

SDMX Technical Working Group

VTL Task Force

VTL - version 2.0

(Validation & Transformation Language)

Part 1 - User Manual

April 2018

26 Foreword

27

28 The Task force for the Validation and Transformation Language (VTL), created in 2012-2013
29 under the initiative of the SDMX Secretariat, is pleased to present the draft version of VTL 2.0.

30 The SDMX Secretariat launched the VTL work at the end of 2012, moving on from the
31 consideration that SDMX already had a package for transformations and expressions in its
32 information model, while a specific implementation language was missing. To make this
33 framework operational, a standard language for defining validation and transformation rules
34 (operators, their syntax and semantics) had to be adopted, while appropriate SDMX formats
35 for storing and exchanging rules, and web services to retrieve them, had to be designed. The
36 present VTL 2.0 package is only concerned with the first element, i.e., a formal definition of
37 each operator, together with a general description of VTL, its core assumptions and the
38 information model it is based on.

39 The VTL task force was set up early in 2013, composed of members of SDMX, DDI and GSIM
40 communities and the work started in summer 2013. The intention was to provide a language
41 usable by statisticians to express logical validation rules and transformations on data,
42 described as either dimensional tables or unit-record data. The assumption is that this logical
43 formalization of validation and transformation rules could be converted into specific
44 programming languages for execution (SAS, R, Java, SQL, etc.), and would provide at the same
45 time, a “neutral” business-level expression of the processing taking place, against which
46 various implementations can be mapped. Experience with existing examples suggests that
47 this goal would be attainable.

48 An important point that emerged is that several standards are interested in such a kind of
49 language. However, each standard operates on its model artefacts and produces artefacts
50 within the same model (property of closure). To cope with this, VTL has been built upon a
51 very basic information model (VTL IM), taking the common parts of GSIM, SDMX and DDI,
52 mainly using artefacts from GSIM 1.1, somewhat simplified and with some additional detail. In
53 this way, existing standards (GSIM, SDMX, DDI, others) would be allowed to adopt VTL by
54 mapping their information model against the VTL IM. Therefore, although a work-product of
55 SDMX, the VTL language in itself is independent of SDMX and will be usable with other
56 standards as well. Thanks to the possibility of being mapped with the basic part of the IM of
57 other standards, the VTL IM also makes it possible to collect and manage the basic definitions
58 of data represented in different standards.

59 For the reason described above, the VTL specifications are designed at logical level,
60 independently of any other standard, including SDMX. The VTL specifications, therefore, are
61 self-standing and can be implemented either on their own or by other standards (including
62 SDMX). In particular, the work for the SDMX implementation of VTL is going in parallel with
63 the work for designing this VTL version, and will entail a future update of the SDMX
64 documentation.

65 The first public consultation on VTL (version 1.0) was held in 2014. Many comments were
66 incorporated in the VTL 1.0 version, published in March 2015. Other suggestions for
67 improving the language, received afterwards, fed the discussion for building the draft version
68 1.1, which contained many new features, was completed in the second half of 2016 and
69 provided for public consultation until the beginning of 2017.

70 The high number and wide impact of comments and suggestions induced a high workload on
71 the VTL TF, which agreed to proceed in two steps for the publication of the final
72 documentation, taking also into consideration that some first VTL implementation initiatives
73 had already been launched. The first step, the current one, is dedicated to fixing some high-
74 priority features and making them as much stable as possible. A second step, scheduled for
75 the next period, is aimed at acknowledging and fixing other features considered of minor
76 impact and priority, which will be added hopefully without affecting either the features
77 already published in this documentation, or the possible relevant implementations. Moreover,
78 taking into account the number of very important new features that have been introduced in
79 this version in respect to the VTL 1.0, it was agreed that the current VTL version should be
80 considered as a major one and thus named VTL 2.0.

81 The VTL 2.0 package contains the general VTL specifications, independently of the possible
82 implementations of other standards; in its final release, it will include:

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

90 The present document is the part 1.

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

92

93 **Acknowledgements**

94 The VTL specifications has been prepared thanks to the collective input of experts from Bank
95 of Italy, Bank for International Settlements (BIS), European Central Bank (ECB), Eurostat, ILO,
96 INEGI-Mexico, ISTAT-Italy, OECD, Statistics Netherlands, and UNESCO. Other experts from the
97 SDMX Technical Working Group, the SDMX Statistical Working Group and the DDI initiative
98 were consulted and participated in reviewing the documentation.

99 The list of contributors and reviewers includes the following experts: Sami Airo, Foteini
100 Andrikopoulou, David Barraclough, Luigi Bellomarini, Marc Bouffard, Maurizio Capaccioli,
101 Vincenzo Del Vecchio, Fabio Di Giovanni, Jens Dossé, Heinrich Ehrmann, Bryan Fitzpatrick,
102 Tjalling Gelsema, Luca Gramaglia, Arofan Gregory, Gyorgy Gyomai, Edgardo Greising, Dragan
103 Ivanovic, Angelo Linardi, Juan Munoz, Chris Nelson, Stratos Nikoloutsos, Stefano Pambianco,
104 Marco Pellegrino, Michele Romanelli, Juan Alberto Sanchez, Roberto Sannino, Angel Simon
105 Delgado, Daniel Suranyi, Olav ten Bosch, Laura Vignola, Fernando Wagener and Nikolaos
106 Zisimos.

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

109 Table of contents

110		
111	FOREWORD	2
112	TABLE OF CONTENTS	4
113	INTRODUCTION	7
114	<i>Structure of the document</i>	7
115	GENERAL CHARACTERISTICS OF THE VTL	9
116	USER ORIENTATION	9
117	INTEGRATED APPROACH.....	9
118	ACTIVE ROLE FOR PROCESSING.....	11
119	INDEPENDENCE OF IT IMPLEMENTATION	12
120	EXTENSIBILITY, CUSTOMIZABILITY	13
121	LANGUAGE EFFECTIVENESS.....	14
122	EVOLUTION OF VTL 2.0 IN RESPECT TO VTL 1.0	16
123	THE INFORMATION MODEL	16
124	STRUCTURAL ARTEFACTS AND REUSABLE RULES.....	16
125	THE CORE LANGUAGE AND THE STANDARD LIBRARY	17
126	THE USER DEFINED OPERATORS.....	17
127	THE VTL DEFINITION LANGUAGE	17
128	THE FUNCTIONAL PARADIGM.....	18
129	THE OPERATORS	19
130	VTL INFORMATION MODEL	20
131	INTRODUCTION.....	20
132	GENERIC MODEL FOR DATA AND THEIR STRUCTURES	22
133	<i>Data model diagram</i>	23
134	<i>Explanation of the Diagram</i>	24
135	<i>Functional Integrity</i>	25
136	<i>Relationships between VTL and GSIM</i>	26
137	<i>Examples</i>	27
138	<i>The data artefacts</i>	30
139	GENERIC MODEL FOR VARIABLES AND VALUE DOMAINS.....	31
140	<i>Variable and Value Domain model diagram</i>	31

141	<i>Explanation of the Diagram</i>	32
142	<i>Relations and operations between Code Items</i>	34
143	<i>Conditioned Code Item Relations</i>	37
144	<i>The historical changes</i>	37
145	<i>The Variables and Value Domains artefacts</i>	39
146	GENERIC MODEL FOR TRANSFORMATIONS.....	41
147	<i>Transformations model diagram</i>	44
148	<i>Explanation of the diagram</i>	44
149	<i>Examples</i>	45
150	<i>Functional paradigm</i>	46
151	<i>Transformation Consistency</i>	46
152	VTL DATA TYPES	48
153	DATA TYPES OVERVIEW	49
154	<i>Data Types model diagram</i>	49
155	<i>Explanation of the diagram</i>	50
156	<i>General conventions for describing the types</i>	50
157	SCALAR TYPES.....	51
158	<i>Basic Scalar Types</i>	51
159	<i>Value Domain Scalar Types</i>	54
160	<i>Set Scalar Types</i>	55
161	<i>External representations and literals used in the VTL Manuals</i>	55
162	<i>Conventions for describing the scalar types</i>	58
163	COMPOUND DATA TYPES	60
164	<i>Component Types</i>	60
165	<i>Data Set Types</i>	62
166	<i>Product Types</i>	64
167	<i>Operator Types</i>	64
168	<i>Ruleset Types</i>	65
169	<i>Universal Set Types</i>	66
170	<i>Universal List Types</i>	66
171	VTL TRANSFORMATIONS	67
172	THE EXPRESSION	68
173	THE ASSIGNMENT.....	69
174	THE RESULT	70

175	THE NAMES.....	71
176	<i>The artefact names</i>	71
177	<i>The environment name</i>	72
178	<i>The connection to the persistent storage</i>	73
179	VTL OPERATORS	74
180	THE CATEGORIES OF VTL OPERATORS.....	74
181	THE INPUT PARAMETERS.....	75
182	THE INVOCATION OF VTL OPERATORS.....	76
183	LEVEL OF OPERATION.....	77
184	THE OPERATORS' BEHAVIOUR.....	78
185	<i>The Join operators</i>	78
186	<i>Other operators: default behaviour on Identifiers, Measures and Attributes</i>	79
187	<i>The Identifier Components and the Data Points matching</i>	80
188	<i>The operations on the Measure Components</i>	83
189	<i>Operators which change the basic scalar type</i>	88
190	<i>Boolean operators</i>	90
191	<i>Set operators</i>	90
192	BEHAVIOUR FOR MISSING DATA.....	90
193	BEHAVIOUR FOR ATTRIBUTE COMPONENTS.....	92
194	<i>The Attribute propagation rule</i>	93
195	<i>Properties of the Attribute propagation algorithm</i>	96
196	GOVERNANCE, OTHER REQUIREMENTS AND FUTURE WORK	97
197	RELATIONS WITH THE GSIM INFORMATION MODEL.....	98
198	ANNEX - EBNF	100
199	PROPERTIES OF VTL GRAMMAR.....	100
200		

201 Introduction

202 This document presents the Validation and Transformation Language (also known as 'VTL')
203 version 2.0.

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

206 The first development of VTL aims at enabling, as a priority, the formalisation of data
207 validation algorithms rather than tackling more complex algorithms for data compilation. In
208 fact, the assessment of business cases showed that the majority of the institutions ascribes
209 (prescribes) a higher priority to a standard language for supporting the validation processes
210 and in particular to the possibility of sharing validation rules with the respective data
211 providers, in order to specify the quality requirements and allow validation also before
212 provision.

213 This document is the outcome of a second iteration of the first phase, and therefore still
214 presents a version of VTL primarily oriented to support the data validation. However, as the
215 features needed for validation also include simple calculations, this version of VTL can
216 support basic compilation needs as well. In general, validation is considered as a particular
217 case of transformation; therefore, the term "Transformation" is meant to be more general,
218 including validation as well. The actual operators included in this version of VTL are
219 described in the Reference Manual.

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

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

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

231 **Structure of the document**

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

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

235 The changes of VTL 2.0 in respect to VTL 1.0.

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

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

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

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

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

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

251 The Extended BNF (EBNF) representation of the VTL 1.0 package is available at
252 https://sdmx.org/?page_id=5096. The VTL 2.0 representation will be added as soon as it is
253 available.

254

255 General characteristics of the VTL

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

260 User orientation

261 ⇒ The language is designed for users without information technology (IT) skills, who
262 should be able to define calculations and validations independently, without the
263 intervention of IT personnel;

264 ○ The language is based on a “user” perspective and a “user” information model
265 (IM) and not on possible IT perspectives (and IMs)

266 ○ As much as possible, the language is able to manipulate statistical data at an
267 abstract/conceptual level, independently of the IT representation used to
268 store or exchange the data observations (e.g. files, tables, xml tags), so
269 operating on abstract (from IT) model artefacts to produce other abstract
270 (from IT) model artefacts

271 ○ It references IM objects and does not use direct references to IT objects

272 ⇒ The language is intuitive and friendly (users should be able to define and understand
273 validations and transformations as easily as possible), so the syntax is:

274 ○ Designed according to mathematics, which is a universal knowledge;

275 ○ Expressed in English to be shareable in most countries;

276 ○ As simple, intuitive and self-explanatory as possible;

277 ○ Based on common mathematical expressions, which involve “operands”
278 operated on by “operators” to obtain a certain result;

279 ○ Designed with minimal redundancies (e.g. possibly avoiding operators
280 specifying the same operation in different ways without concrete reasons).

