manuals, anyway, there is the need to use a disambiguation mechanism in order to explain the behaviour of the VTL operators and give proper examples. This mechanism, therefore, is not intended to be mandatory and, strictly speaking, is not part of the VTL standard.

If VTL rules are exchanged between different VTL systems, the partners of the exchange must be aware of the external representations and the disambiguation mechanisms adopted by the counterparties.

The disambiguation mechanism adopted in these VTL manuals is the following:

- The string literals are written between double quotes, for example the literal "123456" is a string, even if its characters are all numeric, as well as "I am a string! ".
- The numeric literals are assumed to have some pre-definite patterns, which are the numeric patterns used for the external representation of the numbers described above.
  A literal having one of these patterns is assumed to be a number.
- The boolean literals are assumed to be the values TRUE and FALSE (capital letters without quotes).

In these manuals, it is also assumed that the types *time*, *date, time_period* and *duration* do not directly support literals. Literal values of such types can be anyway built from literals of other types (for example they can be written as strings) and converted in the desired type by the cast operator (type conversion). In some cases, the conversion can be made automatically, (i.e., without the explicit invocation of the cast operator – see the Reference Manual for more details).

As mentioned, the VTL implementations may personalize the representation of the literals and the disambiguation mechanism of the basic scalar types as desired, provided that the latter work properly and no ambiguities in understanding the type of the literals arise. For example, in some cases the type of a literal can also be deduced from the context in which it appears. As already pointed out, the possible personalised mechanism should be communicated to the counterparties if the VTL rules are exchanged.

Conventions for describing the scalar types

- The keywords which identify the basic scalar types are the following: scalar, string, number, integer, time, date, time_period, duration, boolean.
- By default, the basic scalar types are considered as nullable, i.e., allowing NULL values.
- The keyword **not null** following the type (and the "literal" keyword if present), means that the scalar type does not allow the NULL value, for example:

  operand :: string literal not null

  means that the operand is a literal of *string* scalar type and cannot be NULL; if not null is omitted the NULL value is meant to be allowed.
- An **expression within square brackets** following the previous keywords, means that the preceding scalar type is restricted by the expression. This is a VTL *boolean* expression[22]

[22] I.e., an expression whose result is *boolean*

(whose result can be TRUE or FALSE) which specifies a membership criterion, that is a condition that discriminates the values which belong to the restriction (sub-type) from the others (the value is assumed to belong to the sub-type only if the expression evaluates to TRUE). The keyword "value" stands for the generic value of the preceding scalar type and is used in the expression to formulate the restrictive condition. For example:

```
integer [ value <= 6 ]
```

is a sub-type of *integer* which contains only the integers lesser than or equal to 6.

The following examples show some particular cases:

- The generic expression **[between (value, x, y)]**[23] restricts a scalar type by indicating a closed interval of possible values going from the value x to the value y, for example

  ```
  integer [between (value, 1, 100 ) ]
  ```

  is the sub-type which contains the integers between 1 and 100.
- The generic expression **[(value > x) and (value < y)]** restricts a scalar type by indicating an open interval of possible values going from the value x to the value y, for example

  ```
  integer [ (value > 1) and (value < 100) ]
  ```

  means integer greater than 1 and lesser than 100 (i.e., between 2 and 99).
- By using **>=** or **<=** in the expressions above, the intervals can be declared as open on one side and closed on the other side, for example

  ```
  integer [ (value >= 1) and (value < 100) ]
  ```

  means integer greater than or equal to one and lesser than 100.
- The generic expressions **[value >= x]** or **[value > x]** or **[value <= y]** or **[value < y]** restrict a scalar type by indicating an interval having one side unbounded, for example

  ```
  integer [ value >= 1 ]
  ```

  means integer equal to or greater than 1, while "integer[value < 100]" means an integer lesser than 100.
- The generic expression **[value in { v_1, ... , v_N }]**[24] restricts a scalar type by specifying explicitly a set of possible values, for example

  ```
  integer {1, 2, 3, 4, 5, 6}
  ```

  means an integer which can assume only the integer values from 1 to 6. The same result is obtained by specifying [value in set_name], where in is the "Element of"

[23] "between (x, y, z)" is the VTL operator which returns TRUE if x is comprised between y and z

[24] "in" is the VTL operator which returns TRUE if an element (in this case the value) belongs to a Set; the symbol {… , … , … } denotes a set defined as the list of its elements (separated by commas)

VTL operator and set_name is the name of an existing Set (Value Domain Subset) of the VTL IM.

- By using in the expression the operator length[25] it is possible to restrict a scalar type by specifying the possible number of digits that the values can have, for example

  integer [between (length (value), 1, 10)]

  means an integer having a length from one to 10 digits.

As obvious, other kinds of conditions are possible by using other VTL operators and more conditions can be combined in the restricting expression by using the VTL boolean operators (and, or, not ...)

- Like in the general case, a restricted scalar type is considered by default as including the NULL value. If the NULL value must be excluded, the type specification must be followed by the symbol **not null**; for example

  integer [between (length (value), 1, 10)] not null

  means a not-null integer having from one to 10 digits.

Compound Data Types

The Compound data types are the types defined in terms of more elementary types.

The compound data types are relevant to artefacts like Components, Data Sets and to other compound structures. For example, the type Component is defined in terms of the scalar type of its values, besides some characteristics of the Component itself (for example the role it assumes in the Data Set, namely Identifier, Measure or Attribute). Similarly, the type of a Data Set (i.e. of a mathematical function) is defined in terms of the types of its Components.

The compound Data Types are described in the following sections.

Component Types

This is the class of the Component types, i.e. of the Components of the Data Structures (for example the country of residence used as an Identifier, the resident population used as a Measure ...).

A Component is essentially a Variable (i.e. an unknown scalar Value with a certain meaning, e.g. the resident population) which takes Values in a Value Domain or a Set and plays a definite role in a data structure (i.e. Identifier, Measure or Attribute). A Component inherits the scalar type (e.g. *number*) from the relevant Value Domain.

The main sub-types of the Component Type depend on the role of the Component in the data structure and are the *identifier*, *measure* and *attribute* types (if the automatic propagation of the Attributes is supported, another sub-type is the *viral attribute*). These types reflect the

[25] "length" is the VTL Operator that returns the length of a string (in the example, the *integer* operand of the length operator is automatically cast to a string and its length is determined)

fact that the VTL behaves differently on Components of different roles. Their common super-type is *component*, which means "a Component having any role".

Moreover, a Component type can be restricted by an associated scalar type (e.g. *number, string …)*, therefore the complete specification of a Component type takes the form

```
role_type < scalar_type >
```

where the scalar type included in angular parenthesis, restricts the specification of the preceding type (the role type); omitted angular parenthesis mean "any scalar type", which is the same as writing <*scalar*>. Examples of Component types are the following:

component (or component<scalar>)	any Component
○ component<number>	any Component of scalar type *number*
○ identifier (or identifier<scalar>)	any Identifier
▪ identifier<time not null>	Identifier of scalar type *time not null*
○ measure (or measure<scalar>)	any Measure
▪ measure<boolean>	Measure of scalar type *Boolean*
○ attribute (or attribute<scalar>)	any Attribute
▪ attribute<string>	Attribute of scalar type *string*

In the list above, the more indented types are sub-types of the less indented ones.

According to the functional paradigm, the Identifiers cannot contain NULL values, therefore the scalar type of the Identifiers Components must be "not null".

In summary, the following conventions are used for describing Component types.

- As already said, the more general type is "**component**" which indicates any component, for example:

  ```
  operand :: component
  ```

  means that "operand" may be any component.

- The main sub-types of the *component* type correspond to the roles that the Component may assume in the Data Set, i.e., **identifier**, **measure**, **attribute**; for example:

  ```
  operand :: measure
  ```

  means that the operand must be a Measure.

  The additional role **viral attribute** exists if the automatic propagation of the Attributes is supported[26]. The type *viral_attribute* is a sub-type of *attribute*.

- By default, a Component can be either specified directly through its name or indirectly through a sub-expression that calculates it.

- The optional keyword **name** following the type keyword means that a component name must be specified and that the component cannot be obtained through a sub-expression; For example:

[26] See the section "Behaviour for Attribute Components"

operand :: measure name <string>

means that the name of a *string* Measure must be specified and not a string sub-expression[27]. If the name keyword is omitted the sub-expression is allowed.

- The symbol **< scalar type >** means that the preceding type is restricted to the scalar type specified within the angular brackets", for example:

  operand :: component < string >

  means that the operand is a Component having any role and belonging to the *string* scalar type; if the restriction is not specified, then the scalar type can be any (for example operand:: attribute means that the operand is an Attribute of any scalar type).

- In turn, the scalar type of a Component can be restricted; for example:

  operand:: measure < integer [value between 1 and 100] not null >

  means that the operand can be a not-null integer Measure whose values are comprised between 1 and 100.

Data Set Types

This is the class of the Data Sets types. The Data Sets are the main kind of artefacts manipulated by the VTL and their types depend on the types of their Components.

The super-type of all the Data Set types is *dataset*, which means "any dataset" (according to the definition of Data Set given in the IM, as obvious).

