SDMX Technical Working Group

VTL Task Force

VTL - version 1.1
(Validation & Transformation Language)

Part 1 - General Description

(DRAFT FOR PUBLIC REVIEW)

October 2016

30

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 draft version of VTL 1.1.

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) had to be adopted, while appropriate SDMX formats for storing and exchanging rules, and web services to retrieve them, had to be designed. The present VTL 1.1 package is only concerned with the first element, i.e. a formal definition of each operator, together with a general description of VTL, its core assumptions and the information model it is based on.

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, whether described as dimensional tables or as 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.) but would provide a "neutral" expression at business level of the processing taking place, against which various implementations can be mapped. Experience with existing examples suggests that this goal would be attainable.

An important point that emerged is that several standards are interested in such a language. However, each standard operates on its model artefacts and produces artefacts within the same model (property of closure). To cope with this, VTL has been built upon a very basic information model (VTL IM), taking the common parts of GSIM, SDMX and DDI, mainly using artefacts from GSIM 1.1, somewhat simplified and with some additional detail. This way, existing standards (GSIM, SDMX, DDI, others) may 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 a logical level, independent 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 (such as SDMX). In particular, the work for the SDMX implementation of VTL is taking place in parallel to the work for designing the VTL 1.1 version, and will entail a future update of the SDMX documentation.

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 present draft version 1.1, which contains many new features.

The VTL 1.1 package, containing the general VTL specifications independent of other standards possible implementations, will include, in its final release:

a) Part 1 – the user manual, highlighting the main characteristics of VTL, its core assumptions and the information model on which the language is based;
b) Part 2 – the reference manual, containing the full library of operators ordered by category, including examples; this version will support more validation and compilation needs compared to VTL 1.0.
c) eBNF notation (extended Backus-Naur Form) which is the technical notation to be used as a test bed for all the examples.

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

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

Acknowledgements

The VTL specifications has been prepared thanks to the collective input of experts from Bank of Italy, Bank for International Settlements (BIS), European Central Bank (ECB), Eurostat, ILO, INEGI-Mexico, 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, 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, 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').

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 main categories of operators and functions included in this version of the VTL-ML syntax are:

General purpose	(e.g. assignment, data access, data storage …)
String	(e.g. substring, concatenation, length …)
Numeric	(e.g. +, -, *, /, round, absolute value …)
Boolean	(e.g. and, or, not …)
Date	(e.g. string from date)
Set	(e.g. union, intersection, …)
Statistical	(e.g. aggregate, analytic functions …)
Data validation	(e.g. check ... of value domains, references, figures …)
Time series	(e.g. time shift …)
Conditional	(e.g. if-then-else …)
Clauses	(e.g. keep, calc, attrcalc …)

The VTL-ML includes operators for defining:

IM artefacts	(e.g. Dataset, Datastructure …)
Ruleset	(e.g. mapping …)

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 statistical production and services (HLG) expressed their wish of having 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.

As a consequence, 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 first part of the document is dedicated to the description of the general characteristics of VTL.

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

A third part explains the language fundamentals, i.e. the basic characteristics of manipulated objects, operators, expressions, user-defined functions, core and derived parts of the language and so on.

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

A final part highlights some issues related to the governance of VTL developments and to future work, following a number of comments, suggestions and other requirements which were submitted to the task-force in order to enhance the 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 1.0 package is available at https://sdmx.org/?page_id=5096. The VTL 1.1 representation will be added as soon as it is available.

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);
    - At a later stage, operators for time series processing (e.g. moving average, seasonal adjustment, correlation) operators for statistics (e.g. aggregation, mean, median, percentiles, variance, indexes, correlation, sampling, inference, estimation);

Integrated approach

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

and mapping this IM to the models of the different standards (SDMX, DDI, GSIM, ...); to the extent that the mapping is straightforward and unambiguous, the language can be inherited by other standards (with the proper adjustments);

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

Active role for processing

- The language is designed to 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 automatically parsed (against the rules of the formal grammar); on the IT level, this requires the implementation of a parser that compiles the expressions and checks their correctness;
  - To be automatically translated from the VTL to the language of the IT tool to be used for the calculation; on the IT level, this requires the implementation of a proper translator;
  - To be automatically translated from one VTL version to another, e.g. following an upgrade of the VTL syntax; on the IT level, this requires the implementation of a proper translator also.

---

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

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

- ➪ The inputs and the outputs of the calculations and the calculations themselves are artefacts of the IM
    - This is a basic property of any robust language because it allows calculated data to be operands of further calculations;
    - If the artefacts are persistently stored, their definition is persistent as well; if the artefacts are non-persistently stored (used only during the calculation process like input from other systems, intermediate results, external outputs) their definition can be non-persistent;
    - Because the definition of calculations is based on the data structure definition of its input artefacts, the latter must be available when the calculation is defined;
    - The VTL is designed to make the data structure of the output of a calculation deducible from the calculation algorithm and from the data structure of the operands (this feature ensures that the calculated data can be defined according to the IM and can be used as operands of further calculations);
    - In the IT implementation, it is advisable to automate (as much as possible) the structural definition of the output of a calculation, in order to enforce the consistency of the definitions and avoid unnecessary overheads for the definers.
- ➪ The VTL and its information model make it possible to check automatically the overall consistency of the definition of the calculations, including with respect to the artefact of the IM, and in particular to check:
    - the correctness of the expressions with respect to the syntax of the language
    - the integrity of the expressions with respect to their input and output artefacts and the corresponding structures and properties (for example, the input artefacts must exist, their structure components referenced in the expression must exist, qualitative data cannot be manipulated through quantitative operators, and so on)
    - the consistency of the overall graph of the calculations (for example, 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;
    - To use the language, the users need to know only the abstract view of the data and calculations and do not need to know the aspects of the IT implementation, like the storage structures, the calculation tools and so on.
- ➪ The language is not oriented to a specific IT implementation and permits many possible different implementations (this property is particularly important in order to allow different institutions to rely on different IT environments and solutions);

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

Extensibility, customizability

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

    - The external routines should be compatible with, and relate back to, the conceptual IM of the calculations as for its inputs and outputs, so that the integrity of the definitions is ensured
    - The external routines are not part of the language, so their use might be subject to some limitations (e.g. it might be impossible to parse them as if they were operators of the language)
    - The use of external routines has some drawbacks, because it may obviously compromise the IT implementation independence, the abstraction and the user orientation; therefore external routines should be used only for specific needs and in limited cases, whereas widespread and generic needs should be fulfilled through the operators of the language;
- 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 core assumptions. Adopting Organizations are totally in charge of any possible maintenance activity deriving from VTL modifications. Such extensions, however, are not recommended because they can compromise the exchange of validation rules and the use of common tools.

Language effectiveness

- The language is oriented to give full support to the various typologies of data of a statistical environment (for example dimensional data, survey data, registers data, micro and macro, quantitative and qualitative, …) described as much as possible in a coherent way, by means of common IM artefacts for their common aspects, and relying on mathematical notions, as mentioned above. The various types of statistical data are considered as mathematical functions, having independent variables (Identifiers) and dependent variables (Measures, Attributes[3]), whose extensions can be thought as logical tables (DataSets) made of rows (Data Points) and columns (Identifiers, Measures, Attributes).
- The language supports operations on the Data Sets (i.e. mathematical functions) in order to calculate new Data Sets from the existing ones, on the structure components of the Data Sets (Identifiers, Measures, Attributes), on the Data Points.
- The algorithms are specified by means of mathematical expressions which compose the operands (Data Sets, Components …) by means of operators (e.g. +,-,*,/,>,<) to obtain a certain result (Data Sets, Components …);
- The validation is considered as a kind of calculation having as an operand the Data Set to be validated and producing a Data Set containing the outcome of the validation (typically having values “true” and “false” in the measure, respectively for successful and unsuccessful validation); being a Data Set, the result of the validation can be further processed (it can be input of further calculations);

---

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

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

Evolution of VTL 1.1 in respect to VTL 1.0

Important contributions gave origin to the work that brought to this VTL 1.1 version.

Firstly, it was not possible to acknowledge immediately - in VTL 1.0 - all of the remarks received during the public review. Secondly, the publication of VTL 1.0 triggered the launch of 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 1.1, some mistakes have been corrected and new kinds of artefacts have been added in order to make the representation more complete.

The artefacts Definition Language

VTL 1.0 was initially intended to work on top of an existing standard, like 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. In other words, VTL 1.0 was not intended to define its artefacts and therefore only contains a manipulation language.

During the work for VTL 1.1, it was acknowledged as being very recommendable and useful to have a complete definition language in VTL, able to define all of the artefacts that VTL can manipulate. This is useful, first, to express structural and reusable definitions directly in VTL (even independently of other standards); second, to facilitate the use of VTL on top of other standards (through a proper mapping, the structural definitions of other standards could be translated into VTL definitions and vice-versa); third, to make it possible to check at parsing time the coherency of the VTL manipulation expressions against the structure of the artefacts to be manipulated (even defined through VTL).

Therefore, VTL 1.1 is also equipped with a definition language for VTL artefacts. In conclusion, in respect to VTL 1.0:

The VTL definition language (VTL-DL) is completely new (there is no definition language in VTL 1.0).

The VTL manipulation language (VTL-ML) has been upgraded (it is the evolution of the VTL 1.0 language).

Reusable artefacts and rules

The artefacts defined by means of the VTL definition language (e.g. a set of code items) as well as the artefacts defined by means of an existing standard (like SDMX, DDI, or others) are reusable. In fact, the VTL manipulation language can reference these so called "structural" artefacts as many times as needed.

In order to empower the capability of reusing definitions, a main requirement for VTL 1.1 has been the introduction of reusable rules (for example, validation rules defined once and applicable to different cases).

Often, the same algorithm for manipulating data can be obtained by defining and referencing either structural artefacts or reusable rules. Current practices of various organizations show that both approaches are actually used. In order to empower the ability of the organizations of acknowledging and applying transformation/validation rules defined by others, which is one of the main goals of the VTL standard, the VTL structural artefacts and reusable rules are harmonized as much as possible. If needed, it should be feasible to convert the definitions of rules specified according to one approach (e.g. through reusable rules) into the other one (e.g. structural artefacts) and vice-versa.

The reusable artefacts and rules are defined through the VTL definition language and reused 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 1.1 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 more formally the semantics of available operators, avoiding possible ambiguities about their behaviour and fostering coherent implementations. The distinction between 'core' and 'standard' library is largely of interest of the VTL technical implementers.

The suggestion above has been acknowledged, so that the VTL 1.1 manipulation language is made 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 VTL 1.0 operators (possibly enhanced) and new operators introduced with VTL 1.1.

The VTL core includes a mechanism called join expressions, described in the following sections, which allows the definition of derived dataset operators and their behaviour, including custom operators (not existing in the standard library) for specific purposes of some institutions.

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 (DataSets) made of rows (Data Points) and columns (Identifiers, Measures, Attributes). Therefore, the main artefacts to be manipulated using VTL are the logical DataSets, i.e. mathematical functions.

Accordingly, VTL uses a functional programming paradigm, meaning a paradigm that treats computations as the evaluation of mathematical functions, avoiding changing-state and mutable data (see also the Language Fundamentals section).

It was observed, however, that the functional paradigm is not completely achieved in VTL 1.0 and that in limited cases this might cause some problem.

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

New operators

VTL 1.1 introduces new operators. As already said, all of the operators of the VTL definition language are completely new. A series of other new operators has been introduced in the VTL manipulation language.

The complete list of the VTL 1.1 operators is in the reference manual.

VTL Information Model

Introduction

The VTL Information Model (IM) describes the artefacts that VTL can manipulate.

The knowledge of the artefacts is essential for performing VTL operations correctly. Therefore, it is assumed that the referenced artefacts are defined beforehand.

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.

VTL can manage persistent or temporary artefacts, the former stored persistently in the information system, the latter only used temporarily.

As already mentioned, VTL is designed to be used either on its own or on top of other standards. It provides a formal description of data at business level against which the information models of other standards can be mapped, so that through these possible mappings to the definitions of VTL, artefacts can be obtained from the definitions of the corresponding artefacts of the other standards and vice-versa.

This is the same purpose as the Generic Statistical Information Model (GSIM) and, consequently, the VTL Information Model uses GSIM artefacts as much as possible (GSIM 1.1 version)[4]. Besides, GSIM already provides a first mapping with SDMX and DDI that can be used for the technical implementation[5]. Note that the description of the GSIM 1.1 classes and relevant definitions can be consulted in the "Clickable GSIM" of the UNECE site[6]. 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.

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, the data are composed of variables, value domains, code items and similar artefacts. These are the basic bricks that compose the data structures, fundamental for understanding the meaning of the data and ensuring harmonization of various data when needed. The second section presents the generic model for these kinds of artefacts.

---

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

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

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

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).

As already said, GSIM artefacts are used as much as possible. 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 mathematical functions having independent and dependent variables and are treated in the same way.

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

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

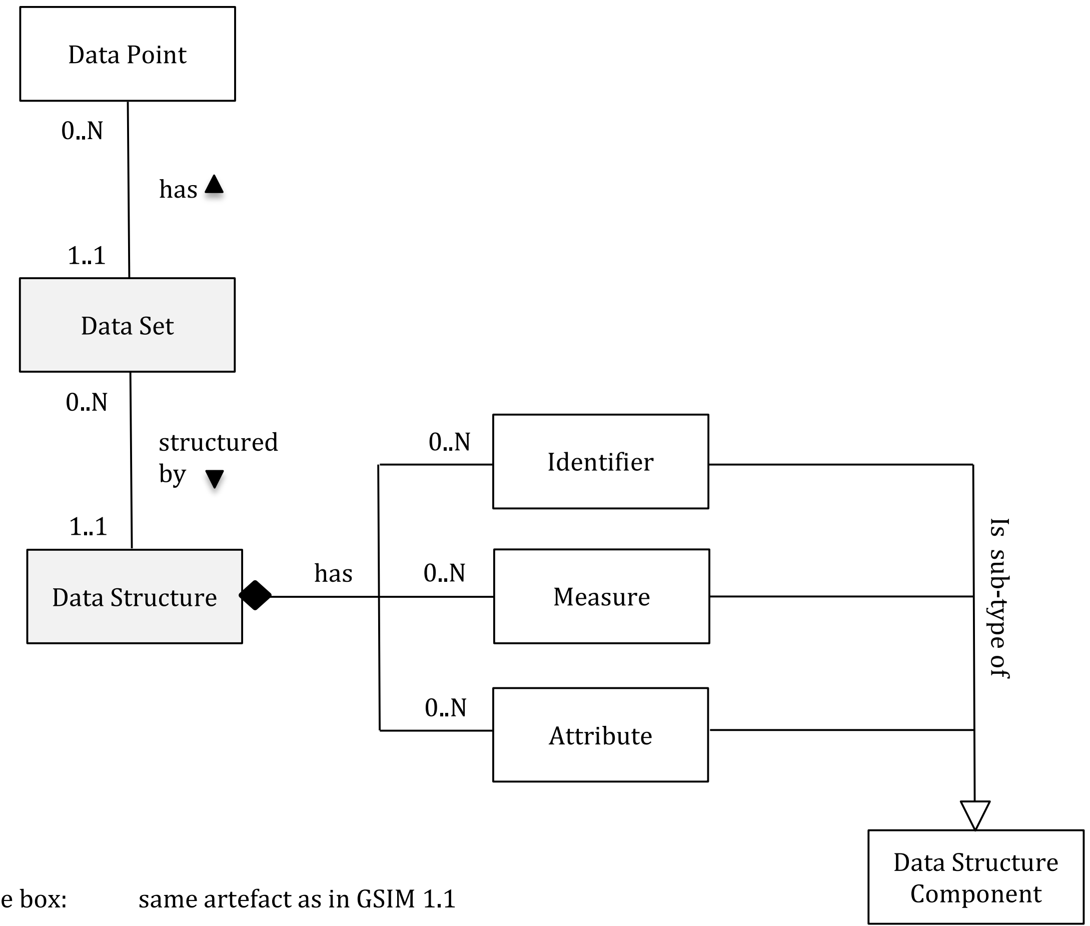
In this way, the manipulation of any kind of data (unit and dimensional) is brought back to the manipulation of very simple and well-known objects, which can be easily understood and managed by users. According to these assumptions, there would be no longer be a need to distinguish between unit and dimensional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

Starting from this assumption, each mathematical function (logical table) may be defined as a GSIM Data Set and its structure as a GSIM Data Structure, having Identifier, Measure and Attribute Components. The Identifier components are the independent variables of the function, the Measures and Attribute Components are the dependent variables. Obviously, the GSIM artefacts "Data Set" and "Data Set Structure" have to be strictly interpreted as **logical artefacts** on a mathematical level, not necessarily corresponding to physical data sets and physical data structures.

Please note that the distinction between Dimensional and Unit Data is not used at all by VTL and is not part of the VTL IM. This distinction is present in this document just for clarifying the basic mapping between the VTL IMs and the GSIM and DDI IMs.