281 ⇒ The language is oriented to statistics, and therefore it is capable of operating on
282 statistical objects and envisages the operators needed in the statistical processes and
283 in particular in the data validation phases, for example:

284 ○ Operators for data validations and edit;

285 ○ Operators for aggregation, even according to hierarchies;

286 ○ Operators for dimensional processing (e.g. projection, filter);

287 ○ Operators for statistics (e.g. aggregation, mean, median, variance ...);

288 Integrated approach

289 ⇒ The language is independent of the statistical domain of the data to be processed;

- 290 ○ VTL has no dependencies on the subject matter (the data content);
- 291 ○ VTL is able to manipulate statistical data in relation to their structure.
- 292 ⇒ The language is suitable for the various typologies of data of a statistical environment
293 (for example dimensional data, survey data, registers data, micro and macro,
294 quantitative and qualitative) and is supported by an information model (IM) which
295 covers these typologies;
- 296 ○ The IM allows the representation of the various typologies of data of a
297 statistical environment at a conceptual/logical level (in a way abstract from IT
298 and from the physical storage);
- 299 ○ The various typologies of data are described as much as possible in an
300 integrated way, by means of common IM artefacts for their common aspects;
- 301 ○ The principle of the Occam's razor is applied as an heuristic principle in
302 designing the conceptual IM, so keeping everything as simple as possible or, in
303 other words, unifying the model of apparently different things as much as
304 possible.
- 305 ⇒ The language (and its IM) is independent of the phases of the statistical process and
306 usable in any one of them;
- 307 ○ Operators are designed to be independent of the phases of the process, their
308 syntax does not change in different phases and is not bound to some
309 characteristic restricted to a specific phase (operators' syntax is not aware of
310 the phase of the process);
- 311 ○ In principle, all operators are allowed in any phase of the process (e.g. it is
312 possible to use the operators for data validation not only in the data collection
313 but also, for example, in data compilation for validating the result of a
314 compilation process; similarly it is possible to use the operators for data
315 calculation, like the aggregation, not only in data compilation but also in data
316 validation processes);
- 317 ○ Both collected and calculated data are equally permitted as inputs of a
318 calculation, without changes in the syntax of the operators/expression;
- 319 ○ Collected and calculated data are represented (in the IM) in a homogeneous
320 way with regards to the metadata needed for calculations.
- 321 ⇒ The language is designed to be applied not only to SDMX but also to other standards;
- 322 ○ VTL, like any consistent language, relies on a specific information model, as it
323 operates on the VTL IM artefacts to produce other VTL IM artefacts. In
324 principle, a language cannot be applied as-is to another information model
325 (e.g. SDMX, DDI, GSIM); this possibility exists only if there is a unambiguous
326 correspondence between the artefacts of those information models and the
327 VTL IM (that is if their artefacts correspond to the same mathematical notion);
- 328 ○ The goal of applying the language to more models/standards is achieved by
329 using a very simple, generic and conceptual Information Model (the VTL IM),
330 and mapping this IM to the models of the different standards (SDMX, DDI,
331 GSIM, ...); to the extent that the mapping is straightforward and unambiguous,

- 332 the language can be inherited by other standards (with the proper
333 adjustments);
- 334 ○ To achieve an unambiguous mapping, the VTL IM is deeply inspired by the
335 GSIM IM and uses the same artefacts when possible¹; in fact, GSIM is designed
336 to provide a formal description of data at business level against which other
337 information models can be mapped; moreover, loose mappings between GSIM
338 and SDMX and between GSIM and DDI are already available²; a very small
339 subset of the GSIM artefacts is used in the VTL IM in order to keep the model
340 and the language as simple as possible (Occam's razor principle); these are the
341 artefacts strictly needed for describing the data involved in Transformations,
342 their structure and the variables and value domains;
 - 343 ○ GSIM artefacts are supplemented, when needed, with other artefacts that are
344 necessary for describing calculations; in particular, the SDMX model for
345 Transformations is used;
 - 346 ○ As mentioned above, the definition of the VTL IM artefacts is based on
347 mathematics and is expressed at an abstract user level.

348 Active role for processing

- 349 ⇒ The language is designed to make it possible to drive in an active way the execution of
350 the calculations (in addition to documenting them)
- 351 ⇒ For the purpose above, it is possible either to implement a calculation engine that
352 interprets the VTL and operates on the data or to rely on already existing IT tools (this
353 second option requires a translation from the VTL to the language of the IT tool to be
354 used for the calculations)
- 355 ⇒ The VTL grammar is being described formally using the universally known Backus
356 Naur Form notation (BNF), because this allows the VTL expressions to be formally
357 parsed and then processed; the formal description allow the expressions:
 - 358 ○ To be parsed against the rules of the formal grammar; on the IT level, this
359 requires the implementation of a parser that compiles the expressions and
360 checks their correctness;
 - 361 ○ To be translated from the VTL to the language of the IT tool to be used for the
362 calculation; on the IT level, this requires the implementation of a proper
363 translator;
 - 364 ○ To be translated from/to other languages if needed (through the
365 implementation of a proper translator.
- 366 ⇒ The inputs and the outputs of the calculations and the calculations themselves are
367 artefacts of the IM

¹ See the next section (VTL Information Model) and the section "Relationships between VTL and GSIM"

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

- 368
369
- This is a basic property of any robust language because it allows calculated data to be operands of further calculations;
- 370
371
372
373
- 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;
- 374
375
376
- Because the definition of the algorithms of the calculations is based on the definition of their input artefacts (in particular on the data structure of the input data), the latter must be available when the calculation is defined;
- 377
378
379
380
- 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);
- 381
382
383
384
- 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.
- 385
386
387
- ⇒ The VTL and its information model make it possible to check automatically the overall consistency of the definitions of the calculations, including with respect to the artefact of the IM, and in particular to check:
- 388
- the correctness of the expressions with respect to the syntax of the language
- 389
390
391
392
393
- 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)
- 394
395
396
397
- 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);

398 Independence of IT implementation

- 399
400
- ⇒ According to the “user orientation” above, the language is designed so that users are not required to be aware of the IT solution;
- 401
402
403
- 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.
- 404
405
406
- ⇒ 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);

- 407 ○ The VTL provides only for a logical/conceptual layer for defining the data
408 transformations, which applies on a logical/conceptual layer of data
409 definitions
- 410 ○ The VTL does not prescribe any technical/physical tool or solution, so that it
411 is possible to implement the VTL by using many different IT tools
- 412 ○ The link between the logical/conceptual layer of the VTL definitions and the IT
413 implementation layer is out of the scope of the VTL;
- 414 ⇒ The language does not require to the users the awareness of the storage data
415 structure; the operations on the data are specified according to the conceptual/logical
416 structure, and so are independent of the storage structure; this ensures that the
417 storage structure may change without necessarily affecting the conceptual structure
418 and the user expressions;
- 419 ○ Data having the same conceptual/logical structure may be accessed using the
420 same statements, even if they have different storage structures;
- 421 ○ The VTL provides for data storage and retrieval at a conceptual/logical level;
422 the mapping and the conversion between the conceptual and the storage
423 structures of the data is left to the IT implementation (and users need not be
424 aware of it);
- 425 ○ By mapping the logical and the storage data structures, the IT
426 implementations can make it possible to store/retrieve data in/from different
427 IT data stores (e.g. relational databases, dimensional databases, xml files,
428 spread-sheets, traditional files);
- 429 ⇒ The language is not strictly connected with some specific IT tool to perform the
430 calculations (e.g. SQL, statistical packages, other languages, XML tools,...);
- 431 ○ The syntax of the VTL is independent of existing IT calculation tools;
- 432 ○ On the IT level, this may require a translation from the VTL to the language of
433 the IT tool to be used for the calculation;
- 434 ○ By implementing the proper translations at the IT level, different institutions
435 can use different IT tools to execute the same algorithms; moreover, it is
436 possible for the same institution to use different IT tools within an integrated
437 solution (e.g. to exploit different abilities of different tools);
- 438 ○ VTL instructions do not change if the IT solution changes (for example
439 following the adoption of another IT tool), so avoiding impacts on users as
440 much as possible;

441 Extensibility, customizability

- 442 ⇒ The language is made of few “core” constructs, which are the fundamental building
443 blocks into which any operation can be decomposed, and a “standard library”, which
444 contains a number of standard operators built from the core constructs; these are the
445 standard parts of the language, which can be extended gradually by the VTL
446 maintenance body, enriching the available operators according to the evolution of the
447 business needs, so progressively making the language more powerful;

- 448 ⇒ Other organizations can define additional operators having a customized behaviour
449 and a functional syntax, so extending their own library by means of custom-designed
450 operators. As obvious, these additional operators are not part of the standard VTL
451 library. To exchange VTL definitions with other institutions, the possible custom
452 libraries need to be pre-emptively shared.
- 453 ⇒ In addition, it is possible to call external routines of other languages/tools, provided
454 that they are compatible with the IM; this requisite is aimed to fulfil specific
455 calculation needs without modifying the operators of the language, so exploiting the
456 power of the other languages/tools if necessary for specific purposes. In this case:
- 457 ○ The external routines should be compatible with, and relate back to, the
458 conceptual IM of the calculations as for its inputs and outputs, so that the
459 integrity of the definitions is ensured
 - 460 ○ The external routines are not part of the language, so their use is subject to
461 some limitations (e.g. it is impossible to parse them as if they were operators
462 of the language)
 - 463 ○ The use of external routines compromises the IT implementation
464 independence, the abstraction and the user orientation; therefore external
465 routines should be used only for specific needs and in limited cases, whereas
466 widespread and generic needs should be fulfilled through the operators of the
467 language;
- 468 ⇒ Whilst an Organisation adopting VTL can extend its own library by defining
469 customized parts, on its own total responsibility, in order to improve the standard
470 language for specific purposes (e.g. for supporting possible algorithms not permitted
471 by the standard part), it is important that the customized parts remain compliant with
472 the VTL IM and the VTL fundamentals. Adopting Organizations are totally in charge of
473 any activity for maintaining and sharing their customized parts. Adopting
474 Organizations are also totally in charge of any possible maintenance activity to
475 maintain the compliance between their customized parts and the possible VTL future
476 versions.

477 Language effectiveness

- 478 ⇒ The language is oriented to give full support to the various typologies of data of a
479 statistical environment (for example dimensional data, survey data, registers data,
480 micro and macro, quantitative and qualitative, ...) described as much as possible in a
481 coherent way, by means of common IM artefacts for their common aspects, and
482 relying on mathematical notions, as mentioned above. The various types of statistical
483 data are considered as mathematical functions, having independent variables
484 (Identifiers) and dependent variables (Measures, Attributes³), whose extensions can
485 be thought as logical tables (DataSets) made of rows (Data Points) and columns
486 (Identifiers, Measures, Attributes).

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

- 487 ⇒ The language supports operations on the Data Sets (i.e. mathematical functions) in
488 order to calculate new Data Sets from the existing ones, on their structure components
489 (Identifiers, Measures, Attributes), on their Data Points.
- 490 ⇒ The algorithms are specified by means of mathematical expressions which compose
491 the operands (Data Sets, Components ...) by means of operators (e.g. +,-,*,/,>,<) to
492 obtain a certain result (Data Sets, Components ...);
- 493 ⇒ The validation is considered as a kind of calculation having as an operand the Data
494 Sets to be validated and producing a Data Set containing information about the result
495 of the validation;
- 496 ⇒ Calculations on multiple measures are supported by most operators, as well as
497 calculations on the attributes of the Data Sets and calculations involving missing
498 values;
- 499 ⇒ The operations are intended to be consistent with the real world historical changes
500 which induce changes of the artefacts (e.g. of the code lists, of the hierarchies ...);
501 however, because different standards may represent historical changes in different
502 ways, the implementation of this aspect is left to the standards (e.g. SDMX, DDI ...), to
503 the institutions and to the implementers adopting the VTL and therefore the VTL
504 specifications does not prescribe any particular methodology for representing the
505 historical changes of the artefacts (e.g. versioning, qualification of time validity);
- 506 ⇒ Almost all the VTL operators can be nested, meaning that in the invocation of an
507 operator any operand can be the result of the invocation of other operators which
508 calculate it;
- 509 ⇒ The results of the calculations can be permanently stored or not, according to the
510 needs;
- 511

512 Evolution of VTL 2.0 in respect to VTL 1.0

513 Important contributions gave origin to the work that brought to this VTL 2017 version.

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

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

520 The main improvements are described below.

521 The Information Model

522 The VTL Information Model describes the artefacts that VTL manipulates (i.e. it provides a
523 generic model for defining Data and their structures, Variables, Value Domains and so on) and
524 the structural metadata which define validations and transformations (i.e. a generic model for
525 Transformations).

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

529 As already said, VTL is intended to operate at logical/conceptual level and independently of
530 the implementation, actually allowing different implementations. For this reason, VTL-IM 2.0
531 provides only for a core abstract view of data and calculations and leaves out the
532 implementation aspects.

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

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

539 The implementation of the VTL-IM 2.0 abstract model artefacts needs to take into account the
540 specificities of the standards (like SDMX, DDI ...) and the information systems for which it is
541 used.

542 Structural artefacts and reusable rules

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

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

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

551 The core language and the standard library

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

558 The suggestion above has been acknowledged, so VTL 2.0 manipulation language consists of a
559 core set of primitive operators and a standard library of derived operators, definable in term
560 of the primitive ones. The standard library contains essentially the VTL 1.0 operators
561 (possibly enhanced) and the new operators introduced with VTL 2.0 (see below).

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

564 The user defined operators

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

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

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

577 The VTL Definition Language

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

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

581 In fact, VTL 2.0 allows reusable rules and user-defined operators, which do not exist in VTL
582 1.0 and need to be defined beforehand in order to be invoked in the expressions of the VTL
583 manipulation language. The VTL-DL provides for their definition.

584 Second, VTL 1.0 was initially intended to work on top of an existing standard, such as SDMX,
585 DDI or other, and therefore the definition of the artefacts to be manipulated (Data and their

586 structures, Variables, Value Domains and so on) was assumed to be made using the
587 implementing standards and not VTL itself.

588 During the work for the VTL 1.1 draft version, it was proposed to make the VTL definition
589 language able to define also those VTL-IM artefacts that have to be manipulated. A draft
590 version of a possible artefacts definition language was included in VTL 1.1 public consultation,
591 held until the beginning of 2017. The comments received and the following analysis
592 evidenced that the artefact definition language cannot include the aspects which are left out of
593 the IM (for example the representation of the historical changes of the real world impacting
594 the model artefacts) yet are: i. needed in the implementations; ii. influenced by other
595 implementation-specific aspects; iii. in real applications, bound to be extended by means of
596 other context-related metadata and adapted to the specific environment.

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

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

601 The functional paradigm

602 In the VTL Information Model, the various types of statistical data are considered as
603 mathematical functions, having independent variables (Identifiers) and dependent variables
604 (Measures, Attributes), whose extensions can be thought of as logical tables (Data Sets) made
605 of rows (Data Points) and columns (Identifiers, Measures, Attributes). Therefore, the main
606 artefacts to be manipulated using VTL are the logical Data Sets, i.e., first-order mathematical
607 functions⁴.

608 Accordingly, VTL uses a functional programming paradigm, meaning a paradigm that treats
609 computations as the evaluation of higher-order mathematical functions⁵, which manipulate
610 the first-order ones (i.e., the logical Data Sets), also termed “operators” or “functionals”. The
611 functional paradigm avoids changing-state and mutable data and makes use of expressions for
612 defining calculations.

613 It was observed, however, that the functional paradigm was not sufficiently achieved in VTL
614 1.0 because in some particular cases a few operators could have produced non-functional
615 results. In effects, even if this regarded only temporary results (not persistent), in specific
616 cases, this behaviour could have led to unexpected results in the subsequent calculation chain.