A sub-type of dataset is the Data Sets of time series, which fulfils the following restrictive conditions:

- The Data Set structure must contain one Identifier Component that acts as the reference time, which must belong to one of the basic scalar types *time*, *date* or *time_period*.
- The possible values of the reference time Identifier Component must be regularly spaced
  - For the type *time*, the time intervals must start (or end) at a regular periodicity and have the same duration
  - For the type *date*, the time values must have a regular periodicity
  - For the type *time_period* there are no additional conditions to fulfil, because the *time_period* values comprise by construction the indication of the period and therefore are regularly spaced by default
- It is assumed that it exists the information about which is Identifier Components that acts as the reference time and about which is the period (frequency) of the time series and that such information is represented in some way in the VTL system. The VTL does not prescribe any predefined representation, leaving different VTL systems free to using they preferred or already existing ones. It is assumed that the VTL operators acting on time series know which is the reference time Identifier and the period of the time series and use this information to perform correct operations.

[27] I.e., a sub-expressions whose result is *string*

Usually, the information about which is the reference time is included in the data structure definition of the Data Sets or in the definition of the Data Set Components.
Some commonly used representations of the periodicity are the following:

- For the types *time* and *date*, the period is often represented through an additional Component of the Data Set (of any possible role) or an additional metadata relevant to the whole Data Set or some parts of it. This Component (or other metadata) is of the "duration" type and is often called "frequency".
- For the type *time_period*, the periodicity is embedded in the *time_period* values.

In any case, if some periodical data exist in the system, it is assumed that a Value Domain representing the possible periods exists and refers to the *duration* scalar type.

Within a Data Set of Time Series, a single Time Series is the set of Data Points that have the same values for all the Identifier Components except the reference time[28]. A Data Set of time series can also contain more time series relevant to the same phenomenon but having different periodicities, provided that one or more Identifiers (other than the reference time) distinguish the Time Series having different periodicity.

The Data Sets of time series are the possible operands of the time series operators (they are described in the Reference Manual).

More specific Data Set Types can be defined by constraining the *dataset* type, for example by specifying the number and the type of the possible Components in the different roles (Identifiers, Measures and Attributes), and even their names if needed. Therefore the general syntax is:

dataset { type_constraint } or dataset_ts { type_constraint }

where the type_constraint may assume many different forms which are described in detail in the following section. Examples of Data Set types are the following:

dataset	Any Data Set (according to the IM)
dataset { measure <number> _* }	A Data Set having one or more Measures of type *number*, without constraints on Identifiers and Attributes
dataset { measure <boolean> _ , attribute<string> _* }	A Data Set having one *boolean* Measure, one or more *string* Attributes and no constraints on Identifiers

In summary, the following conventions are used for describing Data Set types.

- The more general type is "**dataset**" which means any possible Data Set of the VTL IM (in other words, a Data Set having any possible components allowed by the IM integrity rules)
- By default, a Data Set can be either specified directly through its name or indirectly through a sub-expression which calculates it.

[28] Therefore each combination of values of the Identifier Components except the reference time identifies a Time Series

- The optional keyword **name** following **dataset** means that a Data Set name must be specified and that the Data Set cannot be obtained through a sub-expression; for example:

  operand :: dataset name

  means that a Data Set name must be specified and not a sub-expression. If the name keyword is omitted the sub-expression is allowed.
- The symbol **dataset { type_constraint }** indicates that the *type_constraint* included in curly parenthesis restricts the specification of the preceding *dataset type* without giving a complete type specification, but indicating only the constraints in respect to the general structure of the artefact of the Information Model corresponding to such *type*. For example, given that the generic structure of a Data Set in the IM may have any number of Identifiers, Measures and Attributes and that these Components may be of any scalar type, the declaration:

  operand :: dataset { measure<string> _ }

  means that the operand is of type Data Set having any number of Identifiers (like in the IM), just one Measure of string type (as declared in the type declaration) and any number of Attributes (like in the IM).
- Some or all the Data Set Components can also be predetermined. For example writing:

  operand:: dataset { identifier<st_Id$_1$> Id$_1$, …, identifier<st_Id$_N$> Id$_N$, measure<st_Me$_1$> Me$_1$, … , measure<st_Me$_L$> Me$_L$, attribute<st_At$_1$> At$_1$, … , attribute<st_At$_K$> At$_K$ }

  means that the operand is of Data Set type and has the identifier, measure and attribute types and names specified within the curly brackets (in the example, <st_Id$_1$> stands for the scalar type of the Component named Id$_1$ and so on). This is the example of an extremely specific Data Set type in which all the component types and names are fixed in advance.
- If a certain role (i.e. identifier, measure, attribute) is not specified, it means that there are no restrictions on it, for example:

  operand :: dataset { me<st_Me$_1$ > Me$_1$, … , me<st_Me$_L$ > Me$_L$ }

  means that the operand is of Data Set type and has the measure types and names specified within the curly brackets, while the Identifier and Attribute components have no restrictions and therefore can be any.

Product Types

This is the class of the Cartesian products of other types; a product type is written in the form $t_1 * t_2 * ... * t_n$ where t_i $(1 < i <= n)$ is another arbitrary type; the elements of a Product type are n-tuples whose i_{th} element belongs to the type t_i. For instance, the product type:

*string * integer * boolean*

includes elements like[29] ("PfgTj", 7, true), ("kj-o", 80, false), ("", 4, false) but does not include for example ("qwe", 2017-12-31, true), ("kj-o", 80, 92).

The superclass is *product*, which means any product type.

Product types can be used in practice for several reasons. They allow:

i. the natural expression of exclusion or inclusion criteria (i.e., constraints) over values of two or more dataset components;
ii. the definition of the domain of the Operators in term of types of their Parameters
iii. the definition of more complex data types.

Operator Types

This is the class of the Operators' types, i.e., the higher-levels functions that allow transformations from the type t_1 (the type of the input Parameters), to the type t_2 (the type of the output Parameter). An Operator Type is written in the form 't_1 -> t_2', where t_1 and t_2 are arbitrary types. For example, the type of the following operator says that it takes as input two integer Parameters and returns a number:

Op$_1$ :: *integer * integer -> number*

The superclass is *operator*, which means any operator type.

Ruleset Types

The class of the Ruleset types, i.e. the set of Rules that are used by some operators like "check_hierarchy", "check_datapoint", "hierarchy", "transcode". The general syntax for specifying a Ruleset type is main_type_of_ruleset {type_constraint}.

The main Rulesets types are the *datapoint* and the *hierarchical* Rulesets. Their super-type is *ruleset* that means "any Ruleset". Moreover, Rulesets can be defined either on Value domains or on Variables, therefore the main_type_of_rulesets are:

ruleset

- *datapoint*
  - *datapoint_on_value domains*
  - *datapoint_on_variables*
- *hierarchical*
  - *hierarchical_on_value_domains*
  - *hierarchical_on_variables*

In the list above, the more indented types are sub-types of the less indented ones.

The type_constraint is optional and may assume many different forms that depends on the main_type_of_ruleset. If the type_constraint is present, the main_type_of_ruleset must specify if the ruleset is defined on Value Domains or Variables (i.e., it must be one of the more indented types above).

[29] In the VTL syntax the symbol ( ) allows to define a tuple in-line by enumeration of its elements

A datapoint Ruleset is defined on a Cartesian product of Value Domains or Variables, therefore the type_constraint can contain such a list. Examples of constrained datapoint types are:

> datapoint on value domains {(geo_area * sector * time_period * numeric_value)}
>
> datapoint on variables {(ref_date * import_currency * import_country)}
>
> datapoint on value *domains* {date * _+}

The last one is the type of the Data Point Rulesets that are defined on the "date" Value Domain and on one to many other Value Domains ("_+" means "one or more").

A hierarchical Ruleset is defined on one Value Domain or Variable and can contain conditions referred to other Value Domains or Variables; therefore, the type_constraint for hierarchical Rulesets can take one of the following forms:

> {value_domain * (conditioningValueDomain1 * … * conditioningValueDomainN)}
>
> {variable * (conditioningVariable1 * … * conditioningVariableN)}.

Examples of *hierarchical* types are:

> hierarchical on value domains {geo_area * (time_period)}
>
> hierarchical on variables { currency * (date * country) }
>
> hierarchical on value domains { _ }
>
> hierarchical on value domains { _ * (reference_date)}

The last one is the type of the Data Point Rulesets that are defined on the "date" Value Domain and on one to many other Value Domains (in the meta-syntax "_+" means "one or more").

The last one is the type of the Hierarchical Rulesets that are defined on any Value Domain and are not conditioned by other Value Domains.

Universal Set Types

The Universal Sets are unordered collections of other objects that belong to the same type *t* and do not have repetitions (each object can belong to a Set just once). The Universal Sets are denoted as *set<t>*, where *t* is another arbitrary type. If < t > is not specified it means any universal set type.