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 DataSet is a logical set of data, in principle independent of its possible handling (exchange, calculation and so on). The right mapping is between the VTL Data Set and the SDMX Dataflow.

Data model diagram



White box: same artefact as in GSIM 1.1
Light grey box: similar to GSIM 1.1

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 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 VTL Data Set is similar to the GSIM Data Set, the relationship between them is described in the following section.

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. A set of Data Points form the extension of the function (Data Set). 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). This artefact is the same as the GSIM Data Point.

Data Structure: the structure of a mathematical function, having independent and dependent variables. The independent variables are called "Identifier components", the dependent variables are called either "Measure Components" or "Attribute Components". The distinction between Measure and Attribute components is based on their meaning: the Measure Components give information about the real world, while the Attribute components give information about the function itself. The VTL Data Structure is similar to the GSIM Data Structure, the relationship between them is described in the following section.

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

Identifier Component (or simply Identifier): a component of the data structure that is an independent variable of the function. This artefact is the same as in GSIM. In respect to SDMX, an Identifier Component may be either a **Unit Identifier**, which correspond to a SDMX Dimension, or a **Measure Identifier**, which corresponds to a SDMX Measure Dimension. The former is an identifier which contributes to the identification of the Units or Groups of Units, the latter is an identifier which contributes, when needed, to the identification of the Measure[7].

Measure Component (or simply Measure): a component of the data structure that is a dependent variable of the function and gives information about the real world. This artefact is the same as in GSIM[8].

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

Note that the VTL manages 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 is significant for the VTL.

Relationships between VTL and GSIM

As mentioned earlier, 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

---

[7] There can be from 0 to N Identifiers in a Data Structure. The particular case of 0 Identifiers and 1 Measure denotes scalar values, while the particular case of 0 Identifiers and N Measures denote vectors of scalar values.

[8] There can be from 0 to N Measures in a Data Structure. The particular case of 0 Measures denotes a "pure" relationship between the Identifiers (i.e. a relationship that does not have properties). For example, the relationship between the "students" and the "courses" that they follow (without any other information): the corresponding Data Set has StudentId and CourseId as Identifiers and do not have any explicit measure. However, as the existing combination of identifiers are implicitly considered as "TRUE", it can be thought that there is an implicit Boolean measure having the constant value "TRUE".

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;

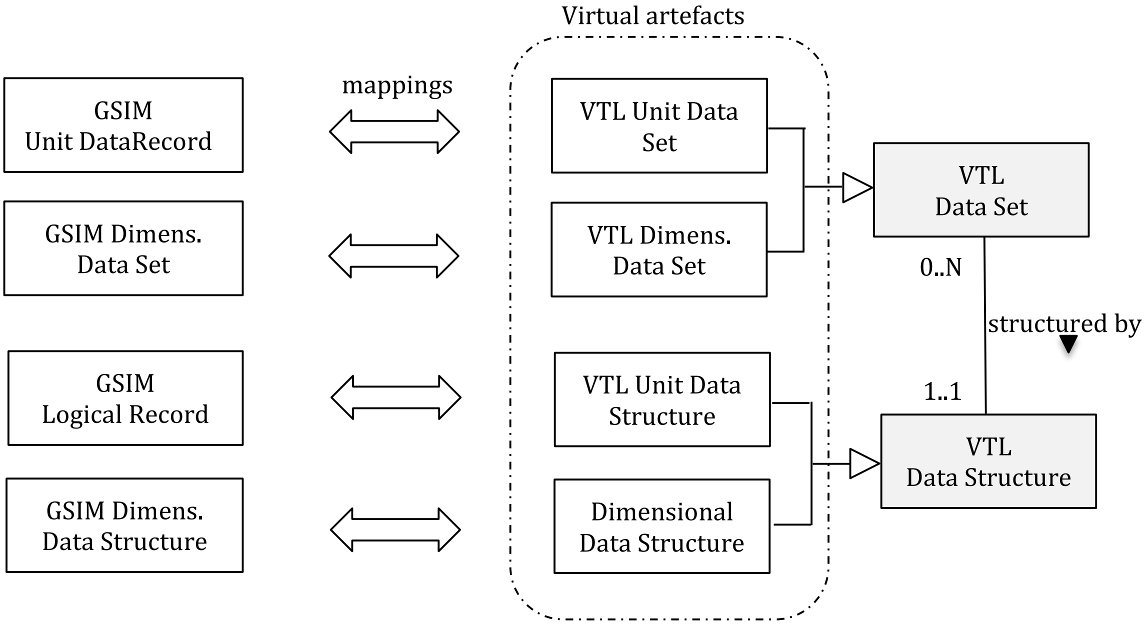
or a (virtual) **VTL Unit Data Set**: a kind of (Logical) Data Set describing single units of a population. This (virtual) artefact would be the same as the Unit Data Record in GSIM, which has its own structure and can be thought of as a mathematical function. The difference is that the VTL Unit Data 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.

GSIM – VTL mapping diagram:



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 just for clarifying the mapping of the VTL IM with GSIM and DDI.

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

Reference Date	(Identifier Component)
Country	(Identifier Component)
Measure Name	(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.

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, as a whole, is not equivalent to a proper mathematical function because its rows (i.e. the Data Points) have different structures (in term of allowed columns). However, it is easy to recognize that there exist two possible 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 2 distinct 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:

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

Data Structure 1:

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

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

Data Structure 2:

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

The data artefacts

The list of the VTL artefacts for the definition of the data is given here, together with the information that the definer have to provide. For the sake of simplicity, we may omit the parts of the names shown between parentheses.

Data Set

DataSetId	*mandatory*
DataSetDescr	*optional*
DataStructureId	*mandatory [this is the reference to the data structure of the Data Set]*
IsCollected	*mandatory [YES if the Data Set is collected, NO if it is. result of a Transformation (i.e. calculated)]*

Data Structure

DataStructureId	*mandatory*
DataStructureDescr	*optional*

(Data Structure) ***Component***

DataStructureId	*mandatory [this is part of the identifier of the Component: the data structure which the Component belongs to]*
VariableId	*mandatory [this is part of the identifier of the Component: the Represented Variable which defines the Component (see also hereinafter]*
ComponentRole	*mandatory [IDENTIFIER | MEASURE | ATTRIBUTE]*

(Sub)SetId	*optional*	*[possible reference to the (sub)Set containing the allowed values for the Component, see the section about the generic model for Variables and Value Domains]*

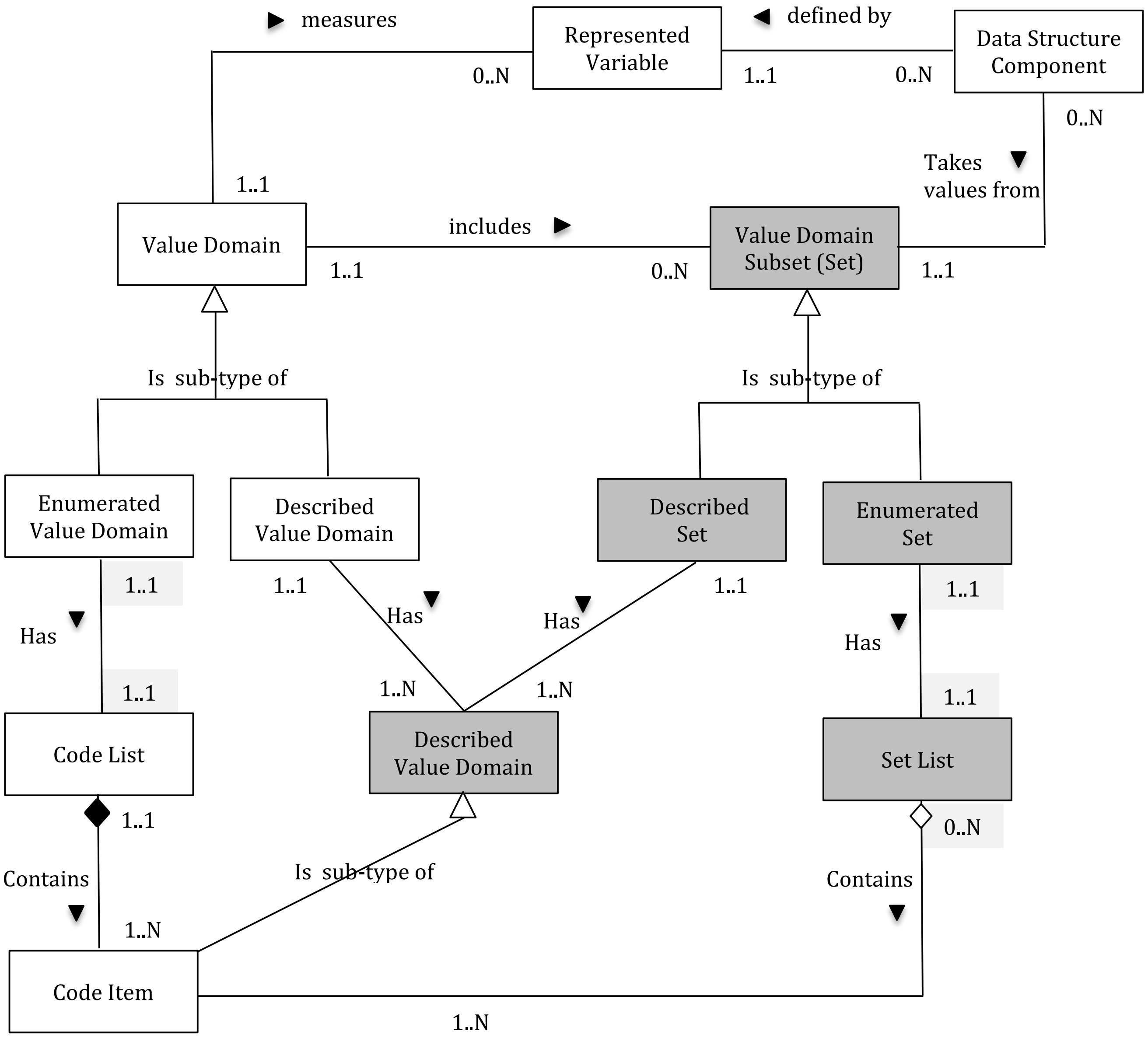
The Data Points have the same structure of the Data Sets they belong to; VTL does not require to define them beforehand.

The Validation and Transformation Definition Language introduces the operators for defining the artefacts above (see the VTL reference manual).

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 1.1
Light grey:	similar to GSIM 1.1
Dark grey	additional detail (in respect to GSIM 1.1)

Explanation of the Diagram

Even in this case, the GSIM artefacts are used as much as possible. The slight differences are mainly due to the fact that GSIM does not distinguish explicitly between Value Domains and their (Sub)Sets, while in the VTL IM this is made more explicit in order to allow different Data Structure 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 Variable, say the "place of birth", in a Data Structure takes values in the Set of the European Counties, 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 Variable can take may be very important for VTL, in particular for validation purposes.

Data Structure Component: a component of the data structure (see the explanation already given above, in the data model section). A Data Structure Component is defined by a Represented Variable (see below) and takes values in a subset of its Value Domain (this subset 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).

Represented Variable: a characteristic of a statistical population (e.g. the country of birth) represented in a specific way (e.g. through the ISO code). This artefact is the same as in GSIM. A represented variable may define any number of Data Structure Components and takes value in one Value Domain.

Value Domain: the domain of allowed values for one or more variables. This artefact is very similar to the corresponding artefact in GSIM. 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[9]. Therefore, a single Value of a Value Domain is a representation of a single "event" belonging to this space of events[10].

An important characteristic of the Value Domain is the data type (e.g. String, Number, Boolean, Date), which is the type that any Value of the Value Domain must correspond to.

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

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

For completeness, consider that in general a Value Domain can be represented also in a multi-dimensional Cartesian space, therefore a 1-dim Value Domain is a Value Domain defined in a

---

[9] 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 all the possible results.

[10] 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, more in general an "event" correspond 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).

1-dimensional Cartesian space, while a N-dim Value Domain is a Value Domain defined in a N-dimensional Cartesian space and therefore composed by 1-dim Value Domains.

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) . Although this detail does not exist in GSIM, these artefacts are compliant with the GSIM artefacts described above, representing Value Domains:

> **Value Domain Subset** (or simply **Set**): a subset of Values of a Value Domain. This artefact does not exist in GSIM, however it is compliant with the GSIM Value Domain. A Value Domain Subset has the same data type as its Value Domain and the same dimensionality. Hereinafter a Value Domain Subset is simply called **Set**, in fact a Value Domain subset can be any set of Values belonging to the Value Domain (even the set of all the values of the Value Domain).
>
>> **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). This artefact does not exist in GSIM, however it is compliant with the GSIM Described Value Domain.
>>
>> **Enumerated Value Domain Subset** (or simply **Enumerated Set**): an enumerated subset of a Value Domain (e.g. the enumeration of the European countries). This artefact does not exist in GSIM, however it is compliant with the GSIM Enumerated Value Domain.
>
> **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.
>
> **Code Item**: an allowed item of an enumerated Value Domain. A Code Item is the association of a Value with the relevant meaning (called “category” in GSIM). An example of Code Item is a single countries’ ISO code (the Value) associated to the name of 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 also the note above).
>
> **Code List**: the list of Code Items belonging to an enumerated Value Domain. This artefact is the same as in GSIM except for the multiplicity of the relationship with the Value Domain. Because of the distinction between Value Domain and Value Domain Subsets and because the Value Domain is meant to be the representation of a space of events, a Code List is assumed to contain all the possible Values of interest of the relevant Value Domain (e.g. all the possible GeoAreas of interest), therefore in the VTL IM each enumerated Value Domain has just one Code List.

Set List: the list of the Code Items belonging to an enumerated Set (e.g. the list of the ISO codes of the European countries). This artefact does not exist in GSIM. However, it has the same role than the Code List in GSIM. The Set List refers only to the Values contained in the list (e.g. the country codes), without the associated categories (e.g. the names of the countries), because the latter are already maintained in the Code List of the relevant Value Domain (which contains all the possible Values with the associated categories).

Relations and operations between Code Items

The VTL allows the representation of logical relations between Code Items, considered as events of the probability theory.

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 at least 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 giving another Code Item as result: 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. As a consequence, 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).

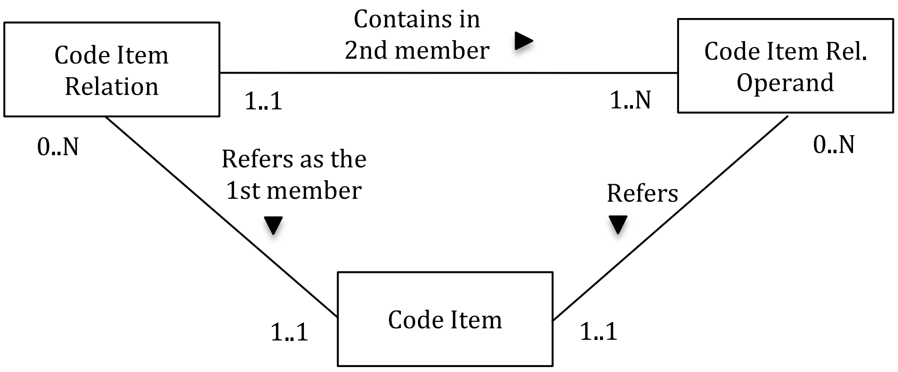
When these relations are applied to additive numeric measures (e.g. the population relevant to geographical areas), they allow the measure values to be obtained from the compound Code Items (i.e. the population of Benelux and EUwithoutUK) by summing or subtracting the measure values relevant to the component 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 possible other dimensions 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.

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:



The historical changes

The changes in the real world may induce changes in the artefacts 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.

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, 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, institutions and even departments of the same Institutions often use different conventions for representing historical changes. The VTL IM tries to manage this heterogeneity by allowing multiple options when possible and clarifying the relationships between these options.

Please note that there are two main temporal aspects: the so-called validity time and operational time. The validity time is the time during which a definition is true in the real world. The operational time is the time period during which a definition is available and may produce operational effects. In this context only the former is considered, while the latter is left to the concrete implementations of processing systems.

Even the **identification of the artefacts** is related to temporal assumptions. Regard to this aspect, two main options can be considered:

a) The artefacts are assumed to be variable in time and therefore represent a given abstraction of the reality even if it changes. As a consequence, a single artefact may represent the whole history of an abstraction. For example, under this option the same artefact (e.g. EU) may represent the European Union even if its geographic area changes (i.e. even if the participant countries change, like happened many times so far). This option follows the intuitive conceptualization in which abstractions are identified independently of time and may change with time maintaining the same identity.

b) The artefacts are assumed to be invariable in time and therefore represent a given abstraction of the reality only for the period in which this abstraction does not change. As a consequence, more artefacts have to be used to represent the whole history of an abstraction, one for each period in which the abstraction does not change. For example, under this option the European Union can be represented by more artefacts, one for each period during which its geographic area was stable (e.g. EU1, … , EU9). This option is based on the conceptualization in which the artefacts are identified in connection with the time, so that an artefact corresponds to the abstraction of some aspects of the reality (e.g. Geo Area) in association with certain times. VTL conventionally assimilates to this case also the common practice of giving a version to the artefacts for representing time changes (e.g. EUv1, … , EUv9 where v=version), being each version of the artefact assumed as invariable.