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

⁴ A first-order function is a function that does not take other functions as arguments and does not provide another function as result.

⁵ A higher-order function is a function that takes one or more other functions as arguments and/or provides another function as result.

619 The operators

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

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

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

634 Introduction

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

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

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

647 VTL can manage persistent or temporary artefacts, the former stored persistently in the
648 information system, the latter only used temporarily. The definition of the persistent artefact
649 must be persistent as well, while the definition of temporary artefacts can be temporary⁶.

650 The VTL IM provides a formal description at business level of the artefacts which VTL can
651 manipulate, which is the same purpose as the Generic Statistical Information Model (GSIM)
652 with a broader scope. As a matter of fact, the VTL Information Model uses GSIM artefacts as
653 much as possible (GSIM 1.1 version)⁷. Besides, GSIM already provides a first mapping with
654 SDMX and DDI that can be used for the technical implementation⁸. Note that the description of
655 the GSIM 1.1 classes and relevant definitions can be consulted in the “Clickable GSIM” of the
656 UNECE site⁹. However, the detailed mapping between the VTL IM and the IMs of the other
657 standards is out of the scope of this document and is left to the competent bodies of the other
658 standards.

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

661 The VTL IM provides for an abstract view of the core artefacts used in the VTL calculations
662 and intentionally leaves out implementation aspects. Some other aspects, even if logically
663 related to the representation of data and calculations, are also left out because they can

⁶ The definition of a temporary artefact can be also persistent, if needed.

⁷ See also the section “Relations with the GSIM Information model”

⁸ 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.

⁹ Hyperlink “<http://www1.unece.org/stat/platform/display/GSIMclick/Clickable+GSIM>”

664 depend on the actual implementation too (for example, the textual descriptions of the VTL
665 artefacts, the representation of the historical changes of the real world).

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

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

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

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

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

680 Level 0 – the real world

681 Level 1 – the extensions of the data which model some aspect of the real world. For
682 example, the content of the data set “*population from United Nations*”:

683	<i>Year</i>	<i>Country</i>	<i>Population</i>
684	<i>2016</i>	<i>China</i>	<i>1,403,500,365</i>
685	<i>2016</i>	<i>India</i>	<i>1,324,171,354</i>
686	<i>2016</i>	<i>USA</i>	<i>322,179,605</i>
687	<i>...</i>		
688	<i>2017</i>	<i>China</i>	<i>1,409,517,397</i>
689	<i>2017</i>	<i>India</i>	<i>1,339,180.127</i>
690	<i>2017</i>	<i>USA</i>	<i>324,459,463</i>
691	<i>...</i>		

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

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

700 A similar approach is very largely used, in particular in the information technology and for
701 example by the Object Management Group¹⁰, even if the terminology and the enumeration of
702 the levels is different. The main correspondences are:

703 VTL Level 1 (extensions) – OMG M0 (instances)

¹⁰ For example in the Common Warehouse Metamodel and Meta-Object Facility specifications

704 VTL Level 2 (definitions) – OMG M1 (models)

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

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

714 The VTL IM is illustrated in the following sections.

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

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

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

726 Generic Model for Data and their structures

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

729 As already said, GSIM artefacts are used as much as possible. Some differences between this
730 model and GSIM are because, in the VTL IM, both unit and dimensional data are considered as
731 first-order mathematical functions having independent and dependent variables and are
732 treated in the same way.

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

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

742 In this way, the manipulation of any kind of data (unit and dimensional) is brought back to the
743 manipulation of very simple and well-known objects, which can be easily understood and
744 managed by users. According to these assumptions, there would no longer be the need of
745 distinguishing between unit and dimensional data, and in fact VTL does not introduces such a

746 distinction at all. Nevertheless, even if such a distinction is not part of the VTL IM, this aspect
 747 is illustrated hereinafter in order to make it easier to map the VTL IM to the GSIM IM and the
 748 DDI IM, which have such a distinction.

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

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

762

763 **Data model diagram**

764

765

766

767

768

769

770

771

772

773

774

775

776

777

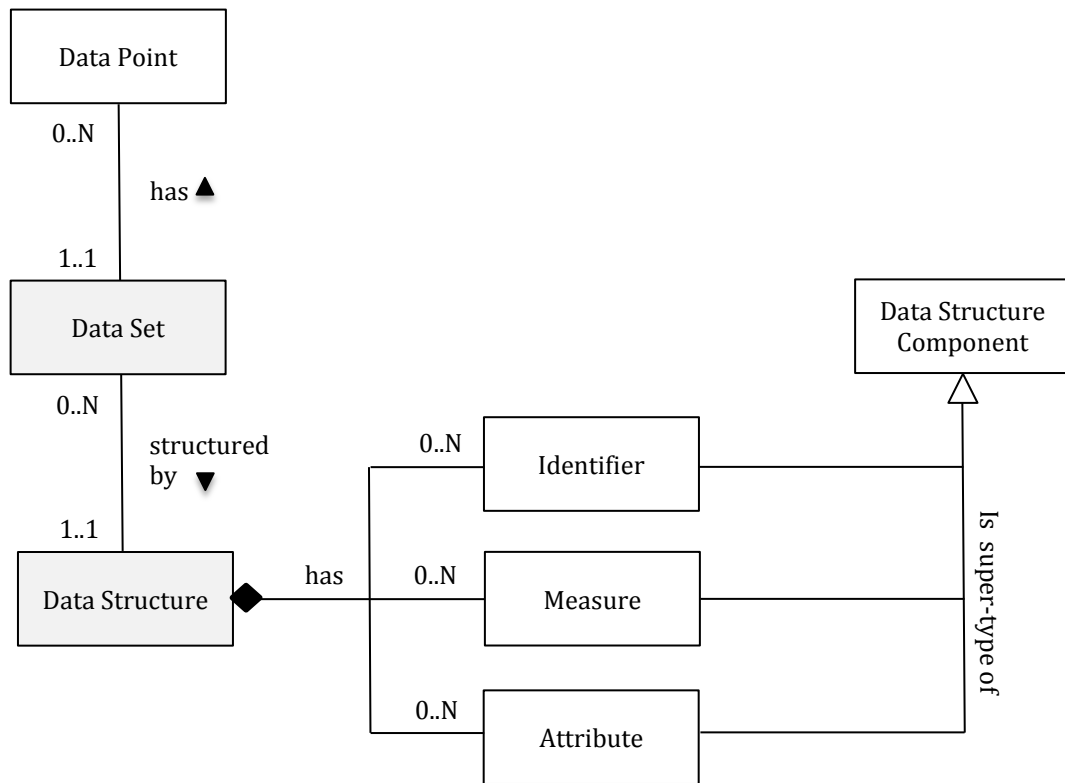
778

779

780

781

782



780 White box: same artefact as in GSIM 1.1

781 Light grey box: similar to GSIM 1.1

782

783 Explanation of the Diagram

784 **Data Set:** a mathematical function (logical table) that describes some properties of some
785 groups of units of a population. In general, the groups of units may be composed of one or
786 more units. For unit data, each group is composed of a single unit. For dimensional data, each
787 group may be composed of any number of units. A VTL Data Set is considered as a logical set
788 of observations (Data Points) having the same logical structure and the same general
789 meaning, independently of the possible physical representation or storage. Between the VTL
790 Data Sets and the physical datasets there can be relationships of any cardinality: for example,
791 a VTL Data Set may be stored either in one or in many physical data sets, as well as many VTL
792 Data Sets may be stored in the same physical datasets (or database tables). The mapping
793 between the VTL logical artefacts and the physical artefacts is left to the VTL implementations
794 and is out of scope of this document. The VTL Data Set is similar to the GSIM Data Set, the
795 relationship between them is described in a following section.

796 **Data Point:** a single value of the function, i.e. a single association between the values of the
797 independent variables and the values of the dependent variables. A Data Point corresponds to
798 a row of the logical table that describes the function, therefore the extension of the function
799 (Data Set) is a set of Data Points. Some Data Points of the function can be unknown (i.e.
800 missing or NULL), for example the possible ones relevant to future dates. The single Data
801 Points do not need to be individually defined, because their definition is the definition of the
802 function (i.e. the Data Set definition). This artefact is the same as the GSIM Data Point.

803 **Data Structure:** the structure of a mathematical function, having independent and dependent
804 variables. The independent variables are called “Identifier components”, the dependent
805 variables are called either “Measure Components” or “Attribute Components”. The distinction
806 between Measure and Attribute components is conventional and essentially based on their
807 meaning: the Measure Components give information about the real world, while the Attribute
808 components give information about the function itself. The VTL Data Structure is similar to
809 the GSIM Data Structure, the relationship between them is described in a following section.

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

812 **Identifier Component** (or simply Identifier): a component of the data structure that is
813 an independent variable of the function. This artefact is the same as in GSIM. In respect
814 to SDMX, an Identifier Component may be either a **Group Identifier**, which contributes
815 to identify a group of statistical units and correspond to a SDMX Dimension, or a
816 **Measure Identifier**, which contributes to identify a Measure and corresponds to a
817 SDMX Measure Dimension.

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

821 **Attribute Component** (or simply Attribute): a component of the data structure that is
822 a dependent variable of the function and gives information about the function itself.
823 This artefact is the same as in GSIM. In case the automatic propagation of the Attributes
824 is supported (see the section “Behaviour for Attribute Components”), the Attributes
825 can be further classified in normal Attributes (not automatically propagated) and Viral
826 Attributes (automatically propagated).

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

829 There can be from 0 to N Measures in a Data Structure. A Data Set without Measures is
830 allowed because the Identifiers can be considered as functional dependent from themselves
831 (so having also the role of Measure). In an equivalent way, the combinations of values of the
832 Identifiers can be considered as “true” (i.e. existing), therefore it can be thought that there is
833 an implicit Boolean measure having value “TRUE” for all the Data Points.¹¹

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

838 Note that the VTL in most cases manages Measure and Attribute Components in different
839 ways, as explained in the section “The general behaviour of operations on datasets” below,
840 therefore the distinction between Measures and Attributes may be significant for the VTL.

841 **Represented Variable:** a characteristic of a statistical population (e.g. the country of birth)
842 represented in a specific way (e.g. through the ISO numeric country code). This artefact is the
843 same as in GSIM. A represented variable may contribute to define any number of Data
844 Structure Components.

845 **Functional Integrity**

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

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

- 853 • Each Component has a distinct name in the Data Structure of the Data Set and contains
854 one scalar value for each Data Point.
- 855 • All the Identifier Components of the Data Set must contain a significant value for all the
856 Data Points (i.e. such value cannot be unknown (“NULL”)).
- 857 • In a Data Set there cannot exist two or more Data Points having the same values for all
858 the Identifier Components (i.e. the same Data Point key).
- 859 • When a Measure or Attribute Component has no significant value (i.e. “NULL”) for a
860 Data Point, it is considered unknown for that Data Point.

¹¹ For example, this is the case of a relationship that does not have properties: imagine a Data Set containing the relationship between the students and the courses that they have followed, without any other information: the corresponding Data Set would have StudentId and CourseId as Identifiers and would not have any explicit measure.

- 861 • When a Data Point is missing (i.e. a possible combination of values of the independent
862 variables is missing), all its Measure and Attribute Components are by default
863 considered unknown (unless otherwise specified).

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

866

867 Relationships between VTL and GSIM

868 As mentioned earlier, the VTL Data Set and Data Structure artefacts are similar to the
869 corresponding GSIM artefact. VTL, however, does not make a distinction between Unit and
870 Dimensional Data Sets and Data Structures.

871 In order to explain the relationships between VTL and GSIM, the distinction between Unit and
872 Dimensional Data Sets can be introduced virtually even in the VTL artefacts. In particular, the
873 GSIM Data Set may be a GSIM Dimensional Data Set or a GSIM Unit Data Set, while a VTL Data
874 Set may (virtually) be:

875 either a (virtual) **VTL Dimensional Data Set**: a kind of (Logical) Data Set describing
876 groups of units of a population that may be composed of many units. This (virtual)
877 artefact would be the same as the GSIM Dimensional Data Set;

878 or a (virtual) **VTL Unit Data Set**: a kind of (Logical) Data Set describing single units of
879 a population. This (virtual) artefact would be the same as the Unit Data Record in
880 GSIM, which has its own structure and can be thought of as a mathematical function.
881 The difference is that the VTL Unit Data Set would not correspond to the GSIM Unit
882 Data Set, because the latter cannot be considered as a mathematical function: in fact it
883 can have many GSIM Unit Data Records with different structures.

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

888 either a (virtual) **VTL Dimensional Data Structure**: the structure of (0..n)
889 Dimensional Data Sets. This artefact would be the same as in GSIM;

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

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

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

903

904 GSIM – VTL mapping diagram about data structures:

905

906

907

908

909

910

911

912

913

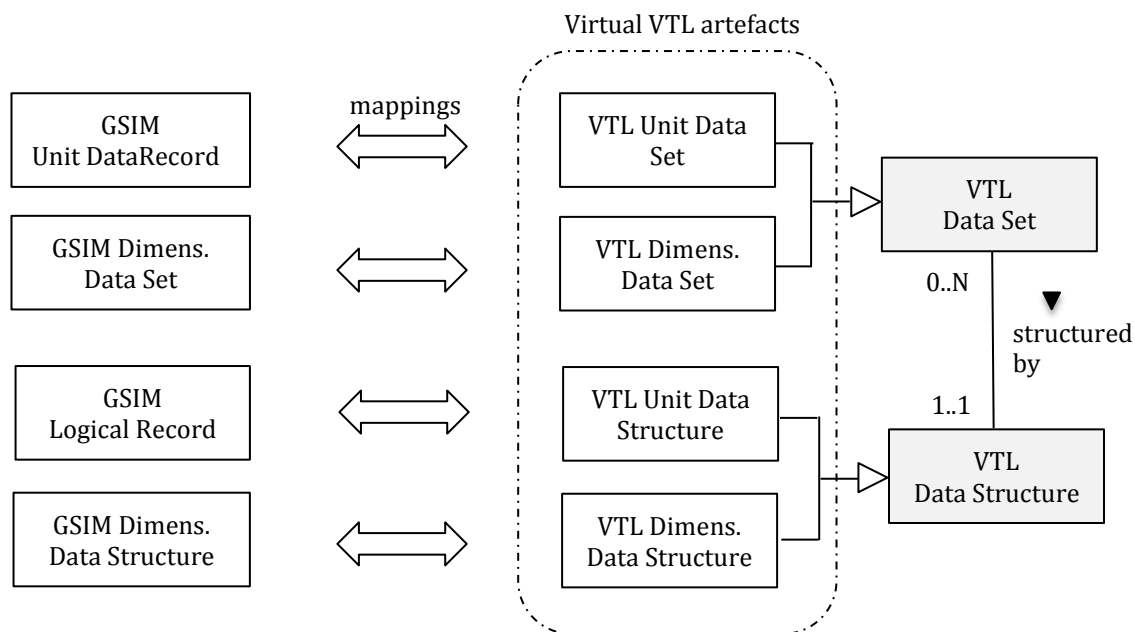
914

915

916

917

918



919 Examples

920 As a first simple example of Data Sets seen as mathematical functions, let us consider the
921 following table:

922

923

Production of the American Countries

924

925

926

927

928

929

930

931

932

933

<i>Ref.Date</i>	<i>Country</i>	<i>Meas.Name</i>	<i>Meas.Value</i>	<i>Status</i>
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
...