Possible examples are the Sets of product types. For instance the Universal Set Type:

> *set < string * integer * boolean >*

includes the sets[30]:

> { ("PfgTj", 7, true), ("kj-o", 80, false), ("", 4, false) }
>
> { ("duo9", 67, true), ("io/p", 540, true) }

But does not includes the sets:

> { ("PfgTj", 7, true), 80, ("", 4, false) }   in fact 80 is not a *product type*

[30] In the VTL syntax, the symbol {…} denotes a set defined as the list of its elements (separated by commas).

{ ("duo9", 67, true), (50, true) }	in fact (50, true) is not the right *product type*
{ ("", 4, false), ("F", 8, true), ("", 4, false) }	in fact ("", 4, false) is repeated

Universal List Types

The Universal Lists are ordered collections of other objects that belong to the same type *t* and can have repetitions (an object can appear in a list more than once). The Universal Lists are denoted as *list<t>*, where *t* is an arbitrary type. If < t > is not specified it means any universal list type.

For instance the following Universal List type:

list < component>

includes elements like[31] [reference date, import, export] but does not include elements like [dataset1, country, sector] and [import, "text"] because dataset1 and "text" are not Components.

[31] In the VTL syntax, the symbol [ ] allows to define a List in-line by enumeration of its elements.

VTL Transformations

This section describes the key concepts, assumptions and characteristics of the VTL which are needed to a VTL user to define Transformations. As mentioned in the section about the general characteristics above, the language is oriented to users without deep information technology (IT) skills, who should be able to define calculations and validations independently, without the intervention of IT personnel. Therefore, the VTL has been designed to make the definition of the Transformations as intuitive as possible and to reduce the chances of errors.

As already said, a Transformation consists of a statement that assigns the outcome of the evaluation of an Expression to an Artefact of the Information Model. Then, Transformations are made of the following components:

- A right side, which contains the expression to be evaluated, whose inputs are the operands of the Transformation
- An assignment operator
- A left side, which specifies the Artefact which the outcome of the expression is assigned to (this is the result of the Transformation)

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

- $D_1 := D_2$
- $D_3 := D_4 + D_5$

Assuming that E is the expression, R is the result and $IO_{i\ (i=1,\dots\ n)}$ the input Operands, the mathematical form of a Transformation based on E can be written as follows:

$$R := E\ (IO_1, IO_2, \dots , IO_n)$$

The expression uses any number of VTL operators in combination to specify a compound operation. Because all the VTL operators are functional, the whole expression is functional too.

Transformations are properly chained for their execution; in fact, the result R_i of a Transformation T_i can be referenced as operand of another Transformation T_j. In this case, the former Transformation is evaluated first in order to provide the input for the latter. To enforce the consistency of the results, the cycles are not allowed, therefore in the case above the result R_j of the Transformation T_j cannot be operand of the Transformation T_i and cannot contribute to the calculation of any operand of T_i, even indirectly through a chain of other Transformations.

The order in which the user defines the Transformations may be important for a better understanding but cannot override the order of execution determined according their input-output relationships.

For the rules for the Transformation consistency, see also the section “Generic Model for Transformation” above.

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

The Expression

A VTL expression constitutes the right side of a Transformation. It takes one or more input operands and returns one output artefact.

An expression is the invocation of one or more operators in combination, in which the result of an operator is passed as input parameter to another operator, and so on, in a tree structure. The root of the tree structure is last operator to be applied and gives the final result.

For example, for the expression *a + b - c* the result of the addition *a+b* is passed to the following subtraction, which gives the final result.

An expression is built from the following ingredients:

- **Operators**, which specify the operation to be performed (e.g. +, - and so on). As mentioned, the standard VTL operators are described in detail in the Reference Manual, moreover the VTL allows defining and then invoking "user defined operators" (see the Reference Manual, the VTL-DL for the definition and the VTL-ML for the invocation). Each operator envisages a certain number of input parameters of definite data types and produces an outcome having a definite data type (the types parameter are described in detail in the Reference Manual for each operator).
- **Operands**, which are the actual arguments passed to the invoked Operators, for example writing $D_1 + D_2$ the Operator "+" is invoked and the Operands D_1 and D_2 are passed to it. The Operands can be:
  - **Named artefacts**, which are VTL artefacts specified through their names. Their actual values are obtained either referring to an external persistent source (persistent artefacts) or as result of previous Transformations (non-persistent artefacts) of the same Transformation Scheme; they are identified by means of a symbolic name (e.g. in $D_1 + D_2$ the Operands D_1 and D_2 are identified by the names D_1 and D_2). Examples of identified artefacts are the Data Sets (like D_1 and D_2 above) and the Data Set Components (like $D_1\#C_1$, $D_1\#C_2$, $D_1\#C_3$, where # means that C_j is a Component of the Data Set D_i).
  - **Literals**, which are VTL artefacts whose actual values are directly written in the expression; for example, in the invocation $D_1 + 7$ the second operand (7) is a literal, in this case a scalar literal. Also other kind of artefacts can be written in the expressions, for example the curly brackets denote the value of a Set (for example *{1, 2, 3, 4, 5, 6}* is the set of the integers from 1 to 6) and the square brackets denote a list (for example *[7, 5, 3, 6, 3]* is a list of numbers).
- **Parenthesis**, which specify the order of evaluation of the operators; for example in the expression $D_1 * (D_2 + D_3)$ first the sum $D_2 + D_3$ is evaluated and then their product for D_1. In case the parenthesis are not used, the default order of evaluation (described in the Reference Manual) is applied (in the example, first the product and then the sum).

An expression implies different steps of calculation, for example the expression:

$$R := O_1 + O_2 \ / \ (O_3 - O_4 \ / \ O_5)$$

can be calculated in the following steps:

I. (O_4 / O_5)

II. $(O_3 - I)$
III. (O_2 / II)
IV. $(O_1 + III)$

The intermediate and final results (I, II, III, IV) of the expression are assumed to be non-persistent (temporary). The persistency of the result Data Set *R* is controlled by the assignment operator, as described below.

An intermediate result within the expression can be only the input of other operators in the same expression.

In general, unless differently specified in the Reference Manual, in the invocation of an operator any operand can be the result of a sub-expression that calculates it. For example, taking the exponentiation whose syntax is

power(base, exponent),

the invocation $power(D_1 + D_2 , 2)$ is allowed and means that first $D_1 + D_2$ is calculated and then the result is squared. As usual, the data type of the calculated operand must comply with the allowed data types of the corresponding Parameter (in the example above, $D_1 + D_2$ must have a numeric data type, otherwise it cannot be squared).

The nesting capabilities allow writing from very simple to very complex expressions. The complexity of the expressions can be managed by the users by splitting or merging transformations. For example, taking again the example above, the following two options would give the same result:

Option 1:

$D_r := power(D_1 + D_2 , 2)$

Option 2:

$D_3 := D_1 + D_2$

$D_r := power(D_3 , 2)$

In both cases, in fact, first $D_1 + D_2$ is evaluated and then the *power* operator is applied to obtain D_r.

In general, it is possible either to have simpler expressions by splitting and chaining Transformations or to have a minor number of Transformations by writing more complex expressions.

The Assignment

The assignment of an expression to an artefact is done through an assignment operator. The VTL has two assignment operators, the persistent and the non-persistent assignment:

<-	persistent assignment
:=	non-persistent assignment

The former assigns the outcome of the expression on the left side to a persistent artefact, the latter to a non-persistent one; therefore, the choice of the assignment operator allows controlling the persistency of the artefact that is result of the Transformation.

The only artefact that can be made persistent is the result (the left side artefact). In fact, as already mentioned, the intermediate and final results of the right side expression are always considered as non-persistent.

For example, taking again the example of Transformation above:

D_r := *power*($D_1 + D_2$, 2)

The result D_r can be declared as persistent by writing:

D_r <- *power*($D_1 + D_2$, 2)

Instead, to make persistent also the intermediate result of $D_1 + D_2$ it is necessary to split the Transformation like in the option 2 above:

D_3 <- $D_1 + D_2$

D_r <- *power*(D_3 , 2)

The persistent assignment operator is also called *Put*, because it is used to specify that a result must be put in a persistent store. The *Put* has two parameters, the first is the final result of the expression on the right side that has to be made persistent, the second is the reference to the persistent Data Set which will contain such a result.

The Result

The left side artefact, i.e. the result of the Transformation, is always a named Data Set (i.e. a Data Set identified by means of a symbolic name like explained in the previous section).

The data type and structure of the left side Data Set coincide with the data type and structure of the outcome of the expression, which must be a Data Set as well.

Almost all VTL operators act on Data Sets. Many VTL operators can act also on Data Set Components to produce other Data Set Components, however even in this case the outcome of the expression is a new Data Set that contains the calculated Components.

An expression can result also in scalar Value; because many VTL operators can act on scalar Values to obtain other scalar Values, furthermore some particular operations on Data Sets can eliminate Identifiers, Measures and Attributes and obtain scalar Values (see the Reference Manual). The result of such expressions is considered as a named Data Set that does not have Components (Identifiers, Measures and Attributes) and therefore contains just one scalar Value. The Data Sets without Components can be manipulated and possibly stored like any other Data Set. Because the VTL notion of Data Set is logical and not physical, more Data Set without Components can be stored in the same physical Data Set if appropriate.