The general assumptions of VTL in relation to the representation of the historical changes are the following:

- VTL artefacts are identified and referenced by means of their univocal identifier, therefore, for VTL, in the option a) there would exist one artefact for Europe (e.g. EU) while in the option b) there would exist 9 different artefacts for Europe (e.g. EU1, … , EU9).
- possible versions of the artefacts aimed at managing temporal changes are considered to be part of the univocal artefact identifier, so that different versions are considered as different artefacts like in the option b); the Europe in this case would be represented by many artefacts (e.g. EUv1, … , EUv9). More in general, the univocal identifiers of the artefacts may be composite in the implementations, so that the adopting standards and organizations may use their own identification conventions, provided that the version is considered part of the VTL identifier.

- The characteristics of the invariable artefacts obviously cannot change with time, so they are assumed to be constant and their time validity is not explicitly considered by VTL (if required, a time validity for these artefact can be managed by the implementations).
- The variable artefacts can have characteristics variable with time. There can be many occurrences of these characteristics for the same artefact, but only one of them is valid in a time instant; the same applies to variable relations between artefacts (for example, the United Kingdom may belong to Europe only for a certain time). In these cases, each occurrence is qualified by means of a validity period (start date - end date). As obvious, the validity periods of these different occurrences cannot overlap. Validity periods are considered as "optional", because they would not be needed if the option b) is assumed. If not specified, the validity period is assumed to be "ever".
- VTL does not consider explicitly possible variations with time of the textual descriptions of the artefacts (if required, this can be managed in the implementations).

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 definer have to provide.

(Represented) Variable

VariableId	*mandatory*
VariableDescr	*optional*
ValueDomainId	*mandatory [reference to the Value Domain which measures the Variable, i.e. in which the Variable takes values]*

Value Domain

ValueDomainId	*mandatory*
ValueDomainDescr	*optional*
IsEnumerated	*mandatory [YES if the Domain is Enumerated, NO if it is Described]*
DataType	*mandatory [this is the data type of the Values of the Value Domain, i.e. one of the allowed VTL data types (see hereinafter)]*
ValueRestriction	*optional [this is a regular expression which expresses a criterion for restricting the allowed Values if needed, for example by specifying a max length, an upper or/and a lower value, and so on]*

Code List (composition) *[mandatory for Enumerated Value Domains]*

ValueDomainId	*mandatory*	*[this is part of the identifier of the Value: the Value Domain which the Value belongs to]*
ValueId	*mandatory*	*[also named Code Item, this is part of the identifier of the Value: i.e. the univocal name of the Value within the Value Domain it belongs to]*
ValueDescr	*optional*	*[in GSIM terms, this is the category associated to the Code Item]*
StartDate	*optional*	*[needed if a Value belongs to a Value Domain only for a certain period]*
EndDate	*optional*	*[needed if a Value belongs to a Value Domain only for a certain period]*

N-dimensional Value Domain

A N-dim Value Domain is a combined space of 1-dim Value Domains. It is not required to define explicitly the N-dim Value Domains, because all the possible combinations of 1-dim Value Domains are considered as defined by default. The Values of a N-dim value domains are combination of Values of the component 1-dim Value Domains.

*(Value Domain Sub)**Set***

ValueDomainId	*mandatory*	*[this is part of the Identifier of the Set: the Value Domain which the set belongs to]*
Set_Id	*mandatory*	*[this is part of the identifier of the Set: i.e. the univocal name of the Set within the Value Domain it belongs to]*
SetDescr	*optional*	
IsEnumerated	*mandatory*	*[YES if the the Set is Enumerated, NO if it is Described]*
SetCriterion	*mandatory for Described Sets*	*[a regular expression which expresses a criterion for identifying the Values belonging to the Set]*
StartDate	*optional*	*[needed if a Set belongs to a Value Domain only for a certain period]*
EndDate	*optional*	*[needed if a Set belongs to a Value Domain only for a certain period]*

Set List (composition) *[mandatory for Enumerated Sets]*

ValueDomainId	*mandatory*	*[this is part of the identifier of the Set List: reference to the Value Domain which the Set and the Value belongs to]*

SetId	*mandatory*	*[this is part of the identifier of the Set List: reference to the Set which contains the Value]*
ValueId	*mandatory*	*[this is part of the identifier of the Set List: reference to the Value which belongs to the Set]*
StartDate	*optional*	*[needed if a Value belongs to a Set only for a certain period]*
EndDate	*optional*	*[needed if a Value belongs to a Set only for a certain period]*

Code Item Relation

1stMemberDomainId	*mandatory*	*[this is part of the identifier of a Relation: reference to the Value Domain of the first member of the Relation; e.g. Geo_Area]*
1stMemberValueId	*mandatory*	*[this is part of the identifier of a Relation: reference to the Value of the first member of the Relation; e.g. Benelux]*
1stMemberCompositionId	*mandatory*	*[this is part of the identifier of a Relation: conventional name of the composition related with the first member, needed to distinguish possible different compositions related to the same first member Value. It must be univocal within the 1stMemberValueId. Not necessarily it has to be meaningful, it can be simply a progressive number ; e.g. "1"]*
CompositionDescr	*optional*	*[e.g. "Benelux from its countries"]*
Relation Type	*mandatory*	*[relation between the first and the second member, having as possible values =, <, <=, >, >=]*
StartDate	*optional*	*[needed if a Relation is valid only for a certain period]*
EndDate	*optional*	*[needed if a Relation is valid only for a certain period]*

Code Item Relation Operand

1stMemberDomainId	*mandatory*	*[this is part of the identifier of a Relation Operand: see its description above; e.g. Geo Area]*
1stMemberValueId	*mandatory*	*[this is part of the identifier of a Relation Operand: see its description above; e.g. Benelux]*
1stMemberCompositionId	*mandatory*	*[this is part of the identifier of a Relation Operand: see its description above; e.g. "1"]*

2ndMemberValueId	*mandatory*	*[this is part of the identifier of a Relation Operand: it references the ValueId of an operand; e.g. Belgium]*
Operator	*optional*	*[it specifies the applied operator, its possible values are "+" and "- "; the default is "+"; e.g. "+"]*
StartDate	*optional*	*[needed if an Operand of a Relation is valid only for a certain period]*
EndDate	*optional*	*[needed if an Operand of a Relation is valid only for a certain period]*

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[11], as DDI and GSIM do not have an explicit transformation model at the moment[12]. 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) [13] developed by the Object Management Group (OMG).

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

The basic brick of this model is the notion of 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.

---

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

[12] 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, and will be addressed in a separate work task with contributions from several standards experts.

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

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

The general form of a Transformation is the following:

```
variable parameter := expression
```

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

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

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

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

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

D_r := If ($D_1 = D_2$, then TRUE, else FALSE)

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

These are only fictitious examples for explanation purposes. The general rules for the composition of Data Sets (e.g. rules for matching their Data Points, for composing their measures …) are described in the sections below, while the actual Operators of the VTL are described in the 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[14], 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. As a matter of fact, not only the VTL but also other languages might be compliant with this specification, provided that they manipulate and produce artefacts of the information model described above. 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.

---

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

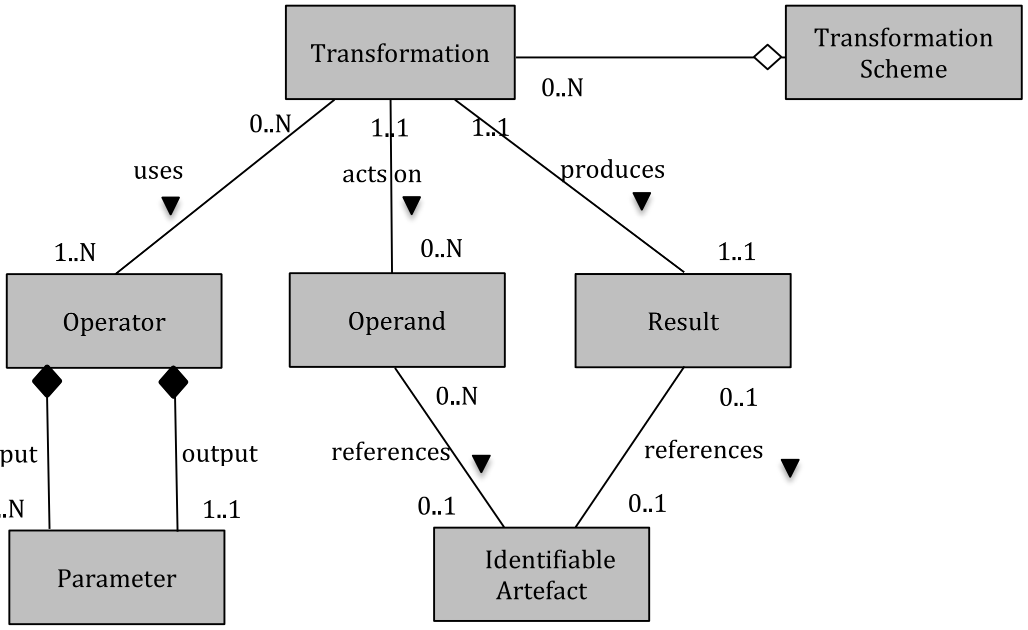
In the practical use of the language, Transformations can be composed one with another to obtain the desired outcomes. In particular, the result of a Transformation can be an operand of other Transformations, in order to define a sequence of calculations as complex as needed.

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

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

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

Transformations model diagram



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

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

Explanation of the diagram

Transformation: the basic element of the calculations, which consists of a statement which 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 string and is a property of a Transformation, as in the SDMX IM;

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 may also be considered as a VTL program;

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

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” …);

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

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

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

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

Example

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

$$D_r \quad := \quad \text{If } ((D_1 + D_2) = D_3\text{, then “true”, else “false”)}$$

In this case:

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

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

Persistency and Identification of the artefacts of the model

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

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

The VTL grammar provides for the identification of the non-persistent artefacts (see the section about the conventions for the grammar of the language) and leaves the accurate definition of the identification mechanism of the persistent artefacts to the standards adopting the VTL (e.g. SDMX, DDI …)[15].

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

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

Therefore, VTL foresees:

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

---

[15] Different standards may have different identification mechanisms.

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

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

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

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

- ESCB.analyis&insight
- EuropeanStatisticalSystem.validation
- OECD.Stat
- Unesco
- Bancaditalia.dissemination.public

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

Language Fundamentals

VTL 1.1 is a powerful language that allows the user to express complex validation and transformation operations on one or more datasets in a clear, concise and readable manner, without the need to program low-level data handling details. The whole language has been designed to simplify the problem of writing validation and transformation tasks, and to free the programmer from writing the usual boilerplate code, therefore making the program maintenance easier and reducing the chance of introducing bugs.

In the Reference Manual chapter on core operators, including the join expressions, we shall present in detail how VTL allows user to write dataset expressions using the familiar arithmetic, logic, string, date and other elementary (or scalar) operators, while the language itself takes care of all low-level details, such as joining and traversing the datasets involved in such an expression. In order to lay the foundation for such treatment, in this chapter we cover the preliminaries -- the key language concepts upon which VTL is built. Considering the power and expressiveness of VTL, there are surprisingly few of them, and the sections that follow aim at providing a thorough and not too technical overview of each of them.

Objects and Types

In VTL, an object is any entity that can be processed or computed. This includes elementary objects as small and simple as numerical, Boolean, string or date scalar values, or as large and complex as the datasets of the Information Model (IM). Whatever their size and complexity, objects share some common features:

- All objects in VTL are immutable. This means that VTL programs never change the content of an input object (e.g., a collection or a dataset), but can, when necessary, generate a new updated version, which is also immutable. VTL internally uses some clever tricks to make sure that working with immutable objects does not incur excessive penalties in terms of computing time and resources.

- Each object has a type. At runtime, each object carries with itself so-called runtime type information, which describes its structure and can be (and is) inspected by various VTL operations in order to decide how that object should be processed. But VTL is also a statically typed system, meaning that before the program is executed, the compiler uses the information about types of literals, variable parameters, and other program elements to automatically infer, or at least approximate as much as possible, the type of more and more complex program constructs. In this way, the compiler can optimize code and prevent an important and large class of potential type errors that might otherwise occur at runtime.

Type `any` is the most general type, and includes all possible objects, without telling us anything about them. On the other extreme, type `()` is the empty type, containing no objects. Nested between these two extremes are all other types in VTL, organized in the following main type families:

- *Scalar types* refer to basic numeric, string, Boolean, and date values that can be stored in a single numeric, string, Boolean, or date-time values in a tabular representation of a dataset. Type `scalar` is the most generic, denoting any scalar value, and type `null` contains only the value `null`, denoting a missing, non-applicable, or undefined scalar value. Nested between `scalar` and `null` are all other scalar types, as described in the text that follows. The scalar types include:

  - ○ `integer` -- any integer, taking implicitly into account the range of supported values, as described below in the *Basic VTL Assumptions*.
  - ○ `integer` [*a*:] -- any integer greater than or equal to some integer constant *a*.
  - ○ `integer` [:*b*] -- any integer less than or equal to some integer constant *b*.
  - ○ `integer` [*a*:*b*] -- any integer that falls between two integer constants *a* and *b*, both inclusive (where *a*<*b*).
  - ○ `integer` {x_1, ..., x_n} -- one of integers enumerated in {x_1, ..., x_n}
  - ○ `float` -- any floating-point number compatible with double-precision IEEE 754.
  - ○ `number` -- the generalization of `integer` and `float`
  - ○ `boolean` -- a Boolean value, either true or false.
  - ○ `date` -- a date-time timestamp
  - ○ `string` -- any string of characters from the UNICODE character set
  - ○ `string` [*a*] -- any strings consisting of exactly *a* characters
  - ○ `string` [*a*:*b*] -- any string consisting of between *a* and *b* characters
  - ○ `string` {s_1, ..., s_n} -- one of strings enumerated in {s_1, ..., s_n}; in effect this type describes elements of a code list.

- *Collection types* are lists and sets of elements of the same type:
  - ○ `list`<*t*> is a list of elements of type *t*. For instance, `list<integer>` is a list of integers
  - ○ `set`<*t*> is a set of elements of type *t*. For instance, `set<string>` is a set of character strings
  - ○ `collection`<t> the generalization of `list`<t> and `set`<t>
- *Dataset types*. Dataset types describe VTL datasets by summarizing the information about their structure (i.e., components) as needed by different functions and procedures operating on datasets, and as seen or inferred at compile-time:
  - ○ `dataset` -- any dataset
  - ○ `dataset` { $role_1$ $name_1$ `as` $type_1$, $role_2$ $name_2$ `as` $type_2$, ..., $role_N$ $name_N$ `as` $type_N$} -- any dataset that has exactly the listed components. Each *role* is either `identifier`, `measure` or `attribute`, each *name* must be distinct, and each *type* is a scalar type.
  - ○ `dataset` { $role_1$ $name_1$ `as` $type_1$, ..., $role_N$ $name_N$ `as` $type_N$ ...} (with "..." before the closing "}")-- any dataset that has at least the listed components, and possibly some more. Each *role* is either `identifier`, `measure` or `attribute`, each *name* must be distinct, and each *type* is a scalar type.

- *Record types*. These types are analogous to the dataset types, except that they use keyword `record` instead of `dataset`, and refer not to a complete dataset, but to an individual row in it.
- *Product types*. Type t_1 * t_2 * ... * t_n (where n>1) describes all n-tuples whose components belong to the corresponding types t1, ..., tn. E.g., `integer * string * boolean` is a type of all triples whose first component is an integer, the second component is a string, and the third component is a Boolean. For instance, `(105,"Luxembourg", false)` is a triple that belongs to this type.
- *Function types*. Type of the form *t* -> *T* describes a function that takes an object of type *t* and produces a result of type *T*. For instance, `integer -> string` is the type a function that takes an integer and returns a string. Or, `(integer * string) -> boolean` is the type of a function that takes a pair consisting of an integer and string, and returns a Boolean.

One of the objectives of the VTL type system is to encode useful information about the objects that belong to a type. That includes meta-information from the data model. Using enumerated string types, one can effectively encode a code list:

```
type BENELUX = string { "BE", "NL", "LU" }
```

```
type EU12 = string { "BE", "DE", "DK", "ES", "FR", "GR", "IE", "IT",
"LU", "NL", "PT", "UK" }
```

This is an example of two user-defined named types.

Another way the compiler can use the type information are integer computations. If the variable parameter *x* is declared as `integer[0:10]`, then the compiler can infer that the expression `y:= 2*x+3` has type `integer [3:23]`, and therefore *y* cannot be negative or zero in looping, branching, or filtering constructs.