934 This table is equivalent to a proper mathematical function: in fact, it fulfils the functional
935 integrity requirements above. The Table can be defined as a Data Set, whose name can be

936 “Production of the American Countries”. Each row of the table is a Data Point belonging to the
 937 Data Set. The Data Structure of this Data Set has five Data Structure Components:

- 938 • Reference Date (Identifier Component)
- 939 • Country (Identifier Component)
- 940 • Measure Name (Identifier Component - Measure Identifier)
- 941 • Measure Value (Measure Component)
- 942 • Status (Attribute Component)

943
 944 As a second example, let us consider the following physical table, in which the symbol “###”
 945 denotes cells that are not allowed to contain a value or contain the “NULL” value.
 946

947 *Institutional Unit Data*

948 <i>Row Type</i>	949 <i>I.U. ID</i>	<i>Ref.Date</i>	<i>I.U. Name</i>	<i>I.U. Sector</i>	<i>Assets</i>	<i>Liabilities</i>
950 I	A	###	AAAAA	Private	###	###
951 II	A	2013	###	###	1000	800
952 II	A	2014	###	###	1050	750
953 I	B	###	BBBBB	Public	###	###
954 II	B	2013	###	###	1200	900
955 II	B	2014	###	###	1300	950
956 I	C	###	CCCCC	Private	###	###
957 II	C	2013	###	###	750	900
958 II	C	2014	###	###	800	850
959

960
 961 This table does not fulfil the functional integrity requirements above because its rows (i.e. the
 962 Data Points) either have different structures (in term of allowed columns) or have NULL
 963 values in the Identifiers. However, it is easy to recognize that there exist two possible
 964 functional structures (corresponding to the Row Types I and II), so that the original table can
 965 be split in the following ones:

966
 967 *Row Type I - Institutional Unit register*

968 <i>I.U. ID</i>	<i>I.U. Name</i>	<i>I.U. Sector</i>
969 A	AAAAA	Private
970 B	BBBBB	Public
971 C	CCCCC	Private
972

973

974

Row Type II - Institutional Unit Assets and Liabilities

975

<i>I.U. ID</i>	<i>Ref.Date</i>	<i>Assets</i>	<i>Liabilities</i>
A	2013	1000	800
A	2014	1050	750
B	2013	1200	900
B	2014	1300	950
C	2013	750	900
C	2014	800	850
...

976

977

978

979

980

981

982

983

984 Each of these two tables corresponds to a mathematical function and can be represented like
985 in the first example above. Therefore, they would be 2 distinct logical Data Sets according to
986 the VTL IM, even if stored in the same physical table.

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

989

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

991 Data Structure 1:

- 992 • I.U. ID (Identifier Component)
- 993 • I.U. Name (Measure Component)
- 994 • I.U. Sector (Measure Component)

995

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

997 Data Structure 2:

- 998 • I.U. ID (Identifier Component)
- 999 • Reference Date (Identifier Component)
- 1000 • Assets (Measure Component)
- 1001 • Liabilities (Measure Component)

1002

1003 These examples clarify the meaning of “logical” table or Data Set in VTL, that is a set of data
1004 that can be considered as the extensional form of a mathematical function, whichever
1005 technical format is used, regardless it is persistent or not and, in case, wherever it is stored.

1006 In the example above, one physical data set corresponds to more than one logical VTL Data
1007 Sets, with a 1 to many correspondence. In the general case, between physical and logical data
1008 sets there can be any correspondence (1 to 1, 1 to many, many to 1, many to many).

1009

1010 **The data artefacts**

1011 The list of the VTL artefacts related to the manipulation of the data is given here, together
1012 with the information that the VTL may need to know about them¹².

1013 For the sake of simplicity, the names of the artefacts can be abbreviated in the VTL manuals
1014 (in particular the parts of the names shown between parentheses can be omitted).

1015 As already mentioned, this list provides an abstract view of the core metadata needed for the
1016 manipulation of the data structures but leaves out implementation and operational aspects.
1017 For example, textual descriptions of the artefacts are left out, as well as any specification of
1018 temporal validity of the artefacts, procedural metadata (specification of the way data are
1019 processed, i.e., collected, stored, validated, calculated/estimated, disseminated ...) and so on.
1020 In order to support real systems, the implementers can conveniently adjust this model to their
1021 environments and integrate it by adding additional metadata (e.g. other properties of the
1022 artefacts, other classes of artefacts, other relationships among artefacts ...).

1023 **Data Set**

1024 *Data Set name* *name of the Data Set*

1025 *Data Structure name* *reference to the data structure of the Data Set*

1026 **Data Structure**

1027 *Data Structure name* *name of the Data Structure (the Structure Components are*
1028 *specified in the following artefact)*

1029 **(Data) Structure Component**

1030 *Data Structure name* *the data structure which the Data Structure Component*
1031 *belongs to*

1032 *Component name* *the name of the Component*

1033 *Component Role* *IDENTIFIER or MEASURE or ATTRIBUTE (or also VIRAL*
1034 *ATTRIBUTE if the automatic propagation is supported)*

1035 *Represented Variable* *the Represented Variable which defines the Component (see*
1036 *also below)*

1037

1038 The Data Points have the same information structure as the Data Sets they belong to, in fact
1039 they form the extensions of the relevant Data Sets; VTL does not require to define them
1040 explicitly.

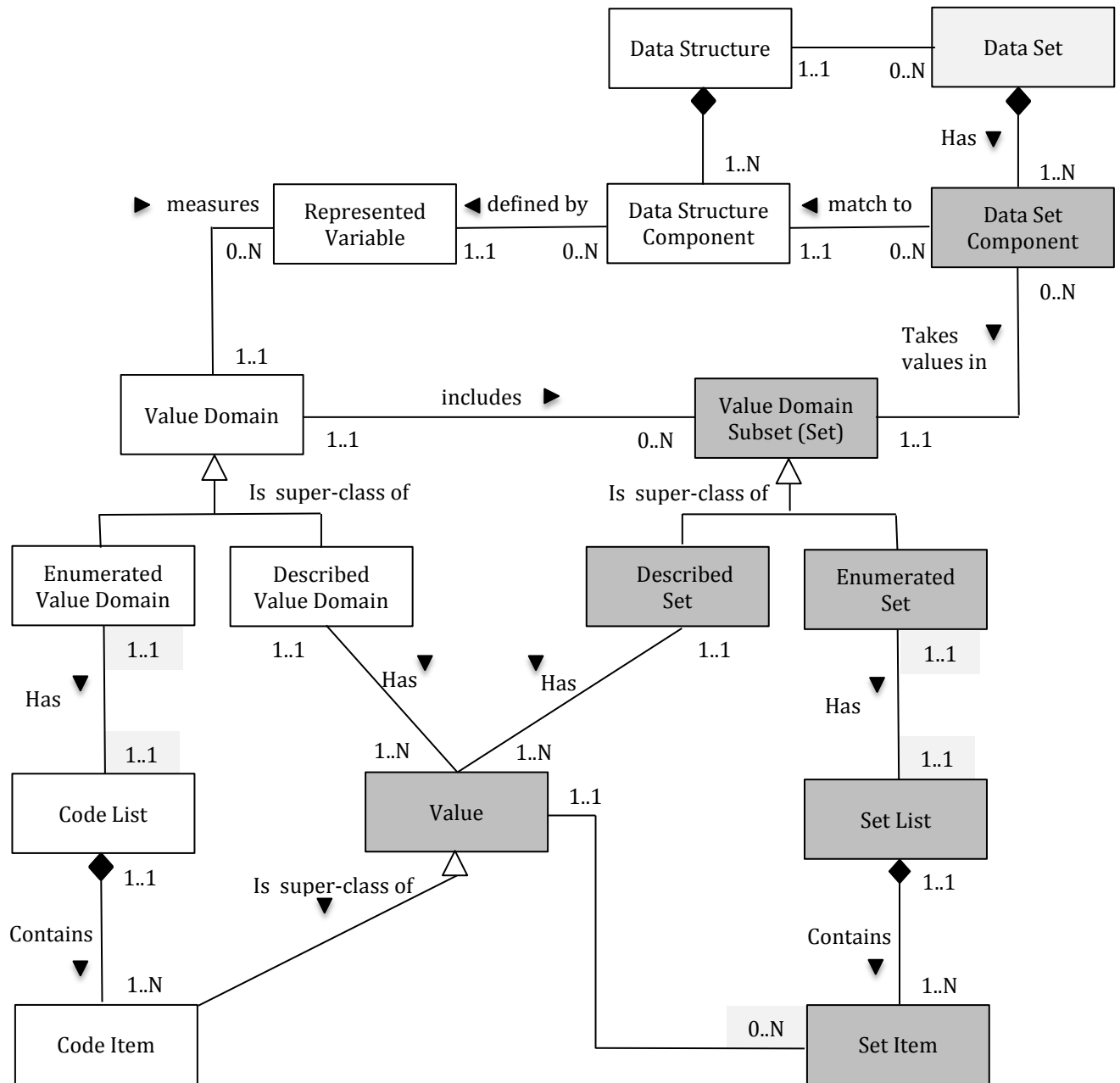
1041

¹² For example, for ensuring correct operations, the knowledge of the Data Structure of the input Data Sets is essential at parsing time, in order to check the correctness of the VTL expression and determine the Data Structure of the result, and at execution time to perform the calculations

1042 **Generic Model for Variables and Value Domains**

1043 This Section provides a formal model for the Variables, the Value Domains, their Values and
 1044 the possible (Sub)Sets of Values. These artefacts can be referenced in the definition of the VTL
 1045 Data Structures and as parameters of some VTL Operators.

1046 **Variable and Value Domain model diagram**



1076 White box: same as in GSIM 1.1
 1077 Light grey: similar to GSIM 1.1
 1078 Dark grey: additional detail (in respect to GSIM 1.1)

1079

1080 Explanation of the Diagram

1081 Even in the case of Variables and Value Domains, the GSIM artefacts are used as much as
1082 possible. The differences are mainly due to the fact that GSIM does not distinguish explicitly
1083 between Value Domains and their (Sub)Sets, while in the VTL IM this is made more explicit in
1084 order to allow different Data Set Components relevant to the same aspect of the reality (e.g.
1085 the geographic area) to share the same Value Domain and, at the same time, to take values in
1086 different Subsets of it. This is essential for VTL for several operations and in particular for
1087 validation purposes. For example, it may happen that the same Represented Variable, say the
1088 “place of birth”, in a Data Set takes values in the Set of the European Counties, in another one
1089 takes values in the set of the African countries, and so on, even at different levels of details
1090 (e.g. the regions, the cities). The definition of the exact Set of Values that a Data Set
1091 Component can take may be very important for VTL, in particular for validation purposes.
1092 The specification of the Set of Values that the Data Set Components may assume is equivalent,
1093 on the mathematical plane, to the specification of the domain and the co-domain of the
1094 mathematical function corresponding to the Data Set.

1095 **Data Set:** see the explanation given in the previous section (Generic Model for Data and their
1096 structures).

1097 **Data Set Component:** a component of the Data Set, which matches with just one Data
1098 Structure Component of the Data Structure of such a Data Set and takes values in a (sub)set of
1099 the corresponding Value Domain¹³; this (sub)set of allowed values may either coincide with
1100 the set of all the values belonging to the Value Domain or be a proper subset of it. In respect to
1101 a Data Structure Component, a Data Set Component bears the important additional
1102 information of the set of allowed values of the Component, which can be different Data Set by
1103 Data Set even if their data structure is the same.

1104 **Data Structure:** a Data Structure; see the explanation already given in the previous section
1105 (Generic Model for Data and their structures).

1106 **Data Structure Component:** a component of a Data Structure; see the explanation already
1107 given in the previous section (Generic Model for Data and their structures). A Data Structure
1108 Component is defined by a Represented Variable.

1109 **Represented Variable:** a characteristic of a statistical population (e.g. the country of birth)
1110 represented in a specific way (e.g. through the ISO code). This artefact is the same as in GSIM.
1111 A represented variable may take value in (or may be measured by) just one Value Domain.

1112 **Value Domain:** the domain of allowed values for one or more represented variables. This
1113 artefact is very similar to the corresponding artefact in GSIM. Because of the distinction
1114 between Value Domain and its Value Domain Subsets, a Value Domain is the wider set of
1115 values that can be of interest for representing a certain aspect of the reality like the time, the
1116 geographical area, the economic sector and so on. As for the mathematical meaning, a Value
1117 Domain is meant to be the representation of a “space of events” with the meaning of the

¹³This is the Value Domain which measures the Represented Variable, which defines the Data Structure Component, which the Data Set Component matches to.

1118 probability theory¹⁴. Therefore, a single Value of a Value Domain is a representation of a
1119 single “event” belonging to this space of events.

1120 **Described Value Domain:** a Value Domain defined by a criterion (e.g. the domain of
1121 the positive integers). This artefact is the same as in GSIM.