The current VTL version does not include operators that produce other output data types, for example, there are not operators that manipulate Sets (however this is a possible future development).

In fact, the Data Set at the moment is the only type of Artefact that can be produced and stored permanently through a command of the language.

The names

The artefact names

The names are the labels that identify the "named" artefacts that are operands or result of the transformations.

For ensuring the correctness of the VTL operations, it is important to distinguish the names from the scalar literals when the expression is parsed. For this purpose, the disambiguation mechanism that distinguishes the types of the scalar literals must also be able of distinguishing names and scalar literals.

As already mentioned in the section about the scalar literals, the VTL does not prescribe any predefined disambiguation mechanism, leaving different VTL systems free to using they preferred or already existing ones. In these VTL manuals, anyway, there is the need to use some disambiguation mechanisms in order to explain the behaviour of the VTL operators and give proper examples. These mechanisms are not intended to be mandatory and therefore, strictly speaking, they are not part of the VTL standard specifications. If no drawbacks exist, however, their adoption is encouraged to foster the convergence between possible different practices. If VTL rules are exchanged, the disambiguation mechanisms should be communicated to the counterparties, at least if they are different from the one suggested hereinafter.

The general rules for the names are given below. As said above, these rules can be personalized (for example restricted) in some implementations (e.g. a particular implementation can require that a name starts with a letter).

The names are strings of characters no more than 128 characters long and are classified in regular and non-regular names.

The **regular names**:

- can contain alphabetic and numeric characters and the special characters underscore (_) and dot (.) ,
- must begin with an alphanumeric character and not with a special character
- must contain at least one alphabetic character
- cannot be a VTL reserved word

Examples or regular names are *abcdef*, *1ab_cde*, *a.b.c_d_e*, *1234_5*.

The regular names are:

- written in the Transformations / Expressions without delimiters
- case insensitive

The non-regular names:

- can contain alphanumeric characters and, in addition to the underscore and the dot, any other Unicode character

- can contain blanks
- can begin with special characters
- can contain only numeric characters
- can be equal to the VTL reserved words

The non-regular names are:

- written in the Transformations / Expressions with single quotes as delimiters
- case sensitive

Examples of non-regular names, which therefore are enclosed in single quotes, are *'_abcdef'*, *'1ab-cde'*, *'12345'*, *'power'* (the first begins with a special character, the second contains the "-" character that is not allowed, the third contains only numeric characters, the fourth coincides to a VTL reserved word (the name of the exponentiation operator). These names would not be recognized by VTL if not enclosed between single quotes.

The **VTL reserved words** (and symbols) are:

- the keywords of the VTL-ML and VTL-DL operators and of their parameters (e.g. <, = , # , inner_join, as, using, filter, apply, rename, to, + , - , power, and, or, not, group by, group except, group all, having ...)
- the names of the classes of VTL artefacts of the VTL-IM (e.g., value, value domain, value domain subset, set, variable, component, data set, data structure, operator, operand parameter, transformation ...)
- additional keywords for possible future use like get, put, join, map, mapping, merge, transcode and the names of commonly used mathematical and statistical functions.

The environment name

In order to ensure non-ambiguous definitions and operations, the names of the artefacts must be unique, meaning that an identifier cannot be assigned to more than one artefact.

In practice, the unicity of the names is ensured in a certain environment, that can be also called namespace (i.e. the space in which the names are assigned without ambiguities). For examples, more Institutions (agencies) which operate independently can assign the same name to different artefacts, therefore they are cannot be considered as part of the same environment.

The artefacts input to a Transformation can come also from other environments than the one in which the Transformation is defined. In these cases, the artefact identifier must be accompanied by a **Namespace**, which specifies the Data Set environment, to univocally identify the artefact to retrieve (for example the Data Set).

Therefore, the reference to an artefact belonging to a different environment assume the following form:

Namespace\Name

Namespace is the identifier of the environment and *Name* is the identifier of the artefact within the environment. The separator is the backslash (\).

When the Namespace is not specified, the artefact is assumed to belong to the same environment as the Transformation.

The result of a Transformation is always assumed to belong to the same environment as the Transformation, therefore the specification of the namespace of the result is not allowed.

Within a given environment, the names of all the VTL artefacts (such as Value Domains, Sets, Variables, Components, Data Sets) are assigned by the users.

Some VTL Operators assume that a VTL environment have certain default names for some kinds of Variables or Value Domains which are needed to perform the correspondent operations (for example, the operators which transform the data type of the Measure of the input Data Sets assign a default name to the resulting Measure, the check operators assign default names to Components and Value Domains needed to represent the results of the checks). In the VTL manuals, some definite default names are adopted for explanatory purposes, however these names are not mandatory and can be personalised if needed. If VTL rules are exchanged between different VTL systems, the partners of the exchange must be aware of the names adopted by the counterparties.

The connection to the persistent storage

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

Therefore, the VTL statements reference the conceptual/logical Data Sets and not the objects in which they are persistently stored. As already mentioned, there can be any relationships between the VTL logical Data Sets and the corresponding persistent objects (one VTL Data Set in one storage object, more VTL Data Sets in one storage object, one VTL Data Set in more storage objects, more VTL Data Sets in more storage objects). The mapping between the VTL Data Sets and the storage objects is out of the scope of the VTL and is left to the implementations.

VTL Operators

As mentioned, the VTL is made of Operators, which are the basic operations that the language can do. For example, the VTL has mathematical operators (e.g. sum (+), subtraction (-), multiplication (*), division (/) …), string operators (e.g. string concatenation, substring …), comparison operators (e.g. equal (=), greater than (>), lesser than (<) …), logical operators (e.g. and, or, not …) and so on.

An Operator has some input and output Parameters, which are its a-priori unknown operands and result, have a definite role in the operation (e.g. dividend, divisor or quotient for the division) and correspond to a certain type of artefact (e.g. a "Data Set", a "Data Set Component", a "scalar Value" …).

The VTL operators are considered as functions (high-order functions[32]), which manipulate one or more input first-order functions (the operands) to produce one output first-order function (the result).

Assuming that F is the function corresponding to an operator, that P_o is its output parameter and that $P_{i\ (i=1,\dots\ n)}$ are its input parameters, the mathematical form of an operator can be written as follows:

$$P_o \ = \ F\ (P_1, \dots , P_n)$$

The function F composes the Parameters P_i to obtain P_o (as mentioned, $P_{i\ (i=1,\dots,n)}$ and P_o must be first order functions). In the common case in which the Parameters are Data Sets, F composes the Data Points of the input Data Sets $D_{i\ (i=1,\dots\ n)}$ to obtain the Data Points of the output Data Set D_o.

When an Operator is invoked, for each input Parameter an actual argument (operand) is passed to the Operator, which returns an argument (result) for the output Parameter.

Each parameter has a type, which is the data type of the possible arguments that can be passed or returned for it. For example, the parameters of a multiplication are of type *number*, because only the numbers can be multiplied (in fact for example the strings cannot). For a deeper explanation of the data types see the corresponding section.

The categories of VTL operators

The VTL operators are classified according to the following categories.

1. The **VTL standard library** contains the standard VTL operators: they are described in detail in the Reference Manual.

   On the technical perspective, the standard VTL operations can be divided into the following two sub-categories:

[32] A high-order function is a function which takes one or more other functions as arguments and/or provides another function as result.

a. The **core set of operations**; these are the primitive ones, so that all the other operations can be defined in terms of them. The core operations are:

   i. The operations that accept scalar arguments as operands and return a scalar value (for example the sum between numeric scalar values, the concatenation between *string* scalar values, the logical operation between *boolean* scalar values …).

   ii. The various kinds of Join operators, which allow to apply the scalar operations at the Data Set level, i.e. to compose Data Sets with scalar values or with other Data Sets.

   iii. Other special operators which cannot be defined by means of the previous two categories (for example the analytical functions).

b. The **non-core standard operations**; they are standard VTL operations as well but are not "primitive" and can be derived from the core operations. Examples of these operations are the ones that allow to compose Data Sets and scalar values or Data Sets and other Data Sets (besides the various kinds of Join operators and the special operators mentioned above). Examples of non-core operations are the sum between numeric Data Sets, the concatenation between *string* Data Sets, the logical operations between *boolean* Data Sets, the *union* operator, some postfix operators like *calc, filter, rename* (see the Reference Manual).

Most VTL Operators of the standard library (for example numerical, string, logical operators and others) can operate both on scalar Values and on Data Sets, an thus they have two variants: a scalar and a data set variant. The scalar variant is part of the VTL core, while the Data Set variant usually not.

Anyway, the VTL users do not need distinguish between core and non-core operators, because in the practice the use of both these categories of Operators is the same.

2. The **user-defined operators** are non-standard VTL operators that can be defined by the users in order to enhance and personalize the language if needed. VTL provides a special operator, called "*define operator*" (see the Reference Manual), for the creation of user-defined operators as well as a special syntax to invoke them.

The input parameters

The input parameters may have various goals and in particular:

- identify the model artefacts to be manipulated
- specify possible options for the operator behaviour
- specify additional scalar values required to perform the operator's behaviour.