Identifiers and Values

As in many other programming languages, VRL uses identifiers to refer to objects of different kinds. Syntactically, regular identifiers start with a (lowercase or uppercase) English alphabet letter, followed by zero or more letters, decimal digits, or underscores. However, such a regular identifier cannot be the same as a keyword or a reserved word.

Regular identifiers (just like keywords) are not case sensitive. Internally, VTL system may either convert them to uppercase or lowercase. In that sense, Pos, pos, and POS are treated as the same identifier.

Also, a regular identifier cannot start with an underscore, which denotes an argument placeholder in a function, as described below.

However, VTL 1.1 allows us to escape the limitations imposed on regular identifiers by enclosing them in single quotes (apostrophes). For instance, `'1'` is a valid VTL identifier, as well as `'_'`, `'a-2'`. `'a:b:c'`, `'a/b/c'`, or `'?x%'`. Also, `'string'` is a valid quoted identifier, while `string` is not (because it is a keyword). Quoted identifiers also may contain apostrophes, but they have to be doubled. For instance `'a''b'` is an identifier consisting of letter `a`, an apostrophe, and letter `b`. And, unlike the regular identifiers, the quoted identifiers are case-sensitive: `'Pos'` is different from `'pos'`, and both are different from `'POS'`. Whether unquoted identifier `pos` is the same as `'Pos'`, `'pos'` or `'POS'` is implementation dependent, and users are advised not to depend on any capitalization scheme in order to ensure portability of their VTL code.

VTL 1.1 makes no difference between the regular and the quoted identifiers. That is to say that wherever an identifier is expected, we can freely use one form or another.

One common use of identifiers in VTL is to store results of computations. For instance:

```
D := 0.2*D1 + 0.8*D2
```

is an assignment statement, where the expression `0.2*D1 + 0.8*D2` is computed, and (supposing that *D1* and *D2* are dataset variable parameters) the resulting dataset is stored in the variable parameter *D*. After the assignment, we can use D to refer to the computed value.

We use the word "variable parameter" for historical reasons, because that is the term commonly used in mathematics and programming. Hereinafter, we shorten this term, for sake of simplicity, to simply "variable". Please note that the same term ("variable") is used in the "VTL Information Model" section with a different meaning, i.e. as an abbreviation of Represented Variable, which is a GSIM artefact also used by the VTL IM, denoting a Statistical Variable that has a representation and can be used as a Component of a Data Structure. Hereinafter, instead, the term "variable" is used as an abbreviation of "variable parameter", so meaning an argument, a priori unknown, of an Operator of the language. Speaking about VTL expressions, therefore, variables are synonym of parameters. However, variables in VTL are less like storage locations in computer memory that can be updated at will, but more like logical variables in mathematics: they are immutable. This is to say that once the assignment is executed, we cannot change the value to which *D* refers. We are allowed to write:

```
D := 0.2*D1 + 0.8*D2

D := 1.2 * D

/* other code using D */
```

but this is internally translated into:

```
D := 0.2*D1 + 0.8*D2

U := 1.2 * D   /* U is a "fresh" variable name not appearing in the
original code */

/* other code using U instead of D */
```

In other words, the second assignment of the form "D := ..." hides the "original" value of D from the rest of the code.

To understand how variables work, we need to understand the concept of scope. A scope is a mapping from a set of identifiers visible at some point in VTL to values or objects to which they refer.

Each assignment statement changes or updates the scope for the statements that follow by associating the assigned variable name to the result of the expression to the right side of ":=". Therefore, when two statements in sequence assign to the same variable name, the first computed value of the variable is visible in the second assignment, but gets overwritten by the second assignment. This creates the illusion of variable update.

It is sometimes useful to limit the scope of variables. For instance, in formula:

```
D := (D1+D2-1-D3)/(D1+D2+1+D3)
```

it may be useful to isolate `D1+D2` and `1+D3` in an auxiliary variable A and B, which makes the code more readable:

```
A := D1 + D2

B := 1 + D3

D := (A-B)/(A+B)
```

However, we may want to limit the scope of *A* and *B* only to the computation of *D*. This can be done using a nested assignment block enclosed in curly braces:

```
D := {

  A := D1 + D2

  B := 1 + D3

  (A-B)/(A+B)
}
```

This points to a general rule in VTL: wherever an expression is expected (as, for instance, to the right of ":="), we can insert a block in curly brackets that introduces local assignments, whose visibility is limited to the block. The final element of the block must be an expression, whose result is the result of the entire block.

The whole VTL program can also be seen as one global block, implicitly closed in curly braces. It may contain zero or more assignments, and may end in a dataset expression which is, effectively, the result of the program. For compatibility with VTL 1.0 and unlike in normal blocks, we allow the last statement in the program to be an assignment, in which case the result of the whole program is the value of the last computed variable.

Expressions

Each VTL program is, essentially, an expression that takes some inputs and returns a result, which on the program level needs to be a dataset. The same holds for user-defined functions that we shall mention later: each function is defined as an expression.

Expressions are built from the following ingredients:

- Literals, such as `1` or `-105` (integer) `2.0` or `10.5e-4` (float), `"Abcdef"` (string), `true` or `false` (Boolean). As a special case, function abstractions (described in the following subsections) such as `_+_` and `\x,y{x+y}` -- both are functions that take two arguments and add them together -- can also be considered a special form of "function literals."
- Variable or column names, given as identifiers (regular or quoted).
- References to dataset components, of the form *D.X*, where *D* is a dataset variable name, and *X* is an identifier naming the component.

- Qualified names of module or object members, of the form *M*::*X*, where *M* is the name of the module or object, and *X* is the identifier naming a member of *M* (a value, function, or other object).

- Function calls of the form *name*(*arg*1, ..., *argN*) (where *N*>0), where *name* is the name of a built-in or user-defined function, and arg_1, ..., arg_N are the function call arguments.

- Built-in unary (prefix and postifx), binary (infix) and ternary (infix) operators, given in the Reference Manual. These can be used to build (sub-)expressions using the prefix, infix, or postfix operator notation.

- Join expressions, discussed in the chapter on Core Operators.

- Dataset clauses, discussed in the Reference Manual.

As usual, parentheses override binary and unary operator priorities.

Expressions in VTL are interpreted in two possible ways, depending on the context in which they appear:

- *General expressions* are those found in the top-level program assignment statements, and bodies of user-defined functions. In these expressions, identifier *X* is always interpreted as a variable name (used as a parameter in an expression), referring to a program input, function argument, or an assigned variable. General expressions can be of any type. For instance, in `A:=D1+D2`, *D1* and *D2* are variable names.

- *Component expressions* appear in record-level statements inside the join expression body and in dataset clauses. In them, identifier *X* (not followed by "." or "::") is interpreted as a dataset component name. To use variable *X*, we have to write $*X*. Component expressions are always scalar. For instance, in `D[filter X>0]`, X is not a variable name, but a component name (of dataset D). However, in `D[filter X>$Limit]`, element `$Limit` stands for variable *Limit* (which may be, for instance, a function argument).

Data Flow Optimization

As we could see in the preceding examples, expressions can be complex and may contain nested blocks that compute temporary variables. For complex block expressions, it is important to understand that in VTL their actual computation may differ from what is usually found in imperative programming languages. In the latter, each assignment is computed sequentially, followed with the computation of the final result.

It is important to understand that from the programmer's perspective, VTL block expressions produce results *as if* they are executed sequentially. For instance, in the block expression:

```
{
  A := D1 + D2
  B := 1 + D3
  D := (A-B)/(A+B)
  D    /* result */
}
```

we can logically think about the result D as being computed gradually: first A is computed, then B, and finally D. The semantics of VTL complex expressions guarantees that the final result is going to be the same *as if* such step-by-step computation has taken place. This makes it easy for the programmer to think about the programming problem and organize and write code in as clear and correct manner as possible.

However, the VTL compiler may perform data flow analysis to infer the data flow graph in the program in order to optimize the handling of datasets. For instance, computing A, B and D sequentially in the previous example would be inefficient, since A would require one dataset join and traversal ($D1$ and $D2$), B another ($D3$), and D the third (A and B). Instead, the compiler can transform this into a more efficient single join and traversal of datasets $D1$, $D2$, and $D3$, where all calculations are done in a single run. The way this optimization is done must guarantee that the result of the block is the same *as if* the computation is performed sequentially. But the actual execution strategy used by a VTL implementation can range from a centralized sequential computation, to translating programs into database or data warehouse queries, to executing different operations on different interconnected nodes in a distributed computing system, by routing or streaming data between them. Whatever execution strategy is actually used, it must be transparent to the programmer.

User-Defined Functions

VTL 1.1 adopts many features from the functional programming languages. In particular, each scalar or dataset operation and operator can be seen as a function that accepts some arguments and returns a result. This means that most of the processing can be viewed as application of functions to data. Sometimes, this is explicit in using functional notation, as in `size(D)`, but even when using infix or prefix operator notation as in `2*X-3`, this is equivalent to (and can indeed be written as) a function call of the form `'-'('*'(2,X),3)`. That makes functions one of the fundamental concepts in VTL, along with the join expressions.

There are essentially two ways to define functions in VTL 1.1. Suppose, for instance, a sorting algorithm that operates on collections of objects of some type t, which requires to be supplied with a function of type (t,t) -> `boolean`, which takes a pair of objects of type t and returns `true` exactly when the first element is considered to precede the second element (making it "smaller" in some ordering scheme). The sorting algorithm is neutral with the respect to the type t of collection elements, and it depends on this function to perform comparison.

Now, let us suppose we want to use that algorithm to sort a collection of integers in a descending order. For that we have to supply a function of type `(integer,integer)->boolean` which returns `true` for arguments (x,y) exactly when x>y. The classical way to do that is to write a named function definition of the form:

```
define function compare_integer_descending(x as integer, y as
integer)
returns boolean
as x > y
```

We can normally omit the "`returns boolean`" part, as the return type information can be inferred by the compiler from the expression "`x>y`".

This is an example of a named function definition. As a result of it, identifier compare_integer_descending refers, in the scope in which it is defined, to a function object of type `(integer,integer)->boolean`. We can then pass this function to the sorting algorithm by name, using identifier compare_integer_descending.

However, for this kind of relatively simple cases, VTL 1.1 allows us to specify a function object directly, without the need to define/create it separately. This we call the anonymous function, and in our case it can look like this:

```
\x as integer, y as integer { x > y }
```

The anonymous function starts with a backslash, followed by arguments (and optionally their types), followed by a block expression that produces the result. This is also a simple example of a function in whose body arguments appear only once, and that in the order in which are listed. When the type of the arguments is unambiguous from the context (i.e., when the compiler can decide that both arguments must be integer, because it already knows we are sorting a collection of integers, we can be even terser and write:

```
_>_
```

Here, we use underscores as placeholders for arguments. When the compiler encounters an underscore, it converts the expression in which it appears into an anonymous function, and turns each underscore into a function argument:

```
\x,y{x>y}
```

An anonymous function that computes an average of three numbers can be written as:

```
(_+_+_)/3
```

We can even write:

```
_ between _ and _
```

(noting the spaces surrounding underscores, to prevent underscores to be treated as a part of identifiers) to denote a function of type (number,number,number)->boolean which takes three numbers and checks whether the first one falls between the second and the third one.

Anonymous functions specified using underscores have a limitation that each argument can be used only once. And, by definition, the anonymous functions cannot be recursive, because they have no way of calling themselves. Therefore, to achieve more general computation tasks, we need to use the most general way for defining functions, which is using the named functions.

As seen in the example above, the general template for defining a function is:

define function *name*(*arg*$_1$, ..., *arg*$_N$)

returns *t*

as *E*

where *name* is the function identifier, and each *arg* is an identifier, optionally followed with keyword **as** and the argument type. The **returns** part is also optional, and it specifies the type of the function's result.

The return type, as well as argument types, are optional, because in many case (although not always) the compiler can infer their type from the context. For instance, function that checks if a quadratic equation $ax^2+bx+c=0$ has a solution can be defined as:

```
define function has_solution(a, b, c)
as b*b-4*a*c>0
```

The compiler knows that the comparison (>) produces a Boolean result. Also, since the right-hand side of > is a numeric literal 0, the left-hand side also has to be a number. And, since the left-hand side produces from variables a, b, and c and arithmetic operators, the compiler is able to convert the above definition into:

```
define function has_solution(a as number, b as number, c as number)
returns boolean
as b*b-4*a*c>0
```

However, it is advisable to provide argument and return types for the more complex or externally visible user-defined functions, in order to help their users, and to make the compiler check that their implementation really produces the result of the desired type.

So far, all function arguments were obligatory. For functions that perform complex operations, this may lead to a large number of function arguments, most of which have some sensible default value.Let us take, for instance, a function that computes n-degree distance between measurements in two datasets. For two real numbers x and y, the distance of n-th degree is n-th root of x^n-y^n.So, the first degree distance is simply $x-y$, the second degree distance is $sqrt(x^2-y^2)$, etc. We can write the function as:

```
define function distance(d1 as dataset,
     d2 as dataset,
     n as integer := 2)
as
  (d1^n - d2^n)^(1/n)
```

Note how we added ":= 2" in the declaration of argument n. This makes it an optional named argument, which, if not specified in a function call, takes on the default value 2. A call distance(x,y) is equivalent to distance(x,y,n:2). The optional named arguments must come after the non-optional arguments, and in a call their values are preceded with the argument name followed by a colon, "n: 2". If we have a function with more than one optional named argument, such as:

```
define function z_transform(x as number,
     mu as number := 0,
     sigma as number := 1)
as (x-mu)/sigma
```

then we can write both

```
z_transform(x, mu: 50, sigma: 4.3)
```

and

```
z_transform(x, sigma: 4.3, mu: 50)
```

That means that the relative ordering of the optional named arguments in a function call is not important, since the compiler always looks at the definition to pass the arguments in a correct sequence. However, as mentioned above, all positional (i.e., not named) arguments must come first.

Procedures

Besides functions, VTL supports procedures. Procedures differ from functions in several important respects.

- Procedures are aimed at automating common processing tasks, and can be used as a means for shortening the code by replacing common processing tasks with a procedure call. On the other hand, functions are concerned only with computing results from arguments.
- Procedures may have several input and output arguments, which are passed by reference, while the procedure call has no return value of its own. In contrast, functions defined via a single expression (which may be a complex, block expression), and exhibit so-called referential integrity. That is to say that a function call with same arguments (always passed by value) should always return the same result.

To understand procedures, let us take a simple example of a procedure that computes a quotient and a remainder of a division of measures in a dataset and a number (the same can be easily extended to two datasets):

```
define procedure quot_rem(in ds as dataset, in divisor as number,
     out quot as dataset, out rem as dataset)
as {
  quot := floor(ds / divisor)
  rem := ds - quot*divisor
}
```

We first note that each argument of a procedure is qualified as **in** or **out**. Input arguments, such as ds and divisor in our example, are passed by value, just like function arguments, and we can pass any expression with compatible type when calling the function. However, output parameters, such as quot and rem in our example, must be specified as names of variables that will hold results computed in the procedure body.

For instance, we can call the above procedure like:

```
call quot_rem(PopPerCountry, AvgPop, Multiple, Remainder)
```

and this call is equivalent to the following two assignments:

```
Multiple := floor(PopPerCountry / AvgPop)
Remainder := PopPerCountry - Multiple*AvgPop
```

Note that in our case the body of the procedure is a sequence of assignments enclosed in curly braces. In general, it is always a sequence of assignments or procedure calls. Also, any assignment in the procedure body to a variable that is not marked as output is invisible to the calling code.

Procedures may look a lot like macros, but they are much more powerful. Firstly, the body of a procedure is compiled and type checked, which means that any syntax or semantic errors in a procedure definition are detected at compile time. This extends to the type checking of input and output arguments. Finally, procedures can be stored in modules and reused.

Language Core

The ability to define user functions and procedures allows development of libraries of reusable and standardized VTL validation and transformation building blocks, which, in turn, adds to the effectiveness and expressiveness of use of VTL 1.1 in normal use case scenarios. But to be useful, these functional and procedural facilities need to rest on a solid foundation directly provided by the language. This includes the two main components:

- Core constructs, which represent the fundamental building blocks into which any dataset processing in VTL 1.1 can be decomposed, and
- Standard library, which contains a large number of utility functions and operators built from the core constructs or other standard library constructs.

Both the core constructs and the standard library are explained in detail in the Reference Manual.