1122 **Enumerated Value Domain:** a Value Domain defined by enumeration of the allowed
1123 values (e.g. domain of ISO codes of the countries). This artefact is the same as in GSIM.

1124 **Code List:** the list of all the Code Items belonging to an enumerated Value Domain,
1125 each one representing a single “event” with the meaning of the probability theory. As
1126 for its mathematical meaning, this list is unique for a Value Domain, cannot contain
1127 repetitions (each Code Item can be present just once) and cannot contain ambiguities
1128 (each Code Item must have a univocal meaning, i.e., must represent a single event of
1129 the space of the events). This artefact is the same as in GSIM except for the
1130 multiplicity of the relationship with the Enumerated Value Domain which is 1:1. In fact
1131 like it happens for the Data Set, the VTL considers the Code List as an artefact at a
1132 logical level, corresponding to its mathematical meaning. A logical VTL Code List,
1133 however, may be obtained as the composition of more physical lists of codes if needed:
1134 the mapping between the logical and the physical lists is out of scope of this document
1135 and is left to the implementations, provided that the basic conceptual properties of the
1136 VTL Code List are ensured (unicity, no repetitions, no ambiguities). In practice, as for
1137 the VTL IM, the Code List artefact matches 1:1 with the Enumerated Value Domain
1138 artefact, therefore they can be considered as the same artefact.

1139 **Code Item:** an allowed Value of an enumerated Value Domain. A Code Item is the association
1140 of a Value with the relevant meaning (called “category” in GSIM). An example of Code Item is a
1141 single country ISO code (the Value) associated to the country it represents (the category). As
1142 for the mathematical meaning, a Code Item is the representation of an “event” of a space of
1143 events (i.e. the relevant Value Domain), according to the notions of “event” and “space of
1144 events” of the probability theory (see also the note above).

1145 **Value:** an allowed value of a Value Domain. Please note that on a logical / mathematical level,
1146 both the Described and the Enumerated Value Domains contain Values, the only difference is
1147 that the Values of the Enumerated Value Domains are explicitly represented by enumeration,
1148 while the Values of the Described Value Domains are implicitly represented through a
1149 criterion.

1150

1151 The following artefacts are aimed at representing possible subsets of the Value Domains. This
1152 is needed for validation purposes, because very often not all the values of the Value Domain
1153 are allowed in a Data Structure Component, but only a subset of them (e.g. not all the
1154 countries but only the European countries). This is needed also for transformation purposes,

¹⁴ 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. Instead an “event” is a set of results (going back to the example of the geographic location, the event “Europe” is the set of points of the European territory and more in general an “event” corresponds to a “geographical area”). The “space of events” is the space of all the possible “events” (in the example, the space of the geographical areas).

1155 for example to filter the Data Points according to a subset of Values of a certain Data Structure
1156 Component (e.g. extract only the European Countries from some data relevant to the World
1157 Countries) . Although this detail does not exist in GSIM, these artefacts are compliant with the
1158 GSIM artefacts described above, aimed at representing the Value Domains:

1159 **Value Domain Subset** (or simply **Set**): a subset of Values of a Value Domain. This artefact
1160 does not exist in GSIM, however it is compliant with the GSIM Value Domain. Hereinafter a
1161 Value Domain Subset is simply called **Set**, because it can be any set of Values belonging to the
1162 Value Domain (even the set of all the values of the Value Domain).

1163 **Described Value Domain Subset** (or simply **Described Set**): a described (defined by
1164 a criterion) subset of Values of a Value Domain (e.g. the countries having more than
1165 100 million inhabitants, the integers between 1 and 100). This artefact does not exist
1166 in GSIM, however it is compliant with the GSIM Described Value Domain.

1167 **Enumerated Value Domain Subset** (or simply **Enumerated Set**): an enumerated
1168 subset of a Value Domain (e.g. the enumeration of the European countries). This
1169 artefact does not exist in GSIM, however it is compliant with the GSIM Enumerated
1170 Value Domain.

1171 **Set List**: the list of all the Values belonging to an Enumerated Set (e.g. the list of the ISO
1172 codes of the European countries), without repetitions (each Value is present just once).
1173 As obvious, these Values must belong to the Value Domain of which the Set is a subset.
1174 This artefact does not exist in GSIM, however, it is compliant with the Code List in GSIM
1175 which has a similar role. The Set List enumerates the Values contained in the Set (e.g.
1176 the European country codes), without the associated categories (e.g. the names of the
1177 countries), because the latter are already maintained in the Code List / Code Items of
1178 the relevant Value Domain (which enumerates all the possible Values with the
1179 associated categories). In practice, as for the VTL IM, the Set List artefact coincides 1:1
1180 with the Enumerated Set artefact, therefore they can be considered as the same
1181 artefact.

1182 **Set Item**: an allowed Value of an enumerated Set. The Value must belong to the same Value
1183 Domain the Set belongs to. Each Set Item refers to just one Value and just one Set. A Value can
1184 belong to any number of Sets. A Set can contain any number of Values.

1185 **Relations and operations between Code Items**

1186 The VTL allows the representation of logical relations between Code Items, considered as
1187 events of the probability theory and belonging to the same enumerated Value Domain (space
1188 of events). The VTL artefact that allows expressing the Code Item Relations is the Hierarchical
1189 Ruleset, which is described in the reference manual.

1190 As already explained, each Code Item is the representation of an event, according to the
1191 notions of “event” and “space of events” of the probability theory. The relations between Code
1192 Items aim at expressing the logical implications between the events of a space of events (i.e. in
1193 a Value Domain). The occurrence of an event, in fact, may imply the occurrence or the non-
1194 occurrence of other events. For example:

- 1195 • The event UnitedKingdom implies the event Europe (e.g. if a person lives in UK he/she
1196 also lives in Europe), meaning that the occurrence of the former implies the occurrence
1197 of the latter. In other words, the geo-area of UK is included in the geo-area of the
1198 Europe.

- 1199 • The events Belgium, Luxembourg, Netherlands are mutually exclusive (e.g. if a person
- 1200 lives in one of these countries he/she does not live in the other ones), meaning that the
- 1201 occurrence of one of them implies the non-occurrence of the other ones (Belgium AND
- 1202 Luxembourg = impossible event; Belgium AND Netherlands = impossible event;
- 1203 Luxembourg AND Netherlands = impossible event). In other words, these three geo-
- 1204 areas do not overlap.
- 1205 • The occurrence of one of the events Belgium, Netherlands or Luxembourg (i.e. Belgium
- 1206 OR Netherlands OR Luxembourg) implies the occurrence of the event Benelux (e.g. if a
- 1207 person lives in one of these countries he/she also lives in Benelux) and vice-versa (e.g.
- 1208 if a person lives in Benelux, he/she lives in one of these countries). In other words, the
- 1209 union of these three geo-areas coincides with the geo-area of the Benelux.

1210 The logical relationships between Code Items are very useful for validation and
 1211 transformation purposes. Considering for example some positive and additive data, like for
 1212 example the population, from the relationships above it can be deduced that:

- 1213 • The population of United Kingdom should be lower than the population of Europe.
- 1214 • There is no overlapping between the populations of Belgium, Netherlands and
- 1215 Luxembourg, so that these populations can be added in order to obtain aggregates.
- 1216 • The sum of the populations of Belgium, Netherlands and Luxembourg gives the
- 1217 population of Benelux.

1218 A **Code Item Relation** is composed of two members, a 1st (left) and a 2nd (right) member. The
 1219 envisaged types of relations are: “is equal to” (=), “implies” (<), “implies or is equal to” (<=),
 1220 “is implied by” (>), and “is implied by or is equal to” (>=). “Is equal to” means also “implies
 1221 and is implied”. For example:

1222 UnitedKingdom < Europe means (UnitedKingdom implies Europe)

1223 In other words, this means that if a point of space belongs to United Kingdom it also
 1224 belongs to Europe.

1225 The left members of a Relation are single Code Items. The right member can be either a single
 1226 Code Item, like in the example above, or a logical composition of Code Items: these are the
 1227 **Code Item Relation Operands**. The logical composition can be defined by means of
 1228 Operators, whose goal is to compose some Code Items (events) in order to obtain another
 1229 Code Item (event) as a result. In this simple algebra, two operators are envisaged:

- 1230 • the logical OR of mutually exclusive Code Items, denoted “+”, for example:

1231 Benelux = Belgium + Luxembourg + Netherlands

1232 This means that if a point of space belongs to Belgium OR Luxembourg OR Netherlands
 1233 then it also belongs to Benelux and that if a point of space belongs to Benelux then it
 1234 also belongs either to Belgium OR to Luxembourg OR to Netherlands (disjunction). In
 1235 other words, the statement above says that territories of Belgium, Netherlands and
 1236 Luxembourg are non-overlapping and their union is the territory of Benelux.
 1237 Consequently, as for the additive measures (and being equal the other possible
 1238 Identifiers), the sum of the measure values referred to Belgium, Luxembourg and
 1239 Netherlands is equal to the measure value of Benelux.

- 1240 • the logical complement of an implying Code Item in respect to another Code Item
 1241 implied by it, denoted “-”, for example:

1242 EUwithoutUK = EuropeanUnion - UnitedKingdom

1243 In simple words, this means that if a point of space belongs to the European Union and
1244 does not belong to the United Kingdom, then it belongs to EUwithoutUK and that if a
1245 point of space belongs to EUwithoutUK then it belongs to the European Union and not
1246 to the United Kingdom. In other words, the statement above says that territory of the
1247 United Kingdom is contained in the territory of the European Union and its
1248 complement is the territory of EUwithoutUK. As a consequence, considering a positive
1249 and additive measure (and being equal the other possible Identifiers), the difference of
1250 the measure values referred to EuropeanUnion and UnitedKingdom is equal to the
1251 measure value of EUwithoutUK.

1252 Please note that the symbols “+” and “-“ do not denote the usual operations of sum and
1253 subtraction, but logical operations between Code Items seen as events of the probability
1254 theory. In other words, two or more Code Items cannot be summed or subtracted to obtain
1255 another Code Item, because they are events (and not numbers), and therefore they can be
1256 manipulated only through logical operations like “OR” and “Complement”.

1257 Note also that the “+” also acts as a declaration that all the Code Items denoted by “+” are
1258 mutually exclusive (i.e. the corresponding events cannot happen at the same time), as well as
1259 the “-“ acts as a declaration that all the Code Items denoted by “-“ are mutually exclusive.
1260 Furthermore, the “-“ acts also as a declaration that the relevant Code item implies the result of
1261 the composition of all the Code Items denoted by the “+”.

1262 At intuitive level, the symbol “+” means “with” (Benelux = Belgium *with* Luxembourg *with*
1263 Netherland) while the symbol “-“ means “without” (EUwithoutUK = EuropeanUnion *without*
1264 UnitedKingdom).

1265 When these relations are applied to additive numeric Measures (e.g. the population relevant
1266 to geographical areas), they allow to obtain the Measure Values of the left member Code Items
1267 (i.e. the population of Benelux and EUwithoutUK) by summing or subtracting the Measure
1268 Values relevant to the component Code Items (i.e. the population of Belgium, Luxembourg and
1269 Netherland in the former case, EuropeanUnion and UnitedKingdom in the latter). This is why
1270 these logical operations are denoted in VTL through the same symbols as the usual sum and
1271 subtraction. Please note also that this is valid whichever the Data Set and the additive
1272 Measure are (provided that the possible other Identifiers of the Data Set Structure have the
1273 same Values).

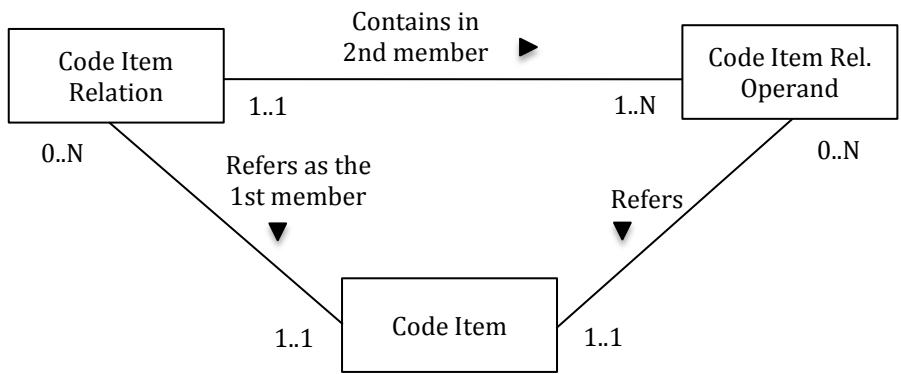
1274 These relations occur between Code Items (events) belonging to the same Value Domain
1275 (space of events). They are typically aimed at defining aggregation hierarchies, either
1276 structured in levels (classifications), or without levels (chains of free aggregations) or a
1277 combination of these options. These hierarchies can be recursive, i.e. the aggregated Code
1278 Items can in their turn be the components of more aggregated ones, without limitations to the
1279 number of recursions.

1280 For example, the following relations are aimed at defining the continents and the whole world
1281 in terms of individual countries:

- 1282 • World = Africa + America + Asia + Europe + Oceania
- 1283 • Africa = Algeria + ... + Zimbabwe
- 1284 • America = Argentina + ... + Venezuela
- 1285 • Asia = Afghanistan + ... + Yemen

- 1286 • Europe = Albania + ... + Vatican City
- 1287 • Oceania = Australia + ... + Vanuatu

1288 A simple model diagram for the Code Item Relations and Code Item Relation Operands is the
 1289 following:



1298 This diagram tells that a Code Item Relation has a first and a second member. The first
 1299 member (the left one) refers to just one Code Item, the second member (the right one) may
 1300 refer to one or more Code Item Relation Operands; each Code Item Relation Operand refers
 1301 to just one Code Item.

1302 **Conditioned Code Item Relations**

1303 The Code Items (coded events) of a Code Item Relation can be conditioned by the Values
 1304 (events) of other Value Domains (spaces of events). Both the Code Items belonging to the first
 1305 and the second member of the Relation can be conditioned.