For example, in the "Join" operator, the first N parameters identify the Data Sets to be joined while the "using" parameter specifies the components on which the join must operate.

Depending on the number of the input parameters, the Operators can be classified in:

Unary	having just one input parameter
Binary	having two input parameters

N-ary	having more input parameters

Examples of unary Operators are the change of sign, the minimum, the maximum, the absolute value. Examples of binary Operators are the common arithmetical operators (+, -, *, /). Examples of N-ary operators are the substring, the string replacement, the Join. It is also possible the extreme case of operators having zero input parameters (e.g. an operator returning the current time).

The invocation of VTL operators

Operators have different invocation styles:

- **Prefix**, only for unary operators, in which the operator is written before the operand; the general forms of invocation is:

  Operator Operand (e.g. *-D_2* which changes the sign of *D_2*)

- **Infix**, only for binary operators. The operator symbol appears between the operands; the general form of invocation is:

  FirstOperand Operator SecondOperand (e.g. *$D_1 + D_2$*)

- **Postfix**, only for unary operators. The operator symbol appears after the operand in square brackets and follows its operand; the general forms of invocation is:

  Operand [Operator]

  (e.g. *DS_2* ***[filter** M_1**>0]*** which selects from Data Set *DS_2* only the Data Points having values greater than zero for measure M_1 and returns such values in the result Data Set.)

  Postfix operators are also called “clause operators” or simply “clauses”.

- **Functional**, for N-ary operators. The operator is invoked using a functional notation; the general form of invocation is:

  Operator(IO_1, … , IO_N) where *IO_1, … , IO_N* are the input operands;

  For example, the syntax for the exponentiation is *power(base, exponent)* and a possible invocation to calculate the square of the numeric Data Set *D_1* is *power(D_1, 2)*.

  The comma (“,”) is the separator between the operands. Parameter binding is fully positional: in the invocation, actual parameters are passed to the Operator in the same positional order as the corresponding formal parameters in the Operator syntax. Parameters can be mandatory or optional: usually the mandatory ones are in the first positions and the optional ones in the last positions. An underscore (“_”) must be used to denote that optional operand is omitted in the invocation; for example, this is a possible invocation of *Operator1(P_1, P_2, P_3)*, where *P_2, P_3* are optional and *P_2* is omitted:

  Operator1 (** IO_1**, _ ,** IO_3 **).

  One or more unspecified operands in the last positions can be simply omitted (including the relevant commas); for example, if both *P_2, P_3* are omitted, the invocation can be simply:

***Operator1 (** IO_1 **)**.*

- **Functional with keywords** (a functional syntax in which some parameters are denoted by special keywords); in this case each operator has its own form of invocation, which is described in the reference manual. For example, a possible invocation of the Join operator is the following:

  inner_join (** D_1 , D_2 **using [** Id_1, Id_2 **])

  In this example, the Data Sets D_1 and D_2 are joined on their Identifiers Id_1 and Id_2. The first two parameters do not have keywords, then the keyword "using" is used to specify the list of Components to join (the square brackets denote a list). A keyword can be composed of more words, substitutes the comma separator and identifies the actual parameter of the Operator. The unspecified optional parameters identified by keywords can be simply omitted (including the relevant keywords, i.e., the underscore "_" is not required). The actual syntax of this kind of operators and the relevant keywords are described in detail in the Reference Manual.

The syntax for the invocation of the user-defined operators is functional.

Independently of the kind of their syntax, the behaviour of the VTL operators is always functional, i.e. they behave as high-order mathematical functions, which manipulate one or more input first-order functions (the operand Data Sets) to produce one output first-order function (the result Data Set).

Level of operation

The VTL Operators can operate at various levels:

- Scalar level, when all the operands and the result are scalar Values
- Data Set level, when at least one operand is a Data Set
- Component level, when the operands and the result are Data Set Components

At the **scalar level**, the Operators compose scalar literals to obtain other scalar Values. The sum, for example, allows summing two scalar numbers and obtaining another scalar number. The behaviour at the scalar level depends on the operator, does not need a general explanation and is described in detail in the Reference Manual. Examples of operations at the scalar level are:

D_r *:= 3 + 7*	*3* and *7* are scalar literals of *number* type
D_r *:= "abcde" || "fghij"*	"*abcde*" and "*fghij*" are scalar literals of *string* type

As already mentioned, the outcome of an operation at the scalar level is a Data Set without Components that contains only a scalar Value.

At the **Data Set level**, the Operators compose Data Sets and possibly scalar literals in order to obtain other Data Sets. As mentioned, the VTL is designed primarily to operate on Data Sets and produce other Data Sets, therefore almost all VTL operators can act on Data Sets, apart some few trivial exceptions (e.g. the parenthesis). The behaviour at the Data Set level deserves a general explanation that is given in the following sections. Examples of operations at the Data Set level are:

$D_r := D_1 + 7$	D_1 is a Data Set with numeric Measures, *7* is a scalar *number*
$D_r := D_1 + D_2$	D_1 and D_2 are Data Sets having Measures of *number* type
$D_r := D_3$ || *"fghij"*	D_3 is a Data Set with *string* Measures, *"fghij"* is a scalar *string*
$D_r := D_3$ || D_4	D_3 and D_4 are Data Sets having Measures of *string* type

At the **Component level**, the Operators compose Data Set Components and possibly scalar literals in order to obtain other Data Set Components. A Component level operation may happen only in the context of a Data Set operation, so that the calculated Component belongs to the calculated Data Set. The behaviour at the Data Set level deserves a general explanation that is given in the following sections. Examples of operations at the Component level are:

$D_r := D_1$ *[calc* $C_3 := C_1 + C_2$ *]*	C_1 and C_2 are numeric Components of D_1
$D_r := D_1$ *[calc* $C_3 := C_1 + 7$ *]*	C_1 is a numeric Component of D_1, *7* is a scalar *number*
$D_r := D_3$ *[calc* $C_6 := C_4$ || C_5 *]*	C_4 and C_5 are string Components of D_3
$D_r := D_3$ *[calc* $C_6 := C_4$ || *"fghij"]*	C_4 is a string Component of D_3, *"fghij"* is a scalar *string*

In these examples, the postfix operator *calc* is applied to the input Data Sets D_1 and D_3, takes in input some their components and produces in output the components C_3 and C_6 respectively, which become part of the result Data Set D_r.

The operations at a component level are performed row by row and in the context of one specific Data Set, so that one input Data point results in no more than one output Data Point.

In these last examples the assignment is used both at the Data Set level (when the outcome of the expression is assigned to the result Data Set) and at the Component level (when the outcome of the operations at the Component level is assigned to the resulting Components). The assignment at Data Set level can be either persistent or non-persistent, while the assignment at the Component level can be only non-persistent, because a Component exists only within a Data Set and cannot be stored on its own.

The Operators' behaviour

As mentioned, the behaviour of the VTL operators is always functional, i.e., they behave as higher-order mathematical functions, which manipulate one or more input first-order functions (the operands) to produce one output first-order function (the result).

The Join operators

The more general and powerful behaviour is supplied by the Join operators, which operates at Data Set level and allow to compose one or more Data Sets in many possible ways.

In particular, the Join operators allow to:

- match the Data Points of the input Data Sets by means of various matching options (inner/left/full/cross join) and by specifying the Components to match ("using" clause). For example the sentence:

  inner_join D_1*,* D_2 *using [reference_date, geo_area]* matches the Data Points of D_1, D_2 which have the same values for the Identifiers *reference_date* and *geo_area*.

- filter the result of the match according to a condition, for example the sentence

  filter $D_1\#M_1 > 0$

  maintains the matched Data Points for which the Measure M_1 of D_1 is positive.

- aggregate according to the values of some Identifier, for example the sentence

  group by [Id_1 *,* Id_2 *]*

  eliminates the Identifiers save than Id_1 and Id_2 and aggregate the Data Points having the same values for Id_1 and Id_2

- combine homonym measures of the matched Data Points according to a formula, for example the sentence

  apply $D_1 + D_2$

  sums the homonymous measures of the matched Data Points of D_1 and D_2

- calculate new Components (or new values for existing Components) according to the desired formulas, also assigning or changing the Component role (Identifier, Measure, Attribute), for example:

  calc measure $M_3 := M_1 + M_2$ *, attribute* $A_1 := A_2 \,||\, A_3$

  calculates the measure M_3 and the Attribute A_1 according to the formulas above

- keep or drop the specified Measures or Attributes, for example the sentence

  keep $[M_1, M_3, A_1]$

  maintains only the specified measures and attributes, instead the sentence

  drop $[M_2, A_2, A_3]$

  drops only the specified measures and attributes

- rename the specified Components, for example:

  rename $[M_1$ *to* M_{10} *,* I_1 *to* $I_{10}]$

As shown above, the Join operator, together with the other operators applied at scalar or at Component level, allows to reproduce the behaviour of the other operators at a Data Set level (save than some special operator) and also to achieve many other behaviours which are impossible to achieve otherwise.

Anyway, even if the *join* would cover most of the VTL manipulation needs, the VTL provides for a number of other Operators that are designed to support the more common manipulation needs in a simpler way, in order to make the use of the VTL simpler in the more recurrent situations. Their features are naturally more limited than the ones of the *join* and a number of default behaviours are assumed.