The role of the core constructs is to express the semantics of simple and complex operations in VTL in an unambiguous manner. For instance, using the scalar operators '+' and '*' that add and multiply numbers, and a join expression, we can define the function:

```
define function midway(d1 as dataset {measure x as number, …},
    d2 as dataset {measure x as number, …})
returns dataset {number x as number, …}
as
    [d1 outer join d2] {
        filter d1.x is not null or d2.x is not null
        x := 0.5*d1.x + 0.5*d2.x
    }
```

which takes two dataset arguments *d1* and *d2*, each containing (at least) a numeric measure component named *x*, and returns a dataset with a numeric measure component named *avg* which is the mean of *x* from *d1* and *d2*. Without going here into too much detail, well explained in the Reference Manual, the function body after `as` is a join expression that:

- Performs a *join* of *d1* and *d2*, by matching records from *d1* and *d2* that share the same values of identifier components. The set of identifier components of *d1*must be equal to, a subset of, or a superset of, the set of identifier components of *d2*.
- The type of join is *outer*, which means that if for some record in d1 there is no matching record in *d2* (or vice versa), the join "invents" the latter with all measure and attribute component values set to `null`.
- The body of the join expression is given inside the curly braces, '{' and '}'. Inside the body, *d1* and *d2* refer to the matched records from the corresponding joined datasets.
- The `filter` statement skips the cases where the numeric measure *x* is undefined In both *d1* and *d2.* This is important, because datasets *d1* and *d2* may have more than one measure component,
- For each pair of matched records *d1* and *d2*, the result contains one record that inherits all identifier component values from *d1* and *d2*, and has a numeric measure component *x* which is computed as `0.5*d1.x + 0.5*d2.x`.

For instance, let us suppose we have these two data sets:

d1 :=

Year	Geo	X
2011	LU	104
2011	NL	812
2012	LU	97

and *d2* :=

Geo	X
LU	128
NL	768

Then `midway(d1,d2)` will produce:

Year	Geo	X
2011	LU	116
2011	NL	790
2012	LU	112.5

Incidentally, the same operation can be directly and simply written in VTL as a dataset expression:

```
    0.5*d1.x + 0.5*d2.x
```

where *d1* and *d2* are two dataset variables. This simple dataset expression is internally automatically translated by the compiler translated into the same expression as given in the body of the function given above. Note that in the dataset expression '*' is a mixed scalar/dataset operator (multiplying a scalar value 0.5 with a dataset), and '+' is a dataset operator (both operands are datasets). However, the meaning of these two scalar/dataset and dataset operators and of the entire expression does not need to be separately defined: it is systematically derived from the core operators and constructs, scalar '+' and '*' and the join, as described in the corresponding chapter below.

It is important to note that the selection of core operators and constructs is entirely driven by the language design and the need for semantic soundness. Users need not be concerned whether they are using a "core" or a "library"operator, function, or another construct. Users should always try to use the construct which is best suited for their intended purpose.

For the language implementers, the existence of the language core represents a contract that controls the correct behaviour of their VTL implementation. It does not always necessarily mean that every implementer needs to use the core constructs as the back-end. While every VTL construct needs to be expressible in terms of the language core, implementations may use more efficient backend-specific algorithms and techniques (in R, SAS, SQL, etc.). However, the implementers must ensure that the user-observable behaviour of their implementations respect the behaviour required by the contract.

Compilation Units and Dialect Selection

Programs and modules are two types of compilation units in VTL. By a compilation unit we here mean a unit of code stored in a single file or transmitted as a message. The main difference between a VTL program and a VTL module is that the former executes some particular dataset processing task (some form of validation or transformation), while the latter creates and packages functions, procedures, values, named types, and other objects so that they can be used by programs and other modules.

Since VTL comes in several versions, which may use different syntax or may interpret the same syntactic forms differently. To indicate the version or dialect of VTL used in a compilation unit, its first line (after leading whitespace and comments) should be the following directive:

```
use syntax "X.Y"
```

(optionally followed by a semicolon) where X.Y is the version number of VTL dialect in which the compilation unit is written. For instance:

```
use syntax "1.1"
```

indicates that what follows in the file uses the VTL 1.1 syntax.

The version number in **use syntax** directive can be followed by one or more of case insensitive tags of the form "+tag" where *tag* consists of one or more Latin letters, decimal digits and underscores. For instance:

```
use syntax "1.1+estat+strict"
```

may indicate VTL 1.1 syntax with custom Eurostat (ESTAT) tags, and strict type checking option.

If a VTL system does not support the version indicated in **use syntax** directive, it is obliged to reject the compilation unit and report an error. However, each VTL implementation can freely decide which tags to recognize, and should ignore all unecognized tags (possibly issuing a compile-time warning).

Program and Module Structure

A module is distinguished from a program by starting with a **module** directive after the leading whitespace, comments, and the optional **use syntax** directive. If the first thing after the leading workspace, comments and the optional **use syntax** directive is not a **module** directive, then the compilation unit is treated as a program, not module.

Module Declaration

The simplest form of the **module** directive is:

```
module name
```

(optionally terminated with a semicolon). *Name* is an identifier giving the module name. This defines a *transient* module, which is created in memory when the module is loaded by the compiler because it is used by a program or another module.

Another more complex form of the **module** directive is:

`module` *name* `in` "*AGENCY*:*ENTITY*:*VERSION*"

(optionally terminated with a semicolon). *AGENCY* is a code for the owner of *ENTITY*, which is a logical name of a persistent entity in the underlying information model used by the VTL system. For instance, in VTL systems based on SDMX, *ENTITY* refers to a named versionable artefact, such as a data structure definition or a dataflow. Finally, *VERSION* gives the version of the *ENTITY* to which the module is associated.

The latter form of the **module** directive creates a persistent module, which the VTL system associates with *ENTITY*.

Module Usage

Both VTL programs and modules can depend on other modules. These dependencies are expressed with **use module** directives. The first form:

`use module` *name*

(optionally followed by a semicolon) expresses a dependency on a transient module with the given *name* which is locally available, i.e., it is supplied together with the program or module using it, for instance as a file in the same directory tree or a part or attachment of the same message.

The second form:

`use module` *name* `in` "*AGENCY:ENTITY:VERSION*"

(optionally followed by a semicolon) expresses a dependency on a persistent module with the given *name* which is attached to the persistent *ENTITY* owned by *AGENCY*. *VERSION* is either a version number, or an asterisk (*) that signifies the latest version. In this case, depending on the underlying concrete information model (such as, for instance, SDMX), the compiler needs to retrieve the module from a

Module dependencies cannot be circular. One advantage of expressing the module dependencies with **use module** directives is that the compiler (or any other source code handling tool, such as a registry) can analyse module dependencies, construct dependency graphs, and detects any problems (such as missing modules or circular dependencies) statically, i.e., before the VTL program is deployed and run.

Definitions

A VTL program or a module can contain zero or more definitions. These include:

- Type definitions
- Function and procedure definitions
- Validation rule / rule set definitions

All definitions introduce a named object (a type, a function, a procedure, a validation rule / rule set) in the scope of the program or module.

To refer to identifier *x* in module *module*, we use the double column syntax:

name :: *x*

which is called a qualified name, in contrast with a simple identifier or simple name.

Module-Level Computations

After definitions, modules can contain computations, which take the form of assignments:

x := *E*

where *x* is a variable, and *E* is an expression. Like a definition, each assignment also associates an object which is the result of *E* with identifier *x* in the module scope, but this time using the general expression syntax. This is useful, for instance, when the module describes a data structure, and needs to have a member which is a set of tuples describing constraints on the dataset component values.

Or, a mathematical module can contain assignment:

```
PI := 4*atan(1.0)
```

Another example where computations come handy is re-exporting a named object from a used module. In the following example:

```
use syntax "1.1"
module A
use module B
/* definitions */
X := B::X
```

module A uses module B, and can refer from A to member named X in B as B::X. But, by assigning it to name X in its own scope, module A re-exports B::X as A::X which is accessible from any module using A (and not necessarily using B).

Program-Level Computations

While computations are optional in modules, they are mandatory in programs. In fact, performing a computation and returning a result is the whole purpose of a program. The computation statements consist of zero or more assignments or procedure call statements, followed by an expression which is the result of the whole program. This final expression can be omitted if the last statement in the program is an assignment; in this case, the result of the program is the result of the last assignment.

Module Instantiation and Incremental Compilation

In the preceding section, we already said that circular dependencies between modules are forbidden in VTL. In fact, we go one step further by requiring that a module needs to be *instantiated* before being used in a program or another module.

A module is instantiated when:

- All modules on which it depends (if any) are (transitively) instantiated
- All type, function, procedure, rule, etc., definitions in the module have created the corresponding objects and bound them to the names in the module scope.

- All module computations have been performed, and all values have been bound to the corresponding variable names in the module scope.

The instantiated module can be seen simply as a map from module member names to the VTL objects created from definitions or computed from assignments. Of course, on the technical level the situation is somewhat more complex, since an instantiated module also needs to carry additional information about types and module dependencies.

One advantage of this approach is that an instantiated module is not only limited to an in-memory representation, but can also be written to a persistent store in some appropriate external format -- for instance by serializing to a file, or populating database tables. Unless the module source code or some of its dependencies change, the VTL compiler needs to compile and instantiate the module only once. This may significantly improve the speed of compilation and execution of VTL programs.

Besides, by requiring that all modules used by a VTL compilation unit need to be previously instantiated, it becomes natural for the compiler to perform incremental compilation, starting from the bottom of the module dependency tree and going upwards towards the top-level target (a program or a module). A recompilation and re-instantiation of a module would be triggered only when its instantiated form is outdated or missing, or when one or more of its dependencies change.

Principle of Introspection

It has already been hinted above that one of the important uses of modules in VTL is to describe data structures of different datasets that are used in a program. Note that the dataset structure can be described in several different ways:

- Using compile-time type information -- we have already seen that the structure of a dataset can be fully or partially described using **dataset** type. The level of detail and precision of a dataset type reflects the information put into the code by the programmer and the characteristics of operations applied to the datasets.
- Using runtime type information -- each dataset at runtime carries with itself a full and precise description of its structure, as fed on input or computed in the VTL program. This information is typically more precise than the type information inferred at compile time.
- By explicitly constructing a description of dataset structure at runtime -- this means constructing VTL objects that represent dataset components, their types, roles, or constraints.

Each of these approaches has certain advantages and disadvantages. The compile-time type analysis prevents using objects that are not datasets in dataset operations, or using datasets that lack the necessary components with the required data types and roles. For instance, if *f* is a function and *ds* a dataset variable, the type system ensures that in the call:

```
f(ds)
```

ds always meets the minimum of requirements imposed on its structure by *f*. However, the compile-time type analysis is limited by what is known before a program is run and before it receives any inputs. Therefore, its characterization of datasets can be sometimes too general and coarse.

We can also define type that describes a particular dataset structure. For instance:

```
type population = dataset {
  identifier geo as string
  identifier year as integer
  measure population as float
  attribute status as string
}
```

If we define *f* to accept an argument of type *population*, the compiler raises a red flag if we try to use a dataset that may not be compliant. But what if we want to check if *ds* can be fed to *f* not in general, but in a particular case of program execution?

At runtime, each input to the program and each result of computation carries with it the precise description of its structure. If *ds* is a dataset variable, we can use **is** operator to ask:

```
ds is population
```

Note that this construct allows us to use the runtime type information of *ds* against a statically defined type *population*. If this test succeeds (returns **true**), we know that passing this particular *ds* to *f* is safe even if at compile-time we had no information to justify the safety of passing *ds* to *f* in general.

The "trick" on which this is based is that population on the right-hand side of **is** is *reified,* which is to say that it is represented as an object at runtime. Thus, **is** takes the run-time type information of *ds* and the reified type information of *population*, and compares them.

But let us go one step further, and imagine we have an arbitrary dataset *ds* and want to inspect its structure from within a VTL program.

One drawback on relying on runtime type information is that the objects describing it can be very complex and unstable in the sense that they can change from one version of the language to another. This means that if a VTL program wants to look into the structure of a dataset at runtime, it would need to rely on a very complex internal API, which would likely change as new features are added to the language.

This seems to suggest that it is better to keep the structure of the runtime type information representation hidden from the programmer. As an alternative, we can construct a simplified description of the structure, which faithfully reflects the data type.

Imagine that from the *population* data set type we generate the following module

```
use syntax "1.1"

module pop_ds

type t = dataset {

 identifier geo as string

 identifier year as integer

 measure population as float
```

```
 attribute status as string
}
structure := list(
 module {
  name := "geo"
  role := "identifier"
  type t = string
 },
 module {
  name := "year"
  role := "identifier"
  type t = integer
 },
 module {
  type t = float
  name := "population"
  role := "measure"
 },
 module {
  name := "status"
  role := "attribute"
  type t = string
 }
}
```

In this module, we have encoded the desired dataset structure in two ways: by defining a type *t* and by providing the list of objects describing individual components. Each **module** { ... } inside list is a component descriptor object.

If we have a module or a program that uses *pop_ds*:

```
use module pop_ds
```

then we can refer to the database type as:

```
pop_ds :: t
```

and if the following test returns true:

```
ds is pop_ds :: t
```

we can inspect the structure of the dataset at runtime by looking at:

```
pop_ds :: structure
```

In order to allow introspection of dataset structure for arbitrary datasets, we can use built-in function *get_dataset_structure* which takes an arbitrary dataset and returns a list of component descriptors whose structure is illustrated our example. In that sense, the dynamic introspection is still possible, but the API is kept at minimum.

Looking at the *pop_ds* module above, it becomes obvious that this kind of modules can be automatically generated from the information model. Indeed, in VTL 1.1 each dataset structure that is identifiable with *AGENCY:NAME:VERSION* coordinates behaves *as if* it has attached a VTL module describing the dataset structure in the described manner. Of course, these modules are not written by hand, but are automatically generated from the information model itself.

For instance, one can write:

```
use module pop_ds in "acme:population:*"
```

to import the dataset structure description module *pop_ds* for the latest version of *population* table owned by *acme*, and then use pop_ds::t and pop_ds::structure in the described manner.

This is the principle of automated introspection of dataset structures from the information model in VTL code.

Core Operators and Join Expressions

Scalar Core Operators

VTL 1.1 scalar operators are unary and binary operators that accept a scalar argument and return a scalar value. In this section, we present only the operators that are "natively" scalar, but can be automatically lifted to the dataset/scalar and dataset levels. There are a number of other operators that can take scalar values, but are not amenable to the automatic lifting. They are all presented systematically in the Reference Manual, and below we give only a brief overview.

Binary scalar operators are always infix, and can be left-associative, right-associative and non-associative. If operator `@` is left-associative, then `X@Y@Z` is the same as `(X@Y)@Z`, and if it is right-associative, then `X@Y@Z` is the same as `X@(Y@Z)`. If `@` is non-associative, the form `X@Y@Z` is syntactically invalid. Unary scalar operators can be prefix and postfix.

Scalar arithmetic operators

The next table presents the arithmetic operators, which take number operands and produce a number result:

Operator	Usage	Associativity	Description
		Additive operators	
`+`	$E + E'$	Left	Addition
`-`	$E - E'$	Left	Subtraction
		Multiplicative operators	
`*`	$E * E'$	Left	Multiplication
`/`	E / E'	Left	Division
`div`	E `div` E'	None	Integer division
`mod`	E `mod` E'	None	Remainder
		Power operators	
`^`	E ^ E'	Right	Exponentiation
		Unary operators	
`-`	$-E$	Prefix	Sign inversion
`+`	$+E$	Prefix	Sign preservation

The unary operators have the highest priority, then the power operators, then the multiplicative operators, and finally the additive operators.

The operands to the scalar arithmetic operators can be any `number`. If at least one operand is `null`, the result is also `null`.

Scalar string operators

There is a string concatenation operator:

Operator	Usage	Associativity	Description
`||`	*E* `||` *E'*	Left	String concatenation

VTL does not distinguish between `null` and the empty string `""`.

Scalar Boolean operators

Scalar Boolean operators correspond to the logical connectives `or`, `xor`, `and`, and `not`. They take Boolean operands and return a Boolean value. Unary `not` has the highest priority, then the multiplicative operator `and`, and finally the two additive operators `or` and `xor`.

Operator	Usage	Associativity	Description
Additive operators			
`or`	*E* `or` *E'*	Right	Logical disjunction
`xor`	*E* `xor` *E'*	Right	Logical exclusive disjunction
Multiplicative operators			
`and`	*E* `and` *E'*	Left	Logical conjunction
Unary operators			
`not`	`not` *E*	Prefix	Logical negation

The treatment of nulls is the following:

X	`not` X	Y	X `and` Y	X `or` Y	X `xor` Y
`true`	`false`	`true`	`true`	`true`	`false`
		`false`	`false`	`true`	`true`
		`null`	`null`	`true`	`null`
`false`	`true`	`true`	`false`	`true`	`true`

		`false`	`false`	`false`	`false`
		`null`	`false`	`null`	`null`
`null`	`null`	`true`	`null`	`true`	`null`
		`false`	`false`	`null`	`null`
		`null`	`null`	`null`	`null`

Scalar relational and test operators

Operator	Usage	Associativity	Description
Binary operators			
`=`	$E = E'$	None	Value equality. E and E' are the same
`<>`	$E <> E'$	None	E and E' are not the same
`<`	$E < E'$	None	E is smaller than E'
`<=`	$E <= E'$	None	E is smaller than or equal to E'
`>`	$E > E'$	None	E is greater than E'
`>=`	$E >= E'$	None	E is greater than or equal to E'
`not =`	E `not` $= E'$	None	Equivalent to $E <> E'$
`not <>`	E `not` $<> E'$	None	Equivalent to $E = E'$
`not <`	E `not` $< E'$	None	Equivalent to $E >= E'$
`not <=`	E `not` $<= E'$	None	Equivalent to $E > E'$
`not >`	E `not` $> E'$	None	Equivalent to $E <= E'$
`not >=`	E `not` $>= E'$	None	Equivalent to $E < E'$
Ternary operators			
`between`	E `between` E' `and` E''	None	Equivalent to `(`$E'<=E$ `and` $E<=E''$`)`
`not between`	E `not between` E' `and` E''	None	Equivalent to `(`$E'>E$ `or` $E>E''$`)`
Unary operators			
`is null`	E `is null`	Postfix	Returns `true` iff E is `null`. Does not distinguish between empty strings and

			nulls.
`is not null`	*E* `is not null`	Postfix	Equivalent to `not(`*E* `is null)`

The equality and inequality operators (=, <>, and their negated variants) can take any scalar values as operands. Scalar relational operators (<, <=, >, >=, `between` and their negated variants) only take numeric operands. If at least one operand to a relational operator is `null`, the result is also `null`.