1306 A common case is the conditioning relevant to the reference time, which allows to express the
 1307 historical validity of a Relation (see also the section about the historical changes below). For
 1308 example, the European Union (EU) changed its composition in terms of countries many times,
 1309 therefore the Code Item Relationship between EU and its component countries depends on
 1310 the reference time, i.e. is conditioned by the Values of the “reference time” Value Domain.

1311 The VTL allows to express the conditionings by means of Boolean expressions which make
 1312 reference to the Values of the conditioning Value Domains (for more details, see the
 1313 Hierarchical Rulesets in the Reference Manual).

1314 **The historical changes**

1315 The changes in the real world may induce changes in the artefacts of the VTL-IM and in the
 1316 relationships between them, so that some definitions may be considered valid only with
 1317 reference to certain time values. For example, the birth of a new country as well as the split
 1318 or the merge of existing countries in the real world would induce changes in the Code Items
 1319 belonging to the Geo Area Value Domain, in the composition of the relevant Sets, in the
 1320 relationships between the Code Items and so on. The same may obviously happen for other
 1321 Value Domains.

1322 A correct representation of the historical changes of the artefacts is essential for VTL, because
 1323 the VTL operations are meant to be consistent with these historical changes, in order to
 1324 ensure a proper behaviour in relation to each time. With regard to this aspect, VTL must face a

1325 complex environment, because it is intended to work also on top of other standards, whose
1326 assumptions for representing historical changes may be heterogeneous. Moreover, different
1327 institutions may use different conventions in different systems.

1328 Naturally, adopting a common convention for representing the historical changes of the
1329 artefacts would be a good practice, because the definitions made by different bodies would be
1330 given through the same methodology and therefore would be easily comparable one another.
1331 In practice, however, different conventions are already in place and have to be taken into
1332 account, because there can also be strong motivations to maintain them. For this reason, the
1333 VTL does not impose any definite representation for the historical changes and leaves users
1334 free of maintaining their own conventions, which are considered as part of the data content to
1335 be processed rather than of the language.

1336 As a matter of fact, the VTL-IM intentionally does not include any mechanism for representing
1337 historical changes and needs to be properly integrated to this purpose. This aspect is left to
1338 the standards and the institutions adopting VTL and the implementers of VTL systems, which
1339 can adapt and enrich the VTL-IM as needed.

1340 Even if presented here for association of ideas with the relations between Code Items, whose
1341 temporal dependency is intuitive, these considerations about the temporal validity of the
1342 definitions are valid in general.

1343 Moreover, as already mentioned, the possibility of integrating the VTL-IM with additional
1344 metadata is needed also for other purposes, and not only for dealing with the temporal
1345 validity.

1346 It is appropriate here to highlight some relationships between the VTL artefacts and some
1347 possible temporal conventions, because this can guide VTL implementers in extending the
1348 VTL-IM according to their needs.

1349 First, we want to distinguish between two main temporal aspects: the so-called validity time
1350 and operational time. Validity time is the time during which a definition is assumed to be true
1351 as an abstraction of the real world (for example, Estonia belongs to EU “from 1st May 2004 to
1352 current date”). Operational time is the time period during which a definition is available in the
1353 processing system and may produce operational effects. The following considerations refers
1354 only to the former.

1355 The **assignment of identifiers to the abstractions of the real world** is strictly related to the
1356 possible basic temporal assumptions. Two main options can be considered:

1357 a) The same identifier is assigned to the abstraction even if some aspects of such an
1358 abstraction change in time. For example, the identifier EU is assigned to the European
1359 Union even if the participant countries change. Under this option, a single identifier
1360 (e.g. EU) is used to represent the whole history of an abstraction, following the
1361 intuitive conceptualization in which abstractions are identified independently of time
1362 and maintain the same identity even if they change with time. The variable aspects of
1363 an abstraction are therefore described by specifying their validity periods (for
1364 example, the participation of Estonia in the EU can be specified through the relevant
1365 start and end dates).

1366 b) Different Identifiers are assigned to the abstraction when some aspects of the
1367 abstraction change in time. For example, more Identifiers (e.g. EU1, ... , EU9) represent
1368 the European Union, one for each period during which its participant countries remain

1369 stable. This option is based on the conceptualization in which the abstractions are
1370 identified in connection with the time period in which they do not change, so that an
1371 Code Item (e.g. EU1) corresponds to an abstraction (e.g. the European Union) only for
1372 the time period in which the abstraction remain stable (e.g. EU1 represents the
1373 European Union from when it was created by the founder countries, to the first time it
1374 changed composition). An example of adoption of this option b) is the common
1375 practice of giving versions to Code Lists or Code Items for representing time changes
1376 (e.g. EUv1, ... , EUv9 where v=version), being each version assumed as invariable.

1377 As a consequence, the general assumptions of VTL for the representation of the historical
1378 changes are the following:

- 1379 • The choice of adopting the options described above is left to the implementations.
- 1380 • The VTL Identifiers are different depending on the two options above; for example in
1381 the option a) there would exist one Identifier for the European Union (e.g. EU) while in
1382 the option b) there would exist many different Identifiers, corresponding to the
1383 different versions of the European Union (e.g. EU1, ... , EU9).
- 1384 • If the Code Items are versioned for managing temporal changes (option b), the version
1385 is considered to be part of the VTL univocal identifier of the Code Item, therefore
1386 different versions are equivalent to different Code Items. As explained above, in fact,
1387 the European Union would be represented by many Code Items (e.g. EUv1, ... , EUv9).
1388 The same applies if the Code Items are versioned by means of dates (e.g. start/end
1389 dates ...) or other conventions instead than version numbers. As obvious, the temporal
1390 validity of EUv1, ... , EUv9, if represented, should not overlap.

1391 The implementers of VTL systems can add the temporal validity (through validity dates or
1392 versions) to any class of artefacts or relations of the VTL-IM (as well as any other additional
1393 characteristic useful for the implementation, like the textual descriptions of the artefacts or
1394 others). If the temporal validity is not added, the occurrences of the class are assumed to be
1395 valid “ever”.

1396 **The Variables and Value Domains artefacts**

1397 The list of the VTL artefacts related to Variables and Value Domains is given here, together
1398 with the information that the VTL need to know about them. For the sake of simplicity, the
1399 names of some artefacts are often abbreviated in the VTL manuals (in particular the parts of
1400 the names shown between parentheses can be omitted).

1401 As already mentioned, this model provides an abstract view of the core metadata supporting
1402 the definition of the data structures but leaves out implementation and operational aspects.
1403 For example, the textual descriptions of the artefacts are left out, as well as the specification of
1404 the temporal validity of the artefacts, the procedural metadata (the specification of the way
1405 data are processed, i.e. collected, stored, validated, calculated/estimated, disseminated ...) and
1406 so on. In order to support real systems, the implementers can conveniently adjust this model
1407 and integrate it by adding other metadata (e.g. other properties of the artefacts, other classes
1408 of artefacts, other relationships among artefacts ...).

1409
1410 ***(Represented) Variable***
1411 *Variable name* *name of the Represented Variable*

1412	<i>Value Domain name</i>	<i>reference to the Value Domain which measures the Variable,</i>
1413		<i>i.e. in which the Variable takes values</i>
1414		
1415	<i>(Data Set) Component</i>	
1416	<i>Data Set name</i>	<i>the Data set which the Component belongs to</i>
1417	<i>Component name</i>	<i>the name of the Component</i>
1418	<i>(Sub) Set name</i>	<i>reference to the (sub)Set containing the allowed values for</i>
1419		<i>the Component</i>
1420		
1421	<i>Value Domain</i>	
1422	<i>Value Domain name</i>	<i>name of the Value Domain</i>
1423	<i>Value Domain sub-class</i>	<i>if it is an Enumerated or Described Value Domain</i>
1424	<i>Basic Scalar Type</i>	<i>the basic scalar type of the Values of the Value Domain, for</i>
1425		<i>example string, number ... and so on (see also the section</i>
1426		<i>“VTL data types”)</i>
1427	<i>Value Domain Criterion</i>	<i>a criterion for restricting the Values of a basic scalar type,</i>
1428		<i>for example by specifying a max length of the</i>
1429		<i>representation, an upper or/and a lower value, and so on</i>
1430		
1431	<i>Code List</i>	<i>this artefact is comprised in the previous one, in fact it</i>
1432		<i>corresponds one to one to the enumerated Value Domain</i>
1433		<i>(see above)</i>
1434		
1435	<i>Value</i>	<i>this artefact has no explicit representation, because the</i>
1436		<i>Values of described Value Domains are not represented by</i>
1437		<i>definition, while the Values of the enumerated Value</i>
1438		<i>Domains are represented via the Code Item artefact (see</i>
1439		<i>below)</i>
1440		
1441	<i>Code Item</i>	<i>this artefact specifies the Code Items of the Enumerated</i>
1442		<i>Value Domains</i>
1443	<i>Value Domain name</i>	<i>the Value Domain which the Value belongs to</i>
1444	<i>Value</i>	<i>the univocal name of the Value within the Value Domain it</i>
1445		<i>belongs to</i>
1446		
1447	<i>(Value Domain Sub)Set</i>	
1448	<i>Value Domain name</i>	<i>the Value Domain which the set belongs to</i>
1449	<i>Set name</i>	<i>the name of the Set, which must be univocal within the</i>
1450		<i>Value Domain</i>

1451	<i>Set sub-class</i>	<i>if it is an Enumerated or Described Set</i>
1452	<i>Set Criterion</i>	<i>a criterion for identifying the Values belonging to the Set</i>
1453		
1454	Set List	<i>this artefact is comprised in the previous one, in fact it</i>
1455		<i>corresponds one to one to the enumerated Set</i>
1456		
1457	Set Item	<i>this artefact specifies the Code Items of the Enumerated Sets</i>
1458	<i>Value Domain name</i>	<i>reference to the Value Domain which the Set and the Value</i>
1459		<i>belongs to</i>
1460	<i>Set name</i>	<i>the Set that contains the Value</i>
1461	<i>Value</i>	<i>Value element of the Set</i>
1462		
1463	Code Item Relation	
1464	<i>1stMember Domain name</i>	<i>Value Domain of the first member of the Relation; e.g.</i>
1465		<i>Geo_Area</i>
1466	<i>1stMember Value</i>	<i>the first member of the Relation; e.g. Benelux</i>
1467	<i>1stMember Composition</i>	<i>conventional name of the composition method, which</i>
1468		<i>distinguishes possible different compositions methods</i>
1469		<i>related to the same first member Value. It must be univocal</i>
1470		<i>within the 1stMember. Not necessarily it has to be</i>
1471		<i>meaningful, it can be simply a progressive number ; e.g. "1"</i>
1472	<i>Relation Type</i>	<i>type of relation between the first and the second member,</i>
1473		<i>having as possible values =, <, <=, >, >=</i>
1474		
1475	Code Item Relation Operand	
1476	<i>1stMember Domain name</i>	<i>Value Domain of the first member of the Relation; e.g.</i>
1477		<i>Geo_Area</i>
1478	<i>1stMember Value</i>	<i>the first member of the Relation; e.g. Benelux</i>
1479	<i>1stMember Composition</i>	<i>see the description already given above</i>
1480	<i>2ndMember Value</i>	<i>an operand of the Relation; e.g. Belgium]</i>
1481	<i>Operator</i>	<i>the operator applied on the 2ndMember Value, it can be "+"</i>
1482		<i>or "-"; the default is "+"</i>
1483		

1484 Generic Model for Transformations

1485 The purpose of this section is to provide a formal model for describing validation and
1486 transformation of data.

1487 A Transformation is assumed to be an algorithm to produce a new model artefact (typically a
1488 Data Set) starting from existing ones. It is also assumed that the data validation is a particular
1489 case of transformation, therefore the term “transformation” is meant to be more general and
1490 to include the validation case as well.

1491 This model is essentially derived from the SDMX IM¹⁵, as DDI and GSIM do not have an explicit
1492 transformation model at the moment¹⁶. In its turn, the SDMX model for Transformations is
1493 similar in scope and content to the Expression metamodel that is part of the Common
1494 Warehouse Metamodel (CWM)¹⁷ developed by the Object Management Group (OMG).

1495 The model represents the user logical view of the definition of algorithms by means of
1496 expressions. In comparison to the SDMX and CWM models, some technical details are omitted
1497 for the sake of simplicity, including the way expressions can be decomposed in a tree of nodes
1498 in order to be executed (if needed, this detail can be found in the SDMX and CWM
1499 specifications).

1500 The basic brick of this model is the notion of Transformation.

1501 A Transformation specifies the algorithm to obtain a certain artefact of the VTL information
1502 model, which is the result of the Transformation, starting from other existing artefacts, which
1503 are its operands.

1504 Normally the artefact produced through a Transformation is a Data Set (as usual considered
1505 at a logical level as a mathematical function). Therefore, a Transformation is mainly an
1506 algorithm for obtaining derived Data Sets starting from already existing ones.

1507 The general form of a Transformation is the following:

1508 $result \quad assignment_operator \quad expression$

1509 meaning that the outcome of the evaluation of *expression* in the right-hand side is assigned to
1510 the *result of the Transformation* in the left-hand side (typically a Data Set). The assignment
1511 operators are two, “<-” and “:=” (for the assignment to a persistent or a non-persistent
1512 result, respectively). A very simple example of Transformation is:

1513 $D_r <- D_1 \quad (D_r, D_1 \text{ are assumed to be Data Sets})$

1514 In this Transformation, the Data Set D_1 is assigned without changes (i.e. is copied) to D_r ,
1515 which is persistently stored.

1516 In turn, the *expression* in the right-hand side composes some operands (e.g. some input Data
1517 Sets, but also Sets or other artefacts) by means of some operators (e.g. sum, product ...) to
1518 produce the desired results (e.g. the validation outcome, the calculated data).

1519 For example: $D_r := D_1 + D_2 \quad (D_r, D_1, D_2 \text{ are assumed to be Data Sets})$

¹⁵ The SDMX specification can be found at https://sdmx.org/?page_id=5008 (see Section 2 - Information Model, package 13 - “Transformations and Expressions”).

¹⁶ 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 out of the scope of the present document.

¹⁷ This specification can be found at <http://www.omg.org/cwm>.