The following sections explain the more common default behaviours of the Operators other than the Join.

Other operators: default behaviour on Identifiers, Measures and Attributes

The default behaviour of the operators other than the Join, when they operate at Data Set level, is different for Identifiers, Measures and Attributes.

In fact, unless differently specified, the Operators at Data Set level act only on the Values of the Measures. The Values of Identifiers are usually left unchanged, save for few special operators specifically aimed at manipulating Identifiers (for example the operators which make aggregations, either dropping some Identifiers or according the hierarchical links between the Code Items of an Identifier). The Values of the Attributes, instead, are manipulated by default through specific Attribute propagation rules explained in a following section.

For example, considering the Transformation $D_r := ln (D_1)$, the operation is applied for each Data Point of D_1, the values of the Identifiers are left unchanged and the values of all the Measures are substituted by their natural logarithm (it is assumed that the Measures of D_1 are all numerical).

Similarly, considering the simple operation $D_r := D_1 + 7$, the addition is done for each Data Point of D_1, the values of the Identifiers are left unchanged and the number 7 is added to the values of all the Measures (it is assumed that the Measures of D_1 are all numerical).

As for the structure, like in the examples above, the Identifiers of the result Data Set D_r are the same as the Identifiers of the input Data Set D_1 (save for the special operators specifically aimed at manipulating Identifiers), and by default also the Measures of D_r remain the same as D_1 (save for the operator which change the basic scalar type of the operand, this case is described in a following section). The Attribute Components of the result depend instead on the Attribute propagation rule.

In the previous examples, only one Data Set is passed in input to the Operator (other possible operands are not Data Sets). The operations on more Data Sets, like $D_r := D_1 + D_2$, behave in the same way than the operations on one Data Set, save that there is the additional need of a preliminary matching of the Identifiers of the Data Points of the input Data Sets: the operation applies on the matched Data Points.

For example, the addition $D_1 + D_2$ above happens between each couple of Data Points, one from D_1 and the other from D_2, whose Identifiers match according to a default rule (which is better explained in a following section). The values of the homonymous Measures of D_1 and D_2 are added, taken respectively from the D_1 and D_2 Data Points (the default rule for composing the measure is better explained in a following section).

The Identifier Components and the Data Points matching

This section describes the default Data Points matching rules for the Operators which operate at Data Set level and which do not manipulate the Identifiers (for example, the behaviour of the Operators which make aggregations is not the same, and is described in the Reference Manual).

As shown in the examples above, the actual behaviour depends also on the number of the input Data Sets.

If just one input Data Set is passed to the operator, the operation is applied for each input Data Point and produces a corresponding output Data Point. This case happens for all the unary operators, which have just on input parameter and therefore cannot operate on more than one

Data Set (e.g. *ln* (D_1)), and for the invocations of unary operators in which just one Data Set is passed to the operator (e.g. $D_1 + 7$).

If more input Data Sets are passed to the operator (e.g. $D_1 + D_2$), a preliminary match between the Data Points of the various input Data Sets is needed, in order to compose their measures (e.g. summing them) and obtain the Data Points of the result (i.e. D_r). The default matching rules envisage that the **Data Points are matched when the values of their homonymous Identifiers are the same**.

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 example above shows what happens under a **strict constraint**: when the input Data Sets have exactly the same Identifier Components. The result will also have the same Identifier Components as the operands.

However, various Data Set operations are possible also under a more **relaxed constraint**, that is when the Identifier Components of one Data Set are a superset of those of the other Data Set[33].

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***
2012	U.K.	12.5
2012	Germany	16.7
2013	U.K.	12.4
2013	Germany	16.2

When the relaxed constraint is applied, therefore, the Data Points are matched when the values of their **common** Identifiers are the same.

[33] This corresponds to the "outer join" form of the join expressions, explained in details in the Reference Manual.

More formally, let *F* be a generic n-ary VTL Data Set Operator, D_r the result Data Set and D_i *(i=1,... 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 than 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.

The actual constraint for each operator is specified in the Reference Manual.

Naturally, it is possible that not all the Data Sets contain the same combinations of values of the Identifiers to be matched. In the cases the match does not happen, the operation is not performed and no output Data Point is produced. In other words, the measures corresponding to of the missing combinations of Values of the Identifiers are assumed to be unknown and to have the value NULL, therefore the result of the operation is NULL as well and the output Data Point is not produced.

This default matching behaviour is the same as the one of the *inner join* Operator, which therefore is able to perform the same operation. The join operation equivalent to $D_1 + D_2$ is:

$$inner\_join\ (D_1,\ D_2\ \ apply\ D_1 + D_2\ )$$

Different matching behaviours can be obtained using the other *join* Operators, for example writing:

$$full\_join\ (D_1,\ D_2\ \ apply\ D_1 + D_2\ )$$

the *full join* brings in the output also the combination of Values of the Identifiers which are only in one Data Set, the operation is applied considering the missing value of the Measure as the neutral element of the operation to be done (e.g. 0 for the sum, 1 for the product, empty string for the string concatenation ...) and the output Data Point is produced.

The operations on the Measure Components

This section describes the default composition of the Measure Components for the Operators which operate at Data Set level and which do not change the basic scalar type of the input Measure (for example, the behaviour of the Operators which convert one type in another, say for example a *number* in a *string*, is not the same and is described in a following section).

As shown in the examples below, the actual behaviour depends also on the number of the input Data Sets and the number of their Measures.

An **Operator applied to one mono-measure Data Set** is intended to be applied to the only Measure of the input Data Set. The result Data Set will have the same Measure Component, 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 **Operators applied to one multi-measure Data Set**, by default the operation is performed on all its Measures. The result Data Set will have the same Measure Components as the operand Data Set.

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

D_1 = Import, Export, Operations

Ref.Date	*Import*	*Export*	*Operations*
2011	1000	1200	5000
2012	1300	1100	6400
2013	1200	1300	4800

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

D_r = 80% of Import & Export

Ref.Date	*Import*	*Export*	*Operations*
2011	800	960	4000
2012	1040	880	5120
2013	960	1040	3840

An Operator can be applied only on Measures of a certain basic data type, corresponding to its semantics[34]. For example, *the multiplication* requires the Measures to be of type *number*, while the *substring* will require them to be *string*. Expressions that violate this constraint are considered an error.

In general, all the Measures of the Operand Data Set must be compatible with the allowed data types of the Operator, otherwise (i.e. if at least one Measure is incompatible) the operation is not allowed. The possible input data types of each operator are specified in the Reference Manual.

Therefore, the operation of the previous example *(D_r := D_1 * 0.80)* , which is assumed to act on all the Measures of *D_1*, would not be allowed and would return an error if *D_1* would contain also a Measure which is not *number* (e.g. *string*).

In case of inputs having Measures of different types, the operation can be done either using the *join* operators, which allows to calculate each measure with a different formula (see the *calc* operator) or, in two steps, first keeping only the Measures of the desired type and then applying the desired compliant operator; the explanation, as explained in the following cases.

If there is the need to **apply an Operator only to one specific Measure**, the membership (#) operator can be used, which allows keeping just one specific Components of 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 keeps only the Import and then calculates its 80%):

D_r = 80% of the Import

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

If there is the need to **apply an Operator only to some specific Measures**, the *keep* operator (or the drop)[35] can be used, which allows keeping in the result (or dropping) the specified Measures (or also Attributes) of the input Data Set. Their invocations are:

dataset_name [keep component_name , component_name ...]
dataset_name [drop component_name, component_name ...]

For example, in the Transformation *D_r := D_1[keep Import, Export] * 0.80*

the operation keeps only the Import and the Export and then calculates its 80%):

[34] As obvious, the data type depends on the parameter for which the Data Set is passed

[35] to preserve the functional behaviour *keep* and *drop* can be applied only on Measures and Attributes, for a deeper description of these operators see the corresponding section in the Reference Manual

D_r = 80% of the Import

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

If there is the need to **perform some operations on some specific Measures and keep the others measures unchanged**, the *calc* operator can be used, which allows to calculate each Measure with a dedicated formula leaving the other Measures as they are. A simple kind of invocation is[36]:

dataset_name [calc component_name ::= cmp_expr, component_name ::= cmp_expr …]

The component expressions (*cmp_expr*) can reference only other components of the input Data Set.

For example, in the Transformation D_r := D_1 [calc Import * 0.80, Export * 0.50]

the operations apply only to Import and Export (and calculate their 80% and 50% respectively), while the Operations values remain unchanged:

D_r = 80% of the Import, 50% of the Export and Operations

Ref.Date	***Import***	***Export***	***Operations***
2011	800	1200	5000
2012	1040	1100	6400
2013	960	1300	4800

In case of **Operators applied on more Data Sets**, by default **the operation is performed between the Measures having the same names** (in other words, on the same Measures). To avoid ambiguities and possible errors, the input Data Sets must have the same Measures and the result Data Set is assumed to 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

[36] The *calc* Operator can be used also to calculate Attributes: for a more complete description of this operator see the corresponding section in the Reference Manual

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 will be 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" operator can be used to make their names equal (for a complete description of the operator 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 to Population] + D_2[rename Inhabitants to Population]

The result will be:

D_r (Population of United States + European Union)

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

Note again that the number and the names of the Measure Components of the input Data Sets are assumed to match (following their possible renaming), otherwise the invocation of the Operator is considered an error.