Unary test operators `is null` and `is not null` test whether the operand is (or is not null) and return the corresponding Boolean value as a result.

Scalar Functions

In VTL 1.1, scalar functions (i.e., functions whose arguments are only scalar and that return scalar as a result) can also be automatically lifted to dataset/scalar and dataset levels, similarly to the unary and binary operators. For instance, `pow(X,N)` computes `N`-th power of number `X`, and `log(X)` computes the natural logarithm of `X`. When one or more arguments to such a function are datasets, they get automatically lifted. For instance, `pow(D1.X,D2.N)` joins `D1` and `D2`, and then for each matched row computes the scalar power, taking the measure `X` from `D1` as the base and measure `N` from `D2` as the exponent, and the result is a joint dataset with an additional column holding the result.

Join Expressions

VTL 1.1 introduces the join expressions as the base mechanism for combining and manipulating datasets, including the lifting of the scalar operators and functions to the dataset/scalar and dataset levels.. The general join expression syntax has the form:

`[`*JOIN*`]` `{`*BODY*`}`

where *JOIN* is one of several join specifications described below, and *BODY* is a list of zero or more join expression statements that perform data filtering, computation, manipulation, grouping and ordering, also described in more detail in the text that follows. The start of the join expression is distinguished by the open square bracket ("`[`").

Join Specifications

The join specification is one of the following:

- *d* – a single dataset variable. In this case we have **dataset traversal** (no join is performed). *BODY* is executed for each record in *d*. Inside *BODY*, *d* refers to the current record in dataset *d*.
- $d_1, d_2, \ldots, d_n$ – where $n>1$, performs an **outer join** of datasets held in dataset variables $d_1, d_2, \ldots, d_n$. These datasets must be joinable: for some index *j*, the set of identifier components in d_j must include identifiers from all other datasets; d_j is called the pivot dataset. Then, d_j is joined using a full outer join with each of datasets d_i ($i<>j$) on shared

identifier components. Inside *BODY*, each of $d_1,d_2,\ldots,d_n$ refers to the matched record from the respective dataset.

- d_1 `outer` d_2`,`…`,` d_n – where $n>1$, is synonymous to the previous case d_1`,` d_2`,`…`,` d_n.
- d_1 `inner` d_2`,`…`,` d_n – where $n>1$, performs an **inner join** of datasets held in dataset variables $d_1,d_2,\ldots,d_n$. As in the outer join case, the datasets must be joinable: for some index *j*, the set of identifier components in d_j must include identifiers from all other datasets; d_j is called the pivot dataset.. Then, d_j is joined using an inner join with each of datasets d_i ($i<>j$) on shared identifier components. Inside *BODY*, each of $d_1,d_2,\ldots,d_n$ refers to the matched record from the respective dataset.
- d_1 `cross` d_2`,`…`,` d_n – where $n>1$, performs a **cross join** (or a Cartesian product) of datasets held in dataset variables $d_1,d_2,\ldots,d_n$. All combinations of records are processed in *BODY*, and each of $d_1,d_2,\ldots,d_n$ refers to the matched record from the respective dataset.

The meaning of the inner and the outer join is the same as the meaning of `INNER JOIN` and `FULL OUTER JOIN` constructs, respectively, in the SQL-92 standard. In the cross join case, *BODY* of the join expression typically filters out record combinations that do not fit some logical condition.

It is possible for two or more dataset variables involved in a join to refer to (i.e., act as aliases for) the same dataset. Inner and outer joins recognize dataset aliases, and automatically simplify the join structure to ensure that each dataset variable refers to a distinct dataset, while the aliases can still be used in the join body and refer to the same matched record from the original dataset. This is an automatic process that is transparent to the user. Indeed, aliases can be safely removed in an inner or outer join because joining a dataset with itself on the same set of identifier components always matches each record with itself.

However, in a cross join, each dataset variable is used, whether or not two or more of them refer to a same dataset. This allows matching of two or more records from the same dataset using custom filter criteria, and is instrumental in implementing multiple-record (combinatorial, first-order, or "diagonal") validation rules.

Functional Integrity

The VTL information model requires of each dataset a functional dependency between the identifier components and all other components. If we look at a dataset as a tabular structure with a finite number of columns (which correspond to components) and rows (which correspond to individual records), this translates into the following *functional integrity* requirements:

- A dataset can have an arbitrary number of identifier, measure and attribute columns. Each column has a distinct name in the dataset, and a scalar data type.
- All `null` values in string columns are implicitly converted into the empty string, and are not seen as `null`s in the points below.
- If a dataset has no identifier columns, but it has at least one measure or attribute column, it must have exactly one row. A dataset that has no columns whatsoever

cannot have any rows. The points below apply only to datasets with one or more identifier components.

- No identifier column can have a `null` value in any dataset row.
- The combination of identifier column values in a dataset row is called the key. Two or more rows in the same dataset cannot have the same key.
- When a measure or attribute column has value `null` in a dataset row, it is considered undefined for that row's key.

The join expressions not only expect the input datasets to be functionally integral, but are engineered in a way that ensures functional integrity of the result. The key to this is the behaviour of join clauses and elements of *BODY*, explained below. Therefore, any construct built with the join expressions, including the lifting of the scalar operators and functions to the dataset/scalar and dataset levels, respects functional integrity by construction.

Successive Dataset Transformations

To explain the meaning of the join expressions, we can logically view it as a series of successive dataset transformations:

First, the join specification that starts a join expression (traversal, inner, outer, or cross join) creates by itself the initial "joined" dataset:

- ○ For a ***dataset traversal***, the initial dataset is identical to the traversed dataset.
- For For ***inner and outer joins***, the initial working record consists of the identifier components from the pivot dataset matching record.
- For ***cross join***, the initial working record consists of identifier components from all input datasets: identifier component *X* from input d_i appears under name d_i_X (name of the dataset variable d_i plus an underscore, plus the name of the component *X*). To avoid possible ambiguities, in the cross join case the names of input dataset variables cannot contain an underscore.

Second, the first join expression statement in *BODY* (if any) operates on this initial dataset and produces a resulting dataset. which is fed as input to the next statement in *BODY*, etc. The dataset which is the result of the last statement in *BODY* is the result of the entire join expression.

It should be noted that this is a logical view on the semantics of the join specification and the statements in *BODY*, which makes it easy to explain and understand. In reality, having each statement making a separate pass through its input dataset would not be efficient. Indeed, it is often the case that all *BODY* statements can be executed in a single pass (e.g., a single SQL query) through the joined datasets.

Kinds of Body Statements

The element *BODY* in a join expression consists of zero or more ***join expression statements*** that define the processing steps applied to the (joined) input datasets inside the join expression. These statements can be divided in two main groups:

- ***Record-level statements*** process each individual record of the statement's input dataset, by adding or updating columns, computing temporary values (i.e., local variables), or deciding whether to keep or discard a record based on a filter condition.
- ***Transposition statements***, which unfold an identifier component (a measure dimension) from several records from its input dataset into a single output record, or perform a symmetric folding operation. The measure dimension breakdown for folding and unfolding is either given explicitly as a part of the transposition statement, or by reference to an externally defined hierarchy.

Record-Level Statements

Several record-level statements use *scalar expressions in the column mode*. These are expressions that evaluate to a scalar value, but differ from normal scalar expressions (in the general mode) in the interpretation of identifiers. In the column mode expressions, the identifiers (that are not followed by an open parenthesis or a `.`) refer to components in the working record which is the input to the statement, and not to variables. To refer to a variable, one has to prefix its name with a dollar sign.

Explicit component computations

These statements compute the value of a component in the working record.

Form	Description
$X := E$	Computing new/updated measure
`measure` $X := E$	Same as the previous
`attribute` $X := E$	Computing new/updated attribute
`identifier` $X := E$	Computing new/updated identifier

In the above table, *X* is a component name (an identifier) for the newly computed component, and *E* is a scalar expression in the column mode. By default, if an explicit role keyword (`measure`, `attribute`, or `identifier`) is omitted, role `measure` is assumed.

An explicit component computation adds to the working record a component named *X* with a given role and value specified by *E*. The working record may already contain a measure or attribute component named *X*, which can be used in *E*, but is replaced with the newly computed *X* (which may have a different role and/or type). An error is raised if the working record has an identifier component named *X*.

The type of component *X* in the resulting working record is the type of expression *E*.

E is not a string expression and it evaluates to `null`.Example 1:

```
[D] {
  Total := Men + Women
  WomenRatio := Women / Total
  MenRatio := 1.0 - WomenRatio
  attribute ObsStatus := ObsStatus || "A"
}
```

Example 2:

```
[D] {
  Population := Population * 1.01
  attribute ObsStatus := ObsStatus || "I"
}
```

Example 3:

```
[D1, D2] {
  Population := D1.Population + D2.Population
  attribute ObsStatus := D1.ObsStatus || D2.ObsStatus
}
```

Implicit component computations

The implicit component computation statements compute the value of a component if it is not already present in the working record.

Form	Description
`implicit` $X := E$	Computing implicit measure
`implicit measure` $X := E$	Same as the previous
`implicit attribute` $X := E$	Computing implicit attribute
`implicit identifier` $X := E$	Computing implicit identifier

In the above table, *X* is a component name (an identifier), and *E* is a scalar expression in the column mode. By default, if an explicit role keyword (`measure`, `attribute`, or `identifier`) is omitted, role `measure` is assumed.

The implicit component computation statements behave similarly like their explicit counterparts (without keyword `implicit`), but they are executed only if the working record does not already have a component named *X*. An error is raised if there is already a component named *X*, but with a different role.

The type of component *X* in the resulting working record is the type of expression *E*.

E is a non-string expression that evaluates to `null`. Example 1:

```
[D] {
  implicit attribute ObsStatus := ""
}
```

Example 2:

```
[D1, D2] {
  Population := D1.Population + D2.Population
  attribute ObsStatus := D1.ObsStatus || D2.ObsStatus
  implicit identifier RefArea := "EU"
}
```

Computing local variables

Local variables store a value for the remainder of the record-level statements in *BODY*.

Form	Description
$\$X := E$	Computing a local variable

In the table above, X is an identifier, used as a variable name, and E is a scalar expression in the column mode.

This statement is useful for computing a value and storing the result temporarily for easier reference, without making it appear in the result.

Example:

```
[D] {
  $Total := Men + Women + Children
  WomenRatio := Women / $Total
  MenRatio := Men / $Total
  ChildrenRatio := 1.0 – WomenRatio - MenRatio
}
```

Filtering records

The filtering statement decides whether to keep the working record in the result or to omit it.

Form	Description
`filter` E	Permit only records satisfying condition E

In this statement, E is a Boolean expression in the column mode.

If at runtime E does not evaluate to `true`, no further record-level statements are executed, and the working record is discarded.

Example 1:

```
[D] {
  $Total := Men + Women + Children
  WomenRatio := Women / $Total
  MenRatio := Men / $Total
  filter MenRatio + WomenRatio >= 0.6 /* Treat only these cases. */
  ChildrenRatio := 1.0 – WomenRatio - MenRatio
}
```

Example 2:

```
[D1, D2] {
  filter D1.Pop is not null or D2.Pop is not null
        /* At least one of D1.Pop and D2.Pop must be defined. */
  Pop := D1.Pop + D2.Pop
}
```

Example 3:

```
[D1 cross D2] {
  filter D1.Pop < D2.Pop    /* Custom join condition. */
  Ratio := D1.Pop / D2.Pop
}
```

Function application to components of the working record

These statements transform components of the working record by applying a function to them.

Form	Description
`apply` F	Apply function to measures of the matching type
`apply` F `to attributes`	Apply function to attributes of the matching type
`apply` F `to measures and attributes`	Apply function to measures and attributes of the matching type

Here, F is a function that takes one argument of some scalar type t and returns a result of some scalar type T. The first form transforms value of each measure X from the working record whose type is compatible with t to value $F(X)$ of type T in the resulting working record.

The statement forms that include `` `to attributes' `` and `` `to measures and attributes' `` apply function F to components with the respective roles, not just to measures as in the first form.

Example:

```
[D] {
  apply _*1000             /* Multiplies all numeric measures by 1000. */
  apply _&"x" to attributes /* Adds "x" to all string attributes. */
}
```

Function application to components of the matched input records

These statements combine components from the matched records of the input datasets by applying a function to their values and adding the result to the working record.

Form	Description
`apply` F `to` $d_{k1}, \ldots, d_{km}$	Apply function to measures from $d_{k1}, \ldots, d_{km}$ with same names and matching types
`apply` F `to attributes in` $d_{k1}, \ldots, d_{km}$	Apply function to attributes from $d_{k1}, \ldots, d_{km}$ with same names and matching types
`apply` F `to measures and attributes in` $d_{k1}, \ldots, d_{km}$	Apply function to measures and attributes from $d_{k1}, \ldots, d_{km}$ with same names and matching types

Here, F is a function that takes $m>0$ arguments of the corresponding scalar types $t_1,\ldots,t_m$, and returns a scalar result of type T. $d_{k1},\ldots,d_{km}$ is a subset of the input dataset variables from *JOIN* that represent the matched records in *BODY*.

The first form of the statement looks for the same-name measure components that appear in each of $d_{k1},\ldots,d_{km}$ and whose respective types are compatible with $t_1,\ldots,t_m$. For each such shared component named X, a measure component X of type T is added (or replaced) in the resulting working record, with value $F(d_{k1}.X,\ldots,d_{km}.X)$.

The forms with `` `to attributes in' `` and `` `to measures and attributes in' `` apply F to the components of the respective role, not just to measures in $d_{k1},\ldots,d_{km}$.

Example:

```
[D1,D2]                                                                     {
  apply 0.3*_+0.7*_ to D1, D2          /* Weighted sum of numeric measures */
  apply _&_ to attributes in D1, D2   /* Concatenating string attributes */
  apply _or_ to attributes in D1, D2 /* Disjunction of Boolean attribs */
}
```

Component renaming statements

These statements change names of one or more components in the working record simultaneously.

Form	Description
`rename` X_1 `to` Y_1, X_2 `to` Y_2, …, X_n `to` Y_n	Simultaneously rename *X*s to *Y*s
`rename` X_1 `->` Y_1, X_2 `->` Y_2, …, X_n `->` Y_n	Same as the above

Each X_i and Y_i (i=1,…,*n*, *n*>0) in the table above an identifier specifying a column name, optionally preceded with a role (`identifier`, `measure`, or `attribute`). Identifiers in X_1, …, X_n must be mutually distinct, as well as those in Y_1, …, Y_n.

Each X_i must exist in the working record. If X_i does not specify the source role, the actual role of the component with that name is in the working record is used. If Y_i does not specify the target role, the source role is used.

The renaming statement (between { }) is performed simultaneously as a whole, which makes column name and role swapping and cycling possible with a single statement. If the working record has a measure or attribute whose name is in Y_1, …, Y_n, but not in X_1, …, X_n, that component is replaced by the renamed component. However, an error is raised if such component is an identifier.

It is also an error to change the role of an identifier component using `rename`.

Example 1:

```
[D] {
  rename A to B, B to A /* Swap component names */
}
```

Example 2:

```
[D] {
  rename identifier Geo to RefArea, /* Rename identifier Geo */
         Age to identifier Age,     /* Make Age an identifier */
         attribute ObsStatus to measure Status,
                                    /* Convert attribute to a measure */
         Z to attribute Z           /* Error if Z is an identifier */
}
```

Component filtering statements

These statements keep or drop the specified components in the working record.

Form	Description
`keep` X_1, …, X_n	Keep measures or attributes in the working record

`drop` $X_1, \ldots, X_n$	Drop measures or attributes from the working record

Each X_i (i=1,…,n, n>0) is an identifier giving the column name, optionally preceded with a role `measure` or `attribute`.