1520 In this example the measure values of the Data Set D_r are calculated as the sum of the measure
1521 values of the Data Sets D_1 and D_2 , by composing the Data Points having the same Values for
1522 the Identifiers. In this case D_r is not persistently stored.

1523 A validation is intended to be a kind of Transformation. For example, the simple validation
1524 that $D_1 = D_2$ can be made through an “if” operator, with an expression of the type:

1525
$$D_r \quad := \quad \text{if } (D_1 = D_2, \text{ then TRUE, else FALSE})$$

1526 In this case, the Data Set D_r would have a Boolean measure containing the value TRUE if the
1527 validation is successful and FALSE if it is unsuccessful.

1528 These are only fictitious examples for explanation purposes. The general rules for the
1529 composition of Data Sets (e.g. rules for matching their Data Points, for composing their
1530 measures ...) are described in the sections below, while the actual Operators of the VTL and
1531 their behaviours are described in the VTL reference manual.

1532 The *expression* in the right-hand side of a Transformation must be written according to a
1533 formal language, which specifies the list of allowed operators (e.g. sum, product ...), their
1534 syntax and semantics, and the rules for composing the expression (e.g. the default order of
1535 execution of the operators, the use of parenthesis to enforce a certain order ...). The Operators
1536 of the language have Parameters¹⁸, which are the a-priori unknown inputs and output of the
1537 operation, characterized by a given role (e.g. dividend, divisor or quotient in a division).

1538 Note that this generic model does not specify the formal language to be used. As a matter of
1539 fact, not only the VTL but also other languages might be compliant with this specification,
1540 provided that they manipulate and produce artefacts of the information model described
1541 above. This is a generic and formal model for defining Transformations of data through
1542 mathematical expressions, which in this case is applied to the VTL, agreed as the standard
1543 language to define and exchange validation and transformation rules among different
1544 organizations

1545 Also, note that this generic model does not actually specify the operators to be used in the
1546 language. Therefore, the VTL may evolve and may be enriched and extended without impact
1547 on this generic model.

1548 In the practical use of the language, Transformations can be composed one with another to
1549 obtain the desired outcomes. In particular, the result of a Transformation can be an operand
1550 of other Transformations, in order to define a sequence of calculations as complex as needed.

1551 Moreover, the Transformations can be grouped into Transformations Schemes, which are sets
1552 of Transformations meaningful to the users. For example, a Transformation Scheme can be
1553 the set of Transformations needed to obtain some specific meaningful results, like the
1554 validations of one or more Data Sets. A Transformation Scheme is meant to be the smaller set
1555 of Transformations to be executed in the same run.

1556 A set of Transformations takes the structure of a graph, whose nodes are the model artefacts
1557 (usually Data Sets) and whose arcs are the links between the operands and the results of the
1558 single Transformations. This graph is directed because the links are directed from the
1559 operands to the results and is acyclic because it should not contain cycles (like in the
1560 spreadsheets), otherwise the result of the Transformations might become unpredictable.

¹⁸ The term is used with the same meaning of “argument”, as usual in computer science.

1561 The ability of generating this graph is a main feature of the VTL, because the graph documents
 1562 the operations performed on the data, just like a spreadsheet documents the operations
 1563 among its cells.

1564 **Transformations model diagram**

1565

1566

1567

1568

1569

1570

1571

1572

1573

1574

1575

1576

1577

1578

1579

1580

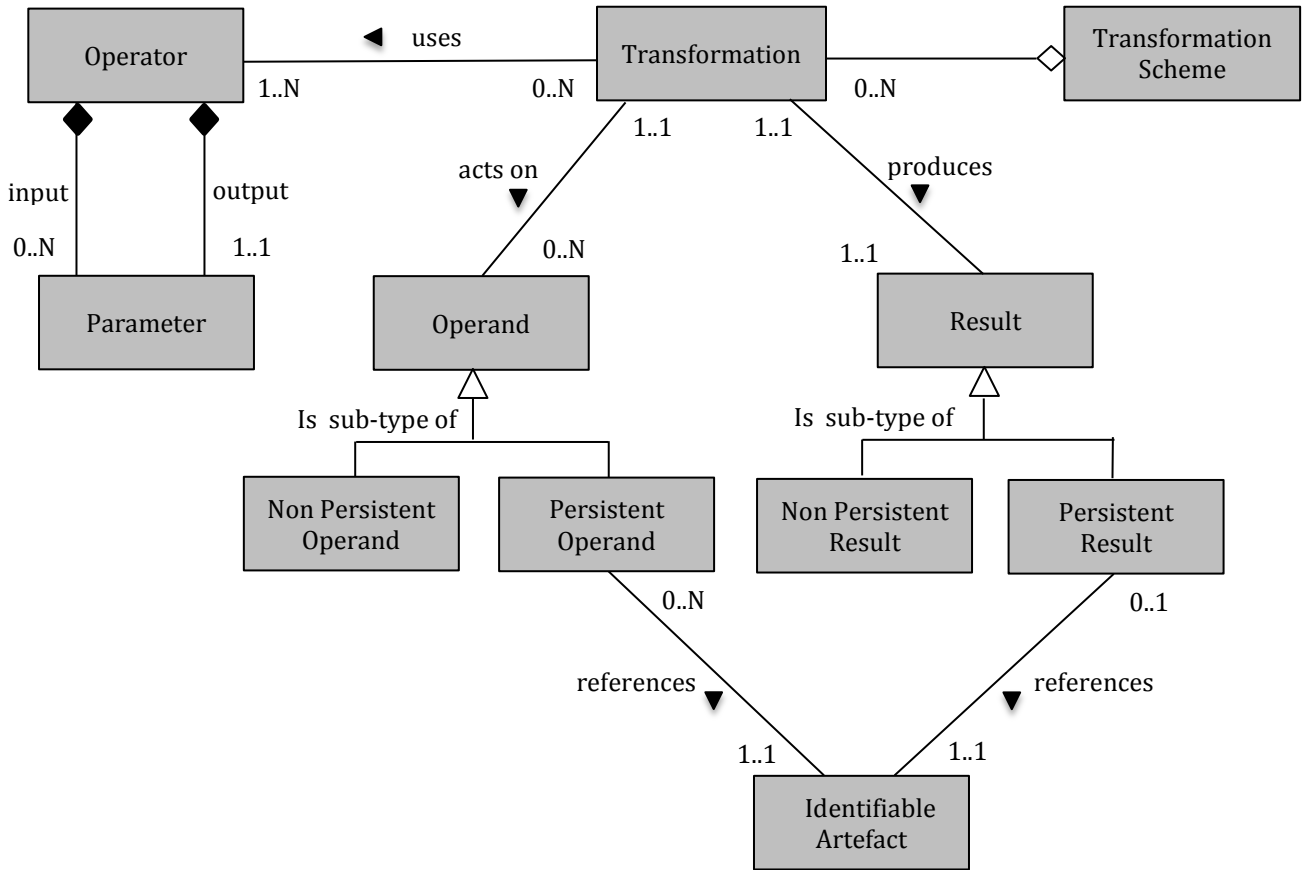
1581

1582

1583

1584

1585



1586 White box: same as in GSIM 1.1
 1587 Dark grey box: additional detail (in respect to GSIM 1.1)

1588

1589 **Explanation of the diagram**

1590 **Transformation:** the basic element of the calculations, which consists of a statement which
 1591 assigns the outcome of the evaluation of an Expression to an Artefact of the Information
 1592 Model;

1593 **Expression:** a finite combination of symbols that is well-formed according to the syntactical
 1594 rules of the language. The goal of an Expression is to compose some Operands in a certain
 1595 order by means of the Operators of the language, in order to obtain the desired result.
 1596 Therefore, the symbols of the Expression designate Operators, Operands and the order of
 1597 application of the Operators (e.g. the parenthesis); an expression is defined as a text string
 1598 and is a property of a Transformation;

1599 **Transformation Scheme:** a set of Transformations aimed at obtaining some meaningful
1600 results for the user (like the validation of one or more Data Sets); the Transformation Scheme
1601 is meant to be the smaller set of Transformations to be executed in the same run and
1602 therefore may also be considered as a VTL program;

1603 **Operator:** the specification of a type of operation to be performed on some Operands (e.g.
1604 sum (+), subtraction (-), multiplication (*), division (/));

1605 **Parameter:** a-priori unknown input or output of an Operator, having a definite role in the
1606 operation (e.g. dividend, divisor or quotient for the division) and corresponding to a certain
1607 type of artefact (e.g. a “Data Set”, a “Data Structure Component” ...), for a deeper explanation
1608 see also the Data Type section below. When an Operator is invoked, the actual input passed in
1609 correspondence to a certain input Parameter, or the actual output returned by the Operator,
1610 is called Argument.

1611 **Operand:** a specific Artefact referenced in the expression as an input (e.g. a specific input
1612 Data Set); a Persistent Operand references a persistent artefact, i.e. an artefact maintained in a
1613 persistent storage, while a Non Persistent Operand references a temporary artefact, which is
1614 produced by another Transformation and not stored.

1615 **Result:** a specific Artefact to which the result of the expression is assigned (e.g. the calculated
1616 Data Set); a Persistent Result is put away in a persistent storage while a Non Persistent Result
1617 is not stored.

1618 **Identifiable Artefact:** a persistent Identifiable Artefact of the VTL information model (e.g. a
1619 persistent Data Set); a persistent artefact can be operand of any number of Transformations
1620 but can be the result of no more than one Transformation.

1621

1622 Examples

1623 Imagine that D_1 , D_2 and D_3 are Data Sets containing information on some goods, specifically:
1624 D_1 the stocks of the previous date, D_2 the flows in the last period, D_3 the current stocks.
1625 Assume that it is desired to check the consistency of the Data Sets using the following
1626 statement:

1627 $D_r \quad := \quad \text{If } ((D_1 + D_2) = D_3, \text{ then “true”, else “false”})$

1628 In this case:

1629 The Transformation may be called “basic consistency check between stocks and flows” and is
1630 formally defined through the statement above.

- | | | |
|------|---|-------------------|
| 1631 | • D_r | is the Result |
| 1632 | • D_1, D_2 and D_3 | are the Operands |
| 1633 | • $\text{If } ((D_1 + D_2) = D_3, \text{ then TRUE, else FALSE})$ | is the Expression |
| 1634 | • “:=”, “If”, “+”, “=” | are Operators |

1635 Each operator has some predefined parameters, for example in this case:

- | | | |
|------|-----------------------------|--|
| 1636 | • input parameters of “+”: | two numeric Data Sets (to be summed) |
| 1637 | • output parameters of “+”: | a numeric Data Sets (resulting from the sum) |
| 1638 | • input parameters of “=”: | two Data Sets (to be compared) |
| 1639 | • output parameter of “=”: | a Boolean Data Set (resulting from the comparison) |

- 1640 • input parameters of “If”: an Expression defining a condition, i.e. $(D_1+D_2)=D_3$
- 1641 • output parameter of “If”: a Data Set (as resulting from the “then”, “else” clauses)

1642

1643 **Functional paradigm**

1644 As mentioned, the VTL follows a functional programming paradigm, which treats
1645 computations as the evaluation of mathematical functions, so avoiding changing-state and
1646 mutable data in the specification of the calculation algorithm. On one side the statistical data
1647 are considered as mathematical functions (first order functions), on the other side the VTL
1648 operators are considered as functions as well (second order functions), applicable to some
1649 data in order to obtain other data.

1650 According to the functional paradigm, the output value of a (second order) function depends
1651 only on the input arguments of the function, is calculated in its entirety and once for all by
1652 applying the function, and cannot be altered or modified once calculated (immutable) unless
1653 the input arguments change.

1654 And in fact the VTL operators, and the expressions built using these operators, specify the
1655 algorithm for calculating the results in their entirety, once for all, and never for updating
1656 them. When some change in the operands occurs (e.g. the input data change), the VTL
1657 assumes that the results are recalculated in their entirety according to the correspondent
1658 expressions¹⁹.

1659 Coherently, a VTL artefact can be result of just one Transformation and cannot be updated by
1660 other Transformations, a Transformation cannot update either its own operands or the result
1661 of other Transformations and the result of a new Transformation is always a new artefact.

1662

1663 **Transformation Consistency**

1664 The Transformation model requires that the Transformations follow some consistency rules,
1665 similar to the ones typical of the spreadsheets; in fact there is a strict analogy between the
1666 generic models of Transformations and spreadsheets.

1667 In this analogy, a VTL artefact corresponds to a non-empty cell of a spreadsheet, a
1668 Transformation to the formula defined in a cell (which references other cells as operands), a
1669 Result to the content of the cell in which the formula is defined ²⁰.

1670 The model artefacts involved in Transformations can be divided into “collected / primary” or
1671 “calculated / derived” ones. The former are original artefacts of the information system, not
1672 result of any Transformation, fed from some external source or by the users (they are
1673 analogous to the spreadsheet cells which are not calculated). The latter are produced as
1674 results of some Transformations (they are analogous to the spreadsheet cells calculated
1675 through a formula).

¹⁹ At the implementation level, which is out of the scope of this document, the update operations are obviously possible

²⁰ The main difference between the two cases is the fact that a cell of a spreadsheet may contain only a scalar value while a VTL artefact may have also a more complex data structure, being typically a Data Set

1676 As already said, a Transformation calculates just one result (“derived” model artefact) and a
1677 result is calculated by just one Transformation. Both “primary” and “derived” model artefacts
1678 can be operands of any number of Transformations. An artefact cannot be operand and
1679 result of the same Transformation.

1680 A Transformation belongs to just one Transformation Scheme, which is analogous to a whole
1681 spreadsheet, in fact it is a set of Transformations executed in the same run and may contain
1682 any number of Transformations in order to produce any number of results.

1683 Because a “derived” model artefact is produced by just one Transformation and a
1684 Transformation belongs to just one Transformation Scheme, it follows also that a “derived”
1685 model artefact is produced in the context of just one Transformation Scheme.

1686 The operands of a Transformation may come either from the same Transformation Scheme
1687 which the Transformation belongs to or from other ones.