To avoid a potentially excessive renaming, and only when just one component is explicitly specified for each dataset by using the *membership* notation, the VTL allows the operation even if the names are different. For instance, this operation is allowed:

D_r := D_1#Residents + D_2#Inhabitants

The result Data Set would have a single Measure named like the Measure of the leftmost operand (i.e. *Residents*), which in turn can be renamed, if convenient:

D_r := (D_1#Residents + D_2#Inhabitants)[rename Residents to Population]

The following options and prescription, already described for the operations on just one multi-measure Data Sets, are valid also for operations on two (or more) multi-measure Data Sets and are repeated here for convenience:

- If there is the need to **apply an Operator only to specific Measures**, it is possible first to apply the *membership*, *keep* or *drop* operators to the input Data Sets in order to maintain only the needed Measures, like explained above for the case of a single input Data Set, and then the desired operation can be performed.
- If there is the need to **apply some Operators to some specific Measures and keep the other ones unchanged**, one of the *join* operators can be used (the choice depends on the desired matching method). The *join* operations, in fact, provides also for a *calc* option which can be invoked and behaves exactly like the *calc* operator explained above.
- Even in the case of operations on more than one Data Set, all the Measures of the input Data Sets must be compatible with the allowed data types of the Operator[37], otherwise (i.e. even if only one Measure is incompatible) the operation is not allowed.

In conclusion, the operation is allowed if the input Data Sets have the same Measures and these are all compliant with the input data type of the parameter that the Data Sets are passed for.

Operators which change the basic scalar type

Some operators change the basic data type of the input Measure (e.g. from *number* to *string*, from *string* to *date*, from *number* to *boolean* …). Some examples are the *cast* operator that

[37] As obvious, the data type depends on the parameters for which the Data Set are passed

converts the data types, the various *comparison* operators whose output is always *boolean*, the *length* operator which returns the length of a string.

When the basic data type changes, also the Measure must change, because a Variable (in this case used with the role of Measure in a Data Structure) has just one type, which is the same wherever the Variable is used[38].

Therefore, when an operator that changes the basic scalar type is applied, the output Measure cannot be the same as the input Measure. In these cases, the VTL systems must provide for a default Measure Variable for each basic data type to be assigned to the output Data Set, which in turn can be changed (renamed) by the user if convenient.

The VTL does not prescribe any predefined name or representation for the default Measure Variable of the various scalar types, leaving different organisations free to using they preferred or already existing ones. Therefore, the definition of the default Measure Variables corresponding to the VTL basic scalar types is left to the VTL implementations.

In the VTL manuals, just for explanatory purposes, the following default Measures will be used:

Basic Scalar Types	***Default Measure Variable***
String	string_var
Number	num_var
└ *Integer*	int_var
└└ *Natural*	nat_var
Time	time_var
└ *Time-instant*	date_var
└ *Time-period*	period_var
Boolean	bool_var

In some cases, in the examples of the Manuals, the default Boolean variable is also called "condition".

When the operators that change the basic data type of the input Measure are applied directly at Data Set level, the VTL do not allow performing multi-Measure operation. In other words, the input Data Set cannot have more than one Measure. In case it has more Measures, a single Measure must be selected, for example by means of the *membership* operator (e.g. dataset_name#measure_name).

The multi-measure operations remain obviously possible when the operators that change the basic data type of the input Measure are applied at Component Level, for example by using the *calc* operator.

For example, taking again the example of import, export and number of operations by reference date:

[38] In fact according to the IM, a Variable takes values in one Value Domain which represents just one basic data type, independently of where the Variable or the Value Domain are used (e.g. if they have the same type everywhere)

D_1 = Import_Export_Operations

Ref.Date	***Import***	***Export***	***Operations***
2011	1000	1200	5000
2012	1300	1100	6400
2013	1200	1300	4800

And assuming that the conversion from number to string of all the Measure Variables is desired, the following statement expressed at Data Set level *cast (D_1, string)* is not allowed because the Data Set D_1 is multi-measure, while the following one, which makes the conversion at the Component level, is allowed:

```
D1 [ calc
         import_string := cast (import, string)
       , export_string := cast (export, string)
       , operations_string := cast ( operations, string )
       ]
```

For completeness, it is worth to say that also the various Join operators allow the same operation that, for example for the inner join, would be written as:

```
inner_join (   D1  calc
                 import_string := cast (import, string)
               , export_string := cast (export, string)
               , operations_string := cast ( operations, string )

               )
```

The join operators is designed primarily to act on many Data Sets and allow applying these operations also when more Data Sets are joined.

Boolean operators

The Boolean operators (*And*, *Or*, *Not* ...) take in input boolean Measures and return booolean Measures. The VTL Boolean operators behave like the operators that change the basic scalar type: if applied at the Data Set level they are allowed only on mono-measure Data Sets, if applied at the Component level they are allowed on mono and multi-measure Data Sets.

Set operators

The Set operators (*union, intersection* ...) apply the classical set operations (union, intersection, difference, symmetric differences) to the input Data Sets, considering them as mathematical functions (sets of Data Points).

These operations are possible only if the Data Sets to be operated have the same data structure, i.e. the same Identifiers, Measures and Attributes.

For these operators the rules for the Attribute propagation are not applied and the Attributes are managed like the Measures.

The Data Points common (or not common) to the input Data Sets are determined by taking into account only the values of the Identifiers: the common Data Points are the ones, which have the same values for all the Identifiers.

If for a common Data Point one or more dependent variables (Measures and Attributes) have different values in different Data Sets, the Data Point of the leftmost Data Set are returned in the result.

Behaviour for 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 are explicitly represented when some Measures or Attributes of a Data Point have the value “NULL”, which denotes the absence of a true value (the “NULL” value is not allowed for the Identifier Components, in order to ensure that the Data Points are always identifiable).

Missing data may also show as the absence of some expected Data Point 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 absence of Data Points, however, does not necessarily denote that the phenomenon under measure is unknown. In some cases, in fact, it means that a certain phenomenon did not happen.

The handling of missing Data Points in VTL operations can be handled in several ways. One way is to require all participating Data Points used in a computation to be present and known; this is the correct approach if the absence of a Data Point means that the phenomenon is unknown and corresponds with the matching method of the *inner join* operator. Another way is to allow some, but not all, Data Points to be absent, when the absence does not mean that the phenomenon is unknown; this corresponds to the behaviour of the left and full join Operator.

On the basic level, most of the scalar operations (arithmetic, logical, and others) return `NULL` when any of their arguments is `NULL`.

The general properties of the `NULL` are the following ones:

- **Data type.** The `NULL` value is the only value of multiple different types (i.e., all the nullable scalar types).
- **Testing**. A built-in Boolean operator **is null** can be used to test if a scalar value is `NULL`.
- **Comparisons.** Whenever a `NULL` value is involved in a comparison (>, <, >=, <=, in, not in, between) the result of the comparison is `NULL`.

- **Arithmetic 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-valued 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* clauses and *is null* 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 *is null* refer to the specific sections in the Reference Manual.

```
Dr :=  D1 [M1 := if M1 is null then 1 else M1] * D2
```

Behaviour for Attribute Components

Given an invocation of one Operator F, which can be written as $D_r := F(D_1, D_2, \ldots , D_n)$, and considering that the input Data Sets D_i *(i=1,... 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 (the specific behaviours of the various operators are described in the Reference Manual).

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 implicitly 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), however the real uses of the Attribute Components are very heterogeneous.

The VTL has different default behaviours for Attributes and for Measures, to comply as much as possible with the relevant manipulation needs.

At the Data Set level, the VTL Operators manipulate by default only the Measures and not the Attributes.

At the Component level, instead, Attributes are calculated like Measures, therefore the algorithms for calculating Attributes, if any, can be specified explicitly in the invocation of the Operators. This is the behaviour of clauses like *calc*, *keep*, *drop*, *rename*, and so on, either inside or outside the *join* (see the detailed description of these operators in the Reference Manual).

The Attribute propagation rule

The users that want also to automatize the propagation of the Attributes' Values when no operation is explicitly defined can optionally enforce a mechanism, called Attribute Propagation rule, whose behaviour is explained here. The adoption of this mechanism is optional, users are free to allow the attribute propagation rule or not. The users that do not want to allow Attribute propagation rules simply will not implement what follows.

The **Attribute propagation rule** is made of two main components, namely the "virality" and the "default propagation algorithm".

The "**virality**" is a characteristic to be assigned to the Attributes Components which determines if the Attribute is propagated automatically in the result or not: a "**viral**" Attribute is propagated while a "**non-viral**" Attribute is not (being a default behaviour, the virality is applied when no explicit indication about the keeping of the Attribute is provided in the expression). If the virality is not defined, the Attribute is considered as non-viral.