Statement `keep` keeps in the working record only the measures and attributes given by X_1, …, X_n, which must all exist in the working record. Identifiers are not affected.

Statement `drop` drops from the working record those measures and attributes given by X_1, …, X_n that exist in the working record. An error is raised if any of X_1, …, X_n is an identifier.

Example 1:

```
[D] {
  $Total := Men + Women + Children
  WomenRatio := Women / $Total
  MenRatio := Men / $Total
  ChildrenRatio := 1.0 - WomenRatio - MenRatio
  keep WomenRatio, MenRatio, ChildrenRatio
                   /* Keep only these measures (no attributes kept)  */

}
```

Example 2:

```
[D] {
  $Total := Men + Women + Children
  WomenRatio := Women / $Total
  MenRatio := Men / $Total
  ChildrenRatio := 1.0 - WomenRatio - MenRatio
  drop Women, Men, Children
       /* Keep all measures and attributes except these three  */
}
```

Transposition Statements

The transposition statements can be used instead of the aggregation statements. These statements also operate on all records resulting from the join and the record-level statements, but instead of aggregating, they transpose columns from several input records into a single output record and back.

Form	Description
`unfold` X, Y `to` $B_1, \ldots, B_n$	Unfold identifier X and measure Y into columns B_1, …, B_n (n>0).
`unfold` X, Y `using` H	Unfold identifier X and measure Y using hierarchy definition H.
`fold` $B_1, \ldots, B_n$ `to` X, Y	Fold columns B_1, …, B_n (n>0) into a new identifier X and measure Y.
`fold using` H `to` X, Y	Fold a new identifier X using hierarchy definition H and measure Y.

In the above table, X is the name of an identifier column

Each B_i in breakdown $B_1, \ldots, B_n$ is either a base element (an identifier), or a computed element of the form $Z=C_1+ \ldots + C_m$, where Z is an identifier, and $C_1, \ldots, C_m$ ($m>0$) are other breakdown elements (base or computed) that go into Z. Circular dependencies between computed breakdown elements are not allowed. Each breakdown element B_i has the base set U_i of base elements that it "covers". If B_i is a base breakdown element, its $U_i=\{B_i\}$. If B_i is a computed breakdown element of the form $Z=C_1+ \ldots + C_m$, its elementary set is the union of the base sets of $C_1, \ldots, C_m$.

The breakdown structure $B_1, \ldots, B_n$ can be specified explicitly in the statement, or it can be defined in a hierarchy object H defined elsewhere (i.e., in metadata). In the text that follows we shall assume that in the latter case the actual structure $B_1, \ldots, B_n$ has been retrieved from H.

The `unfold` statement divides the input dataset with a string identifier component X (the measure dimension) and a numeric measure component Y into groups of records sharing the value of all identifiers other than X.

Each input group is then transformed into a single output record that has:

- A copy of all identifier components from the input group except X.
- Numeric measure columns $B_1, \ldots, B_n$ instead of the single measure column Y. For each B_i (i=1..n), the value of the measure column named B_i in the output record is the sum of Y in the group records where the value of X belongs to the base set of B_i (as a set of string literals).
- All other measure and attribute components, whose value is taken as the maximum in the group.

The `fold` statement works in the opposite direction: for each input record it generates a group of output records, with one output record for each breakdown element B_i (i=1..n) where the value of component B_i is not `null`, consisting of:

- A copy of all identifier components from the input record.
- A new string identifier component named X with value equal to B_i (as a string literal).
- A new numeric measure component Y with value equal to the value of B_i in the input record.
- A copy of all attribute and measure components (other than $B_1, \ldots, B_n$) taken from the input record.

Example 1:

Suppose *BeNeLuxPop* is the following dataset:

Year	**Geo**	**Pop**	***Status***
2015	BE	11,324	A
2015	NE	16,948	
2015	LU	563	AP

Then the result of the join expression:

```
[BeNeLuxPop] {
  unfold Geo, Pop to BE, NE, LU, Total = BE + NE + LU
}
```

is:

<u>**Year**</u>	**BE**	**NE**	**LU**	**Total**	***Status***
2015	11,324	16,948	563	28,835	AP

Example 2:

If D is the result of the previous example, then the following join expression:

```
[D] {
  fold BE, NE, LU, Total = BE + NE + LU to Geo, Pop
}
```

gives the result:

<u>**Year**</u>	<u>**Geo**</u>	**Pop**	***Status***
2015	BE	11,324	AP
2015	NE	16,948	AP
2015	LU	563	AP
2015	Total	28,835	AP

Note that this result is very similar to the original input, except for a couple of differences that illustrate some important aspect of the `fold` and `unfold` statements:

- The computed breakdown element Total appears in the result, while it was not present in the original input dataset *BeNeLuxPop*. If this is undesirable, the `fold` statement should use only the base (not computed) breakdown components BE, NE, and LU.
- In the `fold` statement, the computed breakdown elements, such as *Total*, are not computed, but are treated in the same way as the base breakdown elements (*BE*, *NE*, and *LU*).
- While the *Status* attribute varies in the original input dataset *BeNeLuxPop*, it is uniformly equal to "AP" in all result rows. The reason for this is that unfolding entails a loss of information for attributes like *Status*, where it takes the maximum for the whole group of records where *Year*=2015. Folding, on the other hand, does not entail any loss of information (it can, in fact, create additional information, as seen in the previous point).

Lifting Scalar Operators and Functions With Join Expressions

We now turn to the issue of lifting the scalar operators and functions to the dataset/scalar and dataset level using the join expressions. This lifting is not something a VTL programmer needs to do manually -- it is done automatically under the hood by the compiler. However, it is important for both the programmers and language implementers to understand clearly how the lifting works in order to ensure the correct behaviour.

Liftable Expressions

As a preliminary, we need to define what is a "liftable" expression. For an expression to be liftable, it has to satisfy certain structural and typing constraints. The typing constraints are important because the syntactic form of an expression does not provide sufficient information for deciding whether an expression needs to be lifted and how. For instance, `A+B` may be a scalar or a dataset expression, depending on the types of A and B. For what we need here, we shall take a simplified look at the type analysis:

- The typing of an expression is decided inductively, or bottom-up: from the operation or function argument types to the type of the operator application or function call.
- After determining that the type of an expression E is t, we shall be making simple assertions, such as: "t is a scalar type (i.e., E is a scalar expression)", or "t is a dataset type (i.e., E is a dataset expression)".

Intuitively, we can define a scalar-based expression as an expression that uses only scalar operators and functions on arguments that are scalar variables or literals, datasets and their components, or scalar-based subexpressions. A liftable expression is then a scalar-based expression that returns a dataset, because one or more of the arguments to a scalar operator or function is given as a dataset. Or, in other words, only a scalar-based expression can be liftable, but the property of being liftable is stronger.

More formally, we say that an expression of the form $f(E_1, ..., E_n)$, $n>0$, is a scalar-based expression if:

- f accepts n scalar arguments and returns a scalar result
- Each argument E_i ($i=1..n$) is one of the following:
  - a scalar variable or a numeric, string or Boolean literal [weak argument]
  - a dataset variable [strong argument]
  - an expression of the form $D.X$ where D is a dataset variable, and X is a component identifier [strong argument]
  - a scalar-based expression [strong argument exactly when E is liftable]
- If at least one argument is strong, then the scalar-based expression $f(E_1, ..., E_n)$ is liftable.

We wrote $f(E_1, ..., E_n)$ to denote both a call to function f and an application of an n-ary operator (prefix, infix, or postfix) to its arguments.

Example 1:

Expressions `-X`, `log(X)`, and `X*Y`, where *X* and *Y* are scalar variables, are all scalar-based, but they are not liftable, because they do not use any dataset. However, expression `D.X*log(D.X)`, where *D* is a dataset variable, is both scalar-based and liftable.

Example 2

Expression:

```
D1^2+2*D1*D2+D2^2
```

where *D1* and *D2* are dataset variables, is liftable, because it uses these two dataset variables as arguments to basically scalar operators `+`, `*`, and `^`.

Component Selection And Lifting Scheme

A liftable expression *E* must contain one or more dataset references of the form *D* or dataset component references of the form *D.X*, where *D* is a dataset variable. The shape of these references significantly affects the computation that is performed.

In the sub-sections that follow we cover all dataset and dataset component reference cases that may occur, and give representative examples of the lifting scheme.

Operating on All Shared Components

The first case is when *E* contains only dataset references (*D*), but no dataset component references (*D.X*). In this case, the computation is performed on all shared measure components, i.e., the measure components with the same name and type that appear in all referenced datasets. The resulting dataset uses these shared measure components to hold the result.

Example 1:

As a simple example, `D1+D2`, where *D1* and *D2* are dataset variables with numeric measure components *A* and *B*, will create a result with measure components *A* and *B* whose value is the sum of *A*s and *B*s from *D1* and *D2*. The lifting is then done using a join expression and `apply`:

```
[D1,D2] {
  apply _+_ to D1, D2
}
```

Example 2:

Expression:

```
D1^2+2*D1*D2+D2^2
```

is lifted with:

```
[D1,D2] {
  apply \x,y{x^2+2*x*y+y^2} to D1, D2
}
```

In this example, we had to explicitly name the arguments *x* and *y* in the function, because *D1* and *D2* appear more than once in the original expression.

Operating on Single Named Component

The second case is when *E* contains one or more dataset component references of the form *D.X* where *D* may vary, but *X* is a single component name. In this case, we only operate on that single component *X* in all referenced datasets, and the result contains a single measure component *X* holding the result. All dataset references of the form *D* in *E* are implicitly rewritten into *D.X*. The fixed component *X* must not be null in at least one referenced dataset.

Example 1:

Expression:

```
D1.Pop + D2
```

where *D1* and *D2* are dataset variables, operates on a single component *Pop*. It is therefore equivalent to:

```
D1.Pop + D2.Pop
```

And is lifted as:

```
[D1,D2] {
  filter D1.Pop is not null or D2.Pop is not null
  Pop := D1.Pop + D2.Pop
}
```

The result contains a single measure component named *Pop*.

Example 2:

Expression:

```
D1.Pop * 1.02
```

also uses the single named component *Pop*. It is lifted as follows:

```
[D1] {
  filter D1.Pop is not null
  Pop := D1.Pop * 1.02
}
```

Note that in this example the join expression traverses a single dataset *D1*, and therefore all other measures and attributes are kept unchanged in the result.

Operating on Multiple Named Components

Finally, we may have a case where *E* contains two or more dataset component references of the form *D.X* where *X* is not always the same. This case was illegal in VTL 1.0 because of the rule that differently named components from different datasets cannot mix in a computation. The experience indicates that his requirement can sometimes be too strict, and may force the programmer to frequently explicitly rename components in order to be able to compute on them.

That is why VTL 1.1 allows mixing two or more differently named dataset components in a single liftable expression *E*, provided that *E* contains no dataset references of the form *D* (i.e., only contains dataset component references of the form *D.X*). The resulting dataset contains a single measure component named *Value* holding the result of the computation.

Example 1:

Expression:

```
D1.Pop + D2.Population + D3.Residents + D4.Inhabitants
```

is lifted as follows:

```
[D1, D2, D3, D4] {
  filter D1.Pop is not null or D2.Population is not null
         or D3.Residents is not null or D4.Inhabitants is not null
  Value := D1.Pop * D2.Population + D3.Residents + D4.Inhabitants
}
```

Example 2:

Expression:

```
D1.Pop between D2.Min and D2.Max
```

is lifted as follows:

```
[D1, D2] {
  filter D1.Pop is not null or D2.Min is not null
         or D2.Max is not null
  Value := D1.Pop between D2.Min and D2.Max
}
```

The resulting measure *Value* is Boolean.

Example 2:

Expression:

```
(D1.Pop between D2.Min and D2.Max)[Value->InRange]
```

is lifted as follows:

```
[D1, D2] {
  filter D1.Pop is not null or D2.Min is not null
         or D2.Max is not null
  Value := D1.Pop between D2.Min and D2.Max
  rename Value -> InRange
}
```

The resulting Boolean measure generically named *Value* has been renamed to more domain-specific *InRange*.

Allowing Non-Scalar-Based Subexpressions

The approach for lifting expressions built with scalar operators and functions to the database/scalar and database levels explained above restricts the structure of such expressions to scalar-based expressions defined above. This limitation can sometimes be too strict. For instance, expression:

```
D1.Total + size(D2)
```

where *D1* and *D2* are dataset operations, and *size* is a function that returns the number of records in a dataset, is not scalar-based (and therefore misses the precondition to be lifted) because *size* does not take a scalar, but a dataset argument. Therefore, in this expression *D2* should be treated differently than *D1*: we do not need to join these two datasets, we just first need to count rows in *D2*, remember the result and then use it in the main expression.

Another example is:

```
union(D1, D2) * 1.02
```

This is also a valid expression, where we increase all numeric measures in the union of two datasets *D1* and *D2* by 2%. But it is not a scalar-based expression (and therefore not a liftable one), because *union* is not a scalar function. Still it is clear that first we have to make a union of *D1* and *D2* , and then multiply the result with 1.02.

These two examples hint at a general solution: we can often transform a non-scalar-based expression into a scalar-based one by proceeding step-by-step.

Let us first take E to be an expression that contains some sub-expression A. It is clear that E is equivalent to a VTL block:

```
{
   $V$  :=  $A$
   $E[V/A]$
}
```

where V is a variable name that does not appear in E, and $E[V/A]$ is a copy of E where V replaces A.

This scheme can be automatically applied to all scalar or dataset subexpressions A_1, ..., A_n of E that are not scalar-based. As a result, we transform E into the form:

```
{
   $V_1$  :=  $A_1$  /* $V_1$ does not appear in $E$ */
   $V_2$  :=  $A_2$  /* $V_2$ does not appear in $E$ */
  ...
   $V_n$  :=  $A_n$  /* $V_n$ does not appear in $E$ */
   $E[V/A]$        /* Becomes liftable expression! */
}
```

This transformation can be automatically done by the compiler.

Example 1:

Expression:

```
D1.Total + size(D2)
```

becomes:

```
{
  V := size(D2)
  D1.Total + V /* liftable */
}
```

which after lifting becomes:

```
{
  V := size(D2)
  [D1] {
    filter D1.Total is not null
    Total := D1.Total + V
  }
}
```

Example 2:

Expression:

```
union(D1, D2) * 1.02
```

becomes:

```
{
  V := union(D1, D2)
  V * 1.02 /* liftable */
}
```

which after lifting becomes:

```
{
  V := union(D1, D2)
  [V] {
    apply _*1.02
  }
}
```

Expressing Validation Rules With Join Expressions

In the previous sections we have shown how the VTL 1.1 join expressions can be used for lifting of basically scalar expressions and functions to the dataset/scalar and dataset levels. This lifting is performed automatically and transparently by the compiler, and provides a well-defined semantics for the lifted constructs. We can therefore think about the join expressions as a "core" mechanism for expressing the behaviour of higher-level dataset operations.