The virality is also assigned to the Attribute propagated in the result Data Set. By default, a viral Attribute in the input generates a homonymous viral Attribute also in the result. Vice- versa, by default a non-viral Attribute in the input generates a non-viral Attribute also in the result (this happens when the Attribute in the result is calculated through an explicitly expression but without specifying explicitly its virality). The default assignation of the virality can be overridden by operations at Component level as mentioned above, for example *keep* (i.e., to keep a *non-viral* Attribute or not to keep a *viral* one) and *calc* to alter the virality in the result Data Set, (from *viral* to *non-viral* or vice-versa)[39].

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 too;
- 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;
- whenever in the application of a VTL operator the input Data Points are not combined as for their Measures (i.e., one input Data Point can result in no more than one output Data Point), the values of the viral Attributes are simply copied from the input Data Point to the (possible) output Data Point (obviously, this applies always in the case of unary Operators which do not make aggregations);
- Whenever in the application of a VTL operator two or more Data Points (belonging to the same or different Data Sets) are combined as for their Measures to give one output Data Point, the default propagation algorithm associated to the viral Attribute is applied, producing the Attribute value of the output Data Point. This happens for example for the unary Operators which aggregate Data Points and for Operators which combine the Data Points of more input Data Sets; in the latter case, the Attributes having the same names in such Data Sets are combined.

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:

[39] In particular, the *keep* clause allows the specification of whether or not an attribute is kept in the result while the *calc* clause makes 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.

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.

Another example can be given for operations involving more input Data Sets (e.g. D_r := D_1 + D_2). 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 or not (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_1 = Births & Deaths of the European Union

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

Suppose that the default propagation algorithm associated to the "Estimate" variable works as follows:

- each value of the Attribute is associated to a default weight;
- 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.

Assuming the weights 1 for "false" and 2 for "true", the Transformation Dr := D1 + D2 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

In an expression, the default propagation of the Attributes is performed always in the same order of execution of the Operators of the expression, which is determined by their precedence and associativity rules, as already explained in the relevant section.

For example, recalling the example already given example:

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

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

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

Properties of the Attribute propagation algorithm

An Attribute default propagation algorithm is a user-defined operator that has a group of Values of an Attribute as operands and returns just one Value for the same Attribute.

An Attribute default propagation algorithm (here called A) must ensure the following properties (in respect to the application of a generic Data Set operator "§" which applies 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)

$A(§(D_1)) = A(D_1)$

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).

Governance, other requirements and future work

The SDMX Technical Working Group, as mandated by the SDMX Secretariat, is responsible for 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.

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. In this way, 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 an operator which allows to define new custom operators by means of the existing ones and another operator (Evaluate) whose purpose is to invoke an external calculation function (routine), provided that this is compatible with the VTL IM, basic principles and data types.

As already mentioned, because the user-defined operators does not belong to the standard library, they are not standard VTL operators and are applicable only in the context in which they have been defined. In particular, if there is the need of applying user-defined operators in other contexts, their definitions need to be pre-emptively shared.

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 that use it.

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

As mentioned, whilst an Organisation adopting VTL can extend its own library by defining customized parts and by implementing external routines, on its own total responsibility, in order to improve the standard language for specific purposes (e.g. for supporting possible algorithms not permitted by the standard part), it is important that the customized parts remain compliant with the VTL IM and the VTL fundamentals. Adopting Organizations are totally in charge of any activity for maintaining and sharing their customized parts. Adopting Organizations are also totally in charge of any possible maintenance activity to maintain the compliance between their customized parts and the possible standard VTL future evolution.

Relations with the GSIM Information Model

As already said, GSIM artefacts are used as much as possible for the VTL IM. Some differences between this model and GSIM are due to the fact that, in the VTL IM, both unit and dimensional data are considered as first-order mathematical functions having independent and dependent variables and are treated in the same way.

As explained later, VTL is inspired by GSIM 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 that 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 that 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.

Data Sets and Data Structures

The VTL Data Set and Data Structure artefacts are similar to the corresponding GSIM artefact. VTL, however, does not make a distinction between Unit and Dimensional Data Sets and Data Structures.

In order to explain the relationships between VTL and GSIM, the distinction between Unit and Dimensional Data Sets can be introduced virtually even in the VTL artefacts. In particular, the

GSIM Data Set may be a GSIM Dimensional Data Set or a GSIM Unit Data Set, while a VTL Data Set may (virtually) be:

> either a (virtual) **VTL Dimensional Data Set**: a kind of (Logical) Data Set describing groups of units of a population that may be composed of many units. This (virtual) artefact would be the same as the GSIM Dimensional Data Set;
>
> or a (virtual) **VTL Unit Data Set**: a kind of (Logical) Data Set describing single units of a population. This (virtual) artefact would be 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 would not correspond to the GSIM Unit Data Set, because the latter cannot be considered as a mathematical function: in fact, it can have many GSIM Unit Data Records with different structures.

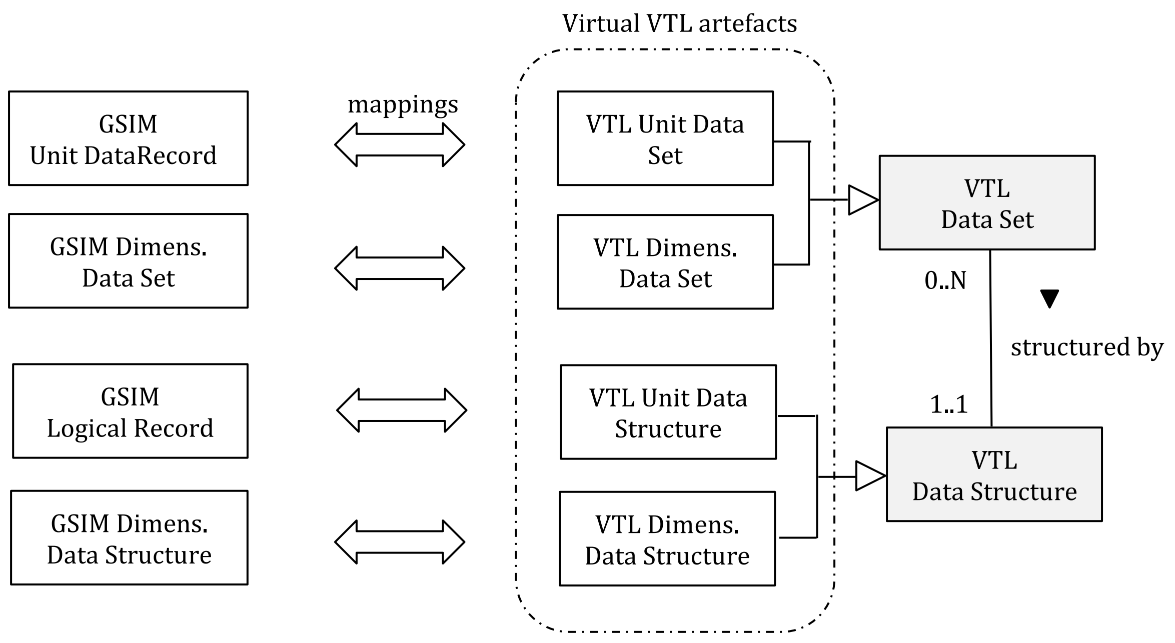
A similar relationship exists between VTL and GSIM Data Structures. In particular, introducing in VTL the virtual distinction between Unit and Dimensional Data Structures, while a GSIM Data Structure may be a GSIM Dimensional Data Structure or a GSIM Unit Data Structure, a VTL Data Structure may (virtually) be:

> either a (virtual) **VTL Dimensional Data Structure**: the structure of (0...n) Dimensional Data Sets. This artefact would be the same as in GSIM;
>
> or a (virtual) **VTL Unit Data Structure**: the structure of (0...n) Unit Data Sets. This artefact would be the same as the Logical Record in GSIM, which corresponds to a single structure and can be thought as the structure of a mathematical function. The difference is that the VTL Unit Data Structure would not correspond to the GSIM Unit Data Structure, because the latter cannot be considered as the structure of a mathematical function: in fact, it can have many Logical Records with different structures.

The following diagram summarizes the relationships between the GSIM and the VTL Data Sets and Data Structures, according to the explanation given above.

Please take into account that the distinction between Dimensional and Unit Data Set and Data Structure is not used by the VTL language and is not part of the VTL IM. This virtual distinction is highlighted here and in the diagram below just for clarifying the mapping of the VTL IM with GSIM and DDI.

GSIM – VTL mapping diagram about data structures:



Value Domains

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.

Transformation model and Business Process Model

VTL is based on a model for defining mathematical expressions that is called "Transformation model". GSIM does not have a Transformation model, which is however available in the SDMX IM. The VTL IM has been built 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. 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 can be part of the future work on VTL.

Annex 1 - EBNF

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

EBNF is a standard[40] 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 also be expressed in EBNF (although expressions are typically lengthier).

Intuitively, the 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 can be found at http://en.wikipedia.org/wiki/Extended_Backus–Naur_Form

Properties of VTL grammar

VTL can be described in terms of a Context-Free grammar[41], 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), and 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[42]), 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.

[40] ISO/IEC 14977

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

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