The same approach can be used for expressing the behaviour of some important classes of validation rules:

- ***Horizontal rules*** -- these rules check validity of individual records (or rows) in a dataset. For the sake of simplicity, let us say that each horizontal rule has a condition *SCOPE_COND* that selects records to which the validation rule needs to be applied, a condition *VALID_COND* that defines when a row is valid, and a string *RULE_CODE* that is inserted in the result column *ERR_CODE* if the validation fails on a record. The validation of a dataset *D* using a horizontal rule is then equivalent to:

    -
      ```
      [D] {
        implicit attribute ERR_CODE := ""
        filter SCOPE_COND
        attribute RULE := VALID_COND
        attribute ERR_CODE :=
          if RULE then ERR_CODE else RULE_CODE
      }
      ```
- ***Vertical rules*** -- these rules apply to values of some measure component *Y* that are stacked "vertically" one under another in each group of records, so that each value of *Y* corresponds to a particular code of some measurement dimension *X*. The breakdown of *X* to individual codes is typically given explicitly in a vertical rule as B_1, ..., B_n. Again, for the sake of simplicity, let us say that each vertical rule has a condition *SCOPE_COND*

that selects groups of records to which it applies, a condition *VALID_COND* that defines when a row is valid, and a string *RULE_CODE* inserted in the result column *ERR_CODE* if the validation fails on a record. The validation of a dataset *D* using a vertical rule is then equivalent to:

-
  ```
  {
    U := [D] { unfold X, Y to $B_1, ..., B_n$ }
    [U] {
      implicit attribute ERR_CODE := ""
      filter SCOPE_COND
      attibute RULE := VALID_COND
      attribute ERR_CODE :=
        if RULE then ERR_CODE else RULE_CODE
    }
  }
  ```

- ***First-order or combination rules*** -- these rules apply to combination of records from two or more datasets $D_1, ..., D_n$ (the same dataset variable can be repeated several times). The criteria for matching these records is specified as *MATCH_COND* , and we here take the other (simplified) assumptions about *VALID_COND*, *RULE*, and *RULE_CODE* as in the examples of the horizontal and vertical rules above. Then, the validation of a dataset *D* using this kind of rules is then equivalent to:

  -
    ```
    [$D_1$ cross $D_2, ..., D_n$] {
      implicit attribute ERR_CODE := ""
      filter MATCH_COND
      attribute RULE := VALID_COND
      attribute ERR_CODE :=
        if RULE then ERR_CODE else RULE_CODE
    }
    ```

The above examples were simplified (among other things) because they refer to a single rule, while VTL 1.1 allows more powerful rule sets to be defined. However, at this point it should be evident that there are ways for expressing rule sets using the same kind of constructs. Suppose, for instance, we have a horizontal rule set consisting of three rules, *RULE1, RULE2* and *RULE3.* The translation would look like this:

```
[D] {
  implicit attribute ERR_CODE := ""
  filter SCOPE_COND1 or SCOPE_COND2 or SCOPE_COND3
  $ERR_CODE := ERR_CODE
  $RULE1 := not (SCOPE_COND1) or VALID_COND1
  $ERR_CODE :=
    if $RULE1 then $ERR_CODE else paste($ERR_CODE, RULE_CODE1, ",")
  $RULE2 := not (SCOPE_COND2) or VALID_COND2
  $ERR_CODE :=
    if $RULE2 then $ERR_CODE else paste($ERR_CODE, RULE_CODE2, ",")
```

```
  $RULE3 := not (SCOPE_COND3) or VALID_COND3
  $ERR_CODE :=
    if $RULE3 then $ERR_CODE else paste($ERR_CODE, RULE_CODE3, ",")
  attribute RULESET := $RULE1 and $RULE2 and $RULE3
  attribute ERR_CODE := $ERR_CODE
}
```

This construct would check all three horizontal rules in the rule set in a single traversal of *D*, and would look only on records where at least one rule is applicable. It would create the attribute *ERR_CODE* if it did not exist, and would add to it (as a comma-separated list) error codes of all failed rules. The result would also have an attribute column *RULESET* (named after the rule set) which holds Boolean true if the `record` has passed all three rules, or `false` if at least one rule has failed on the record.

VTL main assumptions

In this chapter we present some of the main assumption on which the Validation and Transformation Language bases the semantics of its Operators. These core assumptions complement the core language elements presented in the previous chapter, and they specify the general behaviour of the language, and is by default stable. The standard library of operators is presented in detail in the Reference Manual, and presents the built-in functionality that can be gradually enriched following the evolution of the user needs. Possible new functions and operators must obviously comply with the core assumptions presented here.

The main assumptions include:

- Details of operand and result types
- The general behaviour of operations on datasets
- Storage and retrieval of datasets
- The conventions for the grammar of the language

The main assumptions are explained in the following sections.

Details of operand and result types

The Data types of the VTL

As explained in the previous chapter, the type system of VTL 1.1 presents an outline of a type system, which is able to characterize all kind of objects that are used as an input, an intermediate result or auxiliary parameter, or produced as the result of any expression in a VTL program.

In this section, we are concentrating on a subset of VTL types which we call the data types. Data types differ from other types in that they have a well-defined external representation, covered by the VTL Information Model (IM). Obviously, different parts of VTL programs can use or produce other objects, such as anonymous functions or tuples and collections of arbitrary objects, which are transient in nature. Such transient objects exist only in memory during the execution of a VTL program, but cannot be "materialized," i.e., they have no well-defined representation in the IM.

The VTL data types, on the other hand, correspond to various artefacts represented in the IM. They include:

- Datasets, composed of identifier, measure, and attribute components; each component contains a data of the same scalar type.
- Collections of scalar types, or of Cartesian products of scalar types, which are used to express constraints, i.e., the permissible values for one or more scalar variables.
- Modules representing dataset structure, as well as user-defined functions, types, and special objects such as validation rules.

Basic scalar data types

The ***basic (unconstrained) scalar data types*** of the language are: *string*, *number* (including *integer* and *float*), *boolean* and *date.* Their instances written directly in VTL code (i.e. the real objects of those types) are called *literals*. Thecharacteristics of the base scalar types are described in the following table.

Basic scalar data types	
string	A sequence of zero or more UNICODE characters enclosed in double quotes (“). Examples of allowed literals for this data type are: “hello”, “test”, “x”, “this is a string” and "" (the empty string). Note that in the VTL syntax the double quotes are intended to be the standard ones ("), i.e. the same character to open and close the string, even if in this document and in the Part 2 the styled double quotes may be shown. If a string literal needs to include a double quote in its contents, the quote needs to be doubled: literal "a""b" consists of three charaters: letter *a*, the double quote, and letter *b*.
number	Includes both *integer* and a *float*.
float	Floating point numbers, whose precision is compatible with or greater than the IEEE 754 quadruple precision (128 bits encoding). At least the range of floating point numbers (absolute values) between 2^{-16949} (approx. 10^{-4965}) and 2^{16384}-2^{16271} (approx. $1.1897*10^{4932}$) with 34 significant decimal digits should be representable. Alternatively, implementations may use arbitrary-precision floating point numbers. The point (.) is used as the decimal separator and must be present in the literal. Examples of allowed literals for this type are: 1.0, 234.56, 456.45; also the scientific notation is allowed: 12.23E+12, 35.2E-150, -2E10+3, 0.0. The uppercase letter "E" can be written also as the lowercase "e".
integer	The basic signed integer type. At least 64 bit in size. Alternatively, implementations may use arbitrary-precision integers. Examples of allowed literals for this type are: 2, 5, 7, 24, -14, 0.
boolean	The Boolean data type. The allowed literals are *true* and *false*.
date	A point-in-time value. The type stores the year, the month, the day, the hours the minutes and the seconds (after midnight). Date are in 24-hours format: YYYY-MM-DD HH24:MI:SS While the YYYY-MM-DD is mandatory, HH24:MI:SS is optional and, if omitted, 00:00:00 is implied. Examples of allowed literal values are: 2012-09-30, 2013-10-02, 2014-01-01 12:23:35. The format for Date literals is customizable, in the sense that specific supplementary formats may be used in implementations in addition to this one, if properly configured in the system. Alternate literals may also include the ones adopted by commercial systems for compatibility reasons, for example: date’2012-09-30’.

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

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

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

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

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

Type management and checking

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

It is envisaged that there will be “implicit upcasting” between the integer and the number data types. This means that wherever in the language it is possible to use a number, an integer or float is allowed. Obviously, the opposite is not allowed.

The VTL is strongly typed, in the sense that any operand or parameter in an operation belongs to one of the possible types.

The various VTL functions and operators have specific constraints in terms of number and types of parameters (see the corresponding sections in the Part 2).

The type of an expression is computer at compile time.

The function and operator constraints in terms of number and types of their arguments are statically checked (at compile time) so that type errors are not possible at runtime. Moreover, only type-safe upcast conversion for integers into num is performed.

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

The general behaviour of operations on datasets

General rules

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

If we assume that F is a Data Set Operator (i.e., an operation that takes some inputs and produces a dataset), that D_r is its result Data Set and that $D_{i\ (i=1,\dots n)}$ are its input Data Sets, the general form of a Transformation based on F can be written as follows:

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

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

For computing the result of this operation, F follows a number of default behaviours described here.

In general the Data Sets $D_{i\ (i=1,\dots n)}$ and consequently their Data Points may have any number of Identifier, Measure and Attribute Components, nevertheless the VTL Data Set Operators may require specific constraints on the Data Structure Components of their input Data Sets[16].

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

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

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

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

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

The Identifier Components and the Data Points default matching

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

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

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

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

D_1 = United States Data

Ref.Date	*Meas.Name*	*Meas.Value*

---

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

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, most of Data Set operations (including the sum) are also be 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.[17]

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***

---

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

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 Components common to D_1 and D_2 (in this case only the Ref.Date), then the operation will take place.

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

D_r = European Countries / Europe * 100

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

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

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

The "strict" constraint requires that the Identifier Components of the D_i $_{(i=1,\ldots n)}$ 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 $_{(i=1,\ldots n)}$ the Identifier Components of D_i are a (possibly improper) subset of those of D_k. The output Data Set D_r will have the same Identifier Components of D_k.

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

Behaviour for Measure Components

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

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

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

D_1 = Salary of Employees

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

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

D_r = Increased Salary of Employees

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

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

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

D_1 = Import & Export

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

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

D_r = 80% of Import & Export

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

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

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

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

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

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

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

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

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

D_1 = Births & Deaths of the United States

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

D_2 = Birth & Deaths of the European Union

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

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

D_r = Births & Deaths of United States + European Union

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

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

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

For example, given these two Data Sets:

D_1 (Residents in the United States)

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

D_2 (Inhabitants of the European Union)

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

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

$D_r := D_1[Residents \text{ -> } Population] + D_2[Inhabitants \text{ -> } Population]$

The result will be:

D_r (Population of United States + European Union)

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

2013	1250

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

To avoid a potentially excessive renaming, VTL 1.1 additionally allows operations where each participating dataset has an explicitly specified component using the dot notation. For instance,

$D_r := D_1.Residents + D_2.Inhabitants$

creates a result with a single measure component named *Result*, which can then be renamed, if necessary, at will:

$D_r := (D_1.Residents + D_2.Inhabitants)[Result\text{->}Population]$

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

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

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

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

Order of execution

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

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

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

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

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

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

Missing Data

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

Missing data can take up two basic forms.

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

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

The handling of missing data in VTL dataset operation can be handled in several ways. One way is to require all participating dataset components used in a computation to be known (corresponding to the notion of "inner join" of dataset components). Another way is to allow some, but not all, components from the participating dataset components to be unknown (corresponding to the notion of "outer join" of components). The mechanics of these approaches is explained in details in the section on the joinexpressions and treatment of NULLs in the Reference Manual.

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:** `null` value belongs to its own type named null. Type null is subsumed by all scalar types, which is to say that `null` value can (in principle) appear wherever a scalar data is expected; this means that it is an allowed value for any scalar type (string, number, boolean, date). However, complex data types (collections, datasets, records, modules, etc.) do not allow `null` values.
- **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-value logic). Therefore the following deduction rules are applied:

TRUE	*or*	`null`	→	TRUE
FALSE	*or*	`null`	→	`null`
TRUE	*and*	`null`	→	`null`

FALSE *and* `null` → FALSE

- **Conditional operations.** The `null` is considered equivalent to FALSE; for example in the control structures of the type (*if (p) -then -else*), the action specified in *–then* is executed if the predicate *p* is TRUE, while the action *-else* is executed if the *p* is FALSE or `null`;
- **Filter clauses.** The `null` is considered equivalent to FALSE; for example in the filter clause [*filter p*], the Data Points for which the predicate *p* is TRUE are selected and returned in the output, while the Data Points for which *p* is FALSE or `null` are discarded.
- **Aggregations**. The aggregations (like *sum*, *avg* and so on) return one Data Point in correspondence to a set of Data Points of the input. In these operations the input Data Points having a `null` value are in general not considered. In the average, for example, they are not considered both in the numerator (the sum) and in the denominator (the count). Specific cases for specific operators are described in the respective sections.
- **Implicit zero**. Arithmetic operators assuming implicit zeros (+,-,*,/) may generate `null` values for the Identifier Components in particular cases (superset-subset relation between the set of the involved Identifier Components). Because `null` values are in general forbidden in the Identifiers, the final outcome of an expression must not contain Identifiers having `null` values. As a momentary exception needed to allow some kinds of calculations, Identifiers having `null` values are accepted in the partial results. To avoid runtime error, possible `null` values of the Identifiers have to be fully eliminated in the final outcome of the expression (through a selection, or other operators), so that the operation of "assignment" (:=) does not encounter them.

If a different behaviour is desired for `null` values, it is possible to **override** them. This can be achieved with the combination of the *calc* 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 Part 2.

D_r `:=` D_1 `[M1 := if M1 is null then 1 else M1] *` D_2

The Attribute Components

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

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

The VTL has different optional behaviours for Attributes and for Measures.

As specified above, Measures are kept in the result by default, whereas Attributes may be assigned a characteristic called “**virality**”, which determines if the Attribute is kept in the result by default or not: a “**viral**” Attribute is kept while a “**non-viral**” Attribute is not kept (the 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).

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

A third aspect is the operation performed on an Attribute. By default, **the operations which apply to the Measures are not applied to the Attributes**, so that the operations on the Attributes need a dedicated specification. If no operations are explicitly defined on a viral Attribute, a default calculation algorithm is applied in order to determine the Attribute’s values in the result. If needed, the VTL default behaviour described here may be overridden by customized default behaviours.

As already mentioned, when the default behaviour is not desired, a different behaviour can be specified by means of the proper use of the *keep*, *calc* and *attrcalc* clauses. In particular, through these clauses, it is possible to override the virality (to keep a *non-viral* Attribute or not to keep a *viral* one), to alter the virality of the Attributes in the result (from *viral* to *non-viral* or vice-versa) and to define a specific calculation algorithm for an Attribute (see the detailed description of these clauses in the Part 2).[18]

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;
- the values of the Attributes which exist and are viral in only one operand are simply copied (obviously, in the case of unary Operators this applies always);
- the Attributes which exist and are viral in multiple operands (i.e. Attributes having the same names) are combined in one Attribute of the result (having the same name also), whose values are calculated according to the default calculation algorithm explained below;

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

---

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

D_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:

D_r = Increased Salary of Employees

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

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

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

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

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

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

D_1 = Births & Deaths of the United States

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

D_2 = Birth & Deaths of the European Union

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

Assuming the weights 1 for "false" and 2 for "true", the Transformation `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

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

Commutative law (1)

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

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

Associative law (2)

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

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

Reflexive law (3)

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

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

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

With regard to this last aspect, the VTL assumes that the **order of execution** of the operations in an expression is determined by the precedence and associativity rules of the Operators applied on the Measures, as already explained in the relevant section. The operations on the Attributes are performed in the same order, independently of the application of the default Attribute propagation rule or user defined operations.

For example, recalling the example already given:

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

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

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

Storage and retrieval of the Data Sets

The Storage

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The Retrieval

Considering again the general form of Transformation:

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

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

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

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

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

The references to persistent Data Sets

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

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

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

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

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

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

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

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

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

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

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

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

The Identification of a persistent Data Set

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

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

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

Examples of good references to Data Sets are:

"NAMESPACE/DS_NAME" (explicit Namespace definition)

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

The specification of a subset of a persistent Data Set

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

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

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

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

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

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

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

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

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

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

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

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

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

I. `DS_NAME/2014.USA.PUBLIC`

This definition in the query string syntax corresponds to:

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

II. `DS_NAME/.USA.PUBLIC`

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

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

III. `DS_NAME/..PUBLIC`

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

```
DS_NAME?SECTOR=PUBLIC
```

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

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

This definition in the query string syntax corresponds to:

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

Conventions for the grammar of the language

General conventions

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

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

```
variable parameter := expression
```

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

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

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

Variable Parameter names

The variable parameters are non-persistent (temporary).

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

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

Reserved words

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

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

Comments

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

For example:

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

Constraints and errors

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

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

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

While the three categories (and subcategories for compile errors) are standardized with codes reported in the remainder of this section, an encoding for specific errors (identified by the last two digits, EE) is not enforced here and can be independently defined by the adopting organization.[19]

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

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

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

---

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

c) only the violating Data Point to be discarded, without generating any exception.

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

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

Compile Time errors (VTL-0xxx)

The VTL grammar specifies the rules to be followed in writing expressions. The VTL language allows the detection at compile time of the possible violation of the **correct syntax**, the use of **wrong types** as parameters for the operators or the **violation of any of the static constraints of the operators** (with respect to the rules described in the Part 2).

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

Syntax errors (VTL-01xx)

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

Examples of syntactically invalid expressions are:

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

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

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

Type errors (VTL-02xx)

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

Examples of expressions that are type-invalid are:

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

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

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

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

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

Static constraint violation errors (VTL-03xx)

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

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

Examples of expressions that violate static constraints are:

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

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

Runtime errors (VTL-1xxx)

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

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

Particular types of runtime errors are:

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

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

Examples of expressions generating runtime errors are:

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

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

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

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

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

Validation errors (VTL-2xxx)

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

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

For example:

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

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

C1		
K1	**K2**	M1
1	A	1000
2	B	200

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

and produces:

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

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

Governance, other requirements and future work

The SDMX Technical Working Group, as mandated by the SDMX Secretariat, is 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 a specific operator (Evaluate) whose purpose is to invoke an external calculation function (routine), provided that this is compatible with the VTL IM and data types.

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

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

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

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

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

Relations with the GSIM Information Model

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

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

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

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

VTL is based on a model for defining mathematical expressions which is called "Transformation model". GSIM does not have a Transformation model, which is however available in the SDMX IM. The VTL IM has been 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 1.0. One reason is that the business process models available in GSIM, DDI and SDMX are not yet fully compatible and univocally mapped.

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

Annex 1 - EBNF

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

EBNF is a standard[20] 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[21], 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[22]), 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.

---

[20] ISO/IEC 14977

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

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