1688 Within a Transformation Scheme, it can be built a graph of the Transformations by assuming
1689 that each model artefact is a node and each Transformation is a set of arcs, starting from the
1690 Operand nodes and ending in the Result node;

1691 This graph must be a directed acyclic graph (DAG): in particular, each arc is oriented from the
1692 operand to the result; the absence of cycles makes it possible to unambiguously calculate the
1693 “derived” nodes by applying the Transformations by following the topological order of the
1694 graph.

1695 Therefore, like in the spreadsheet, not necessarily the Transformations are performed in the
1696 same order as they are written, because the order of execution depends on their input-output
1697 relationships (a Transformation which calculates a result that is operand of other
1698 Transformations must be executed first).

1699 In the analogy between VTL and a spreadsheet, the correspondences would be the following:
1700

- 1701 • VTL model artefact \leftrightarrow non-empty cell of a spreadsheet;
- 1702 • VTL “collected / primary” model artefact \leftrightarrow non-empty cell of a spreadsheet whose
1703 value is fed from an external source or by the user;
- 1704 • A “calculated / derived” model artefact \leftrightarrow a non-empty cell of a spreadsheet
1705 whose value is calculated by a formula;
- 1706 • A VTL Transformation \leftrightarrow A spreadsheet formula assigned to a cell
- 1707 • a VTL Transformation Scheme \leftrightarrow A whole spreadsheet

1708

1710 The possible operations in VTL depend on the data types of the artefacts. For example,
1711 numbers can be multiplied but text strings cannot.

1712 When an Operator is invoked, for each (formal) input Parameter, an actual argument
1713 (operand) is passed to the Operator, and for the output Parameter, an actual argument
1714 (result) is returned by the Operator. The data type of the argument must comply with the
1715 allowed data types of the corresponding Parameter (the allowed data types of each Parameter
1716 for each Operator are specified in the Reference Manual).

1717 Every possible argument for a VTL Operator (with special attention to artefacts of the
1718 Information Model, e.g., Values, Sets, Data Sets) must be typed and such type deterministically
1719 inferable.

1720 In other words, VTL Operators are strongly typed and type compliance is statically checked,
1721 i.e., violations result in compile-time errors.

1722 Data types can be related one another, and in particular a data type can have sub-types and
1723 super-types. For example *integer number* is a sub-type of the type *number*, and *number* is in
1724 turn a super-type of *integer number*: this means that any integer number is also a number but
1725 not the reverse, because there is no guarantee that a generic number is also an integer
1726 number. More in general, an object of a certain type is also of the respective super-types, but
1727 there is no guarantee that an object of a super-type is of any of its sub-types.

1728 As a consequence, if a Parameter is required to be of certain type, the arguments have either
1729 this very type or any of its sub-types; arguments of its super-types are not allowed (e.g. if a
1730 Parameter is a *number*, an argument of type *integer* is accepted; vice versa, if it is an *integer*,
1731 an argument of type *number* will not be accepted).

1732 The data types depend on two main factors: the kind of values adopted for the representation
1733 (e.g. text strings, numbers, dates, Boolean values) and the kind of structure of the data (e.g.
1734 elementary scalar values or compound values organized in more complex structures like Sets,
1735 Components, Data Sets ...).

1736 The data types for scalar values also called “scalar types” (e.g. the scalar 15 is of the scalar
1737 type “*number*”, while “hello” is of the scalar type “*string*”). The scalar types are elementary
1738 because they are not defined in term of other data types. All the other data types are
1739 compound.

1740 For the sake of simplicity, hereinafter the term “data type” is sometimes abbreviated to “type”
1741 and the term “scalar type” to “scalar”.

1742 A particular meta-syntax is used to specify the type of the Parameters. For example, the
1743 symbol `::` means “is of the type ...” or simply “is a ...” (e.g. “15 `:: number`” means “15 is of
1744 the type *number*”).

1745 In the following sections, the classes of the VTL types are illustrated, as well as some
1746 relationships between the types and the artefacts of the Information Model.

1747

1748 Data Types overview

1749 Data Types model diagram

1750

1751

1752

1753

1754

1755

1756

1757

1758

1759

1760

1761

1762

1763

1764

1765

1766

1767

1768

1769

1770

1771

1772

1773

1774

1775

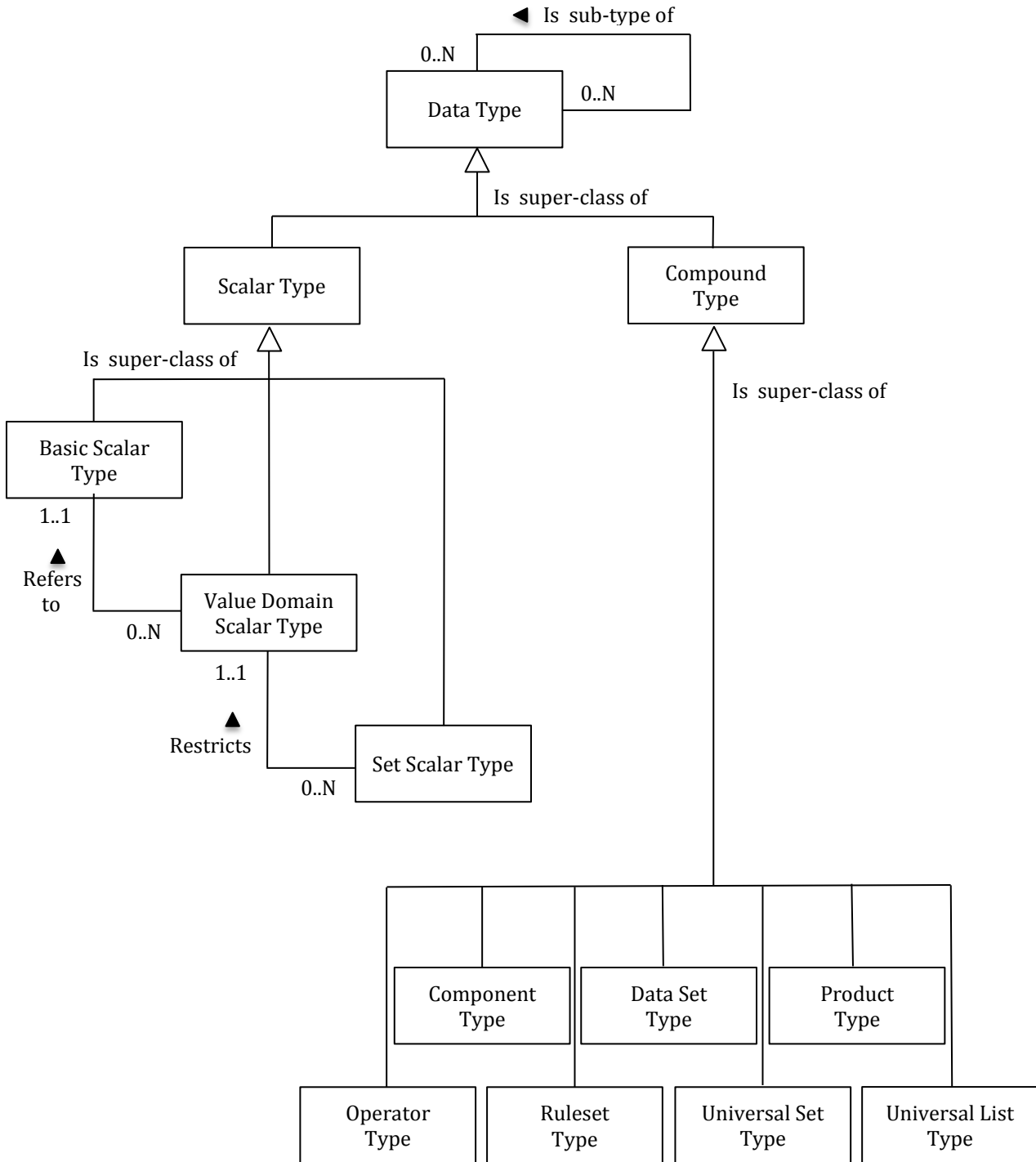
1776

1777

1778

1779

1780



1781 **Explanation of the diagram**

1782 **Data Type:** this is the class of all the data types manipulated by the VTL. As already said, the
1783 actual data type of an object depends on its kind of representation and structure. As for the
1784 structure, a Data Type may be a Scalar Data Type or a Compound Data Type.

1785 **Scalar Type:** the class of all the scalar types, i.e., the possible types of scalar Values. The scalar
1786 types are elementary because they are not defined in terms of other types. The Scalar Types
1787 can be Basic Scalar Types, Value Domain Scalar Types and Set Scalar Types.

1788 **Compound Data Type:** the class of the compound types, i.e. the types that are defined in
1789 terms of other types.

1790 **Basic Scalar Type:** the class of the scalar types which exist by default in VTL (namely, *string*,
1791 *number*, *integer*, *time*, *date*, *time_period*, *duration*, *boolean*).

1792 **Value Domain Scalar Type:** the class of the scalar types corresponding to all the scalar
1793 Values belonging to a Value Domain.

1794 **Set Scalar Type:** the class of the scalar types corresponding to all the scalar Values belonging
1795 to a Set (i.e., Value Domain Subset).

1796 **Component Type:** the class of the types which the Components of the Data Sets belong to, i.e.
1797 Represented Variables which assume a certain Role in the Data Set Structure.

1798 **Data Set Type:** the class of the Data Sets' types, which are the more common input types of
1799 the VTL operators.

1800 **Operator Type:** the class of the Operators' types, i.e., the functions which convert the types
1801 of the input operands in the type of the result.

1802 **Ruleset Type:** the class of the Rulesets' types, i.e. the set of Rules defined by users which
1803 specify the behaviour of other operators (like the check and the hierarchy operators).

1804 **Product Type:** the class of the types which contain Cartesian products of artefacts belonging
1805 to other generic types.

1806 **Universal Set Type:** the class of the types that contain unordered collections of other
1807 artefacts which belong to another generic type and do not have repetitions.

1808 **Universal List Type:** the class of the types that contain ordered collections of other artefacts
1809 which belong to another generic type and can have repetitions.

1810 **General conventions for describing the types**

1811 • The name of the type is written in lower cases and without spaces (for example the Data
1812 Set type is named "dataset").

1813 • The double colon :: means "is of the type ..." or simply "is a ..."; for example the
1814 declaration

1815 operand :: string

1816 means that the operand is a *string*.

1817 • The vertical bar | indicates mutually exclusive type options, for example

1818 operand :: scalar | component | dataset

1819 means that "operand" can be either *scalar*, or *component*, or *dataset*.

1820 • The angular parenthesis **< type2 >** indicates that type2 (included in the parenthesis)
1821 restricts the specification of the preceding type, for example:

1822 operand :: component <string>

1823 means “the operand is a component of *string* basic scalar type”.

1824 If the angular parenthesis are omitted, it means that the preceding type is already
1825 completely specified, for example:

1826 operand :: component

1827 means “the operand is a component without other specifications” and therefore it can be
1828 of any *scalar* type, just the same as writing `operand :: component<scalar>` (in fact as
1829 already said, “scalar” means “any *scalar* type”).

1830 • The underscore `_` indicates that the preceding type appears just one time, for example:

1831 measure<string> _

1832 indicates just one Measure having the scalar type *string*; the underscore also mean that
1833 this is a non-predetermined generic element, which therefore can be any (in the example
1834 above, the string Measure can be any)

1835 • A specific `element_name` in place of the underscore denotes a predetermined element of
1836 the preceding type, for example

1837 measure<string not null> my_text

1838 means just one Measure Component, which is a not-null *string* type and whose name is
1839 “my_text”.

1840 • The symbol `_+` means that the preceding type may appear from 1 to many times, for
1841 example:

1842 measure<string> _+

1843 means one or more generic Measures having the scalar type *string* (these Measures are
1844 not predetermined).

1845 • The symbol `_*` means that the preceding type may appear from 0 to many times, for
1846 example:

1847 measure<string> _*

1848 means zero or more generic Measures having the scalar type *string* (these Measures are
1849 not predetermined).

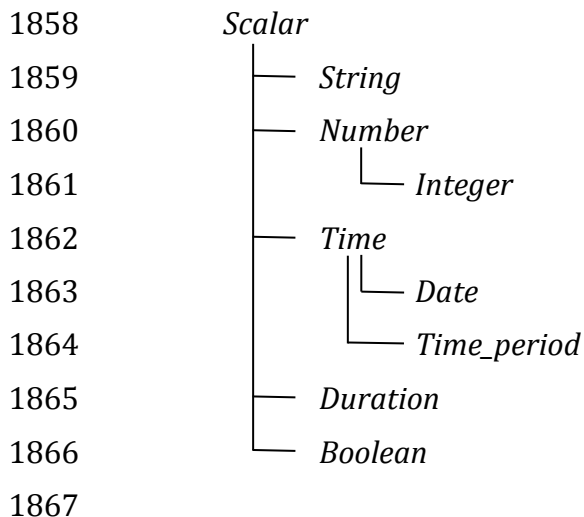
1850 Scalar Types

1851 Basic Scalar Types

1852 The Basic Scalar Types are the scalar types on which VTL is founded.

1853 The VTL has various basic scalar types (namely, *string*, *number*, *integer*, *time*, *date*,
1854 *time_period*, *duration*, *boolean*). The super-type of all the scalar types is the type *scalar*, which
1855 means “any scalar value”. The type *number* has the sub-type *integer* and the type *time* has two
1856 independent sub-types, namely *date* and *time_period*.

1857 The hierarchical tree of the basic scalar types is the following:



1868 A scalar Value of type **string** is a sequence of alphanumeric characters of any length. On string
1869 Values, all the string operations are allowed, such as: concatenation of strings, splitting of
1870 strings, extraction of a part of a string (substring) and so on.

1871 A Scalar Value of type **number** is a rational number of any magnitude and precision, also used
1872 as approximation of a real number. On values of type *number*, the numeric operations are
1873 allowed, such as: addition, subtraction, multiplication, division, power, square root and so on.
1874 The type *integer* (positive and negative integer numbers and zero) is a subtype of the type
1875 *number*.

1876 A Scalar Value of type **time** denotes time intervals of any duration and expressed with any
1877 precision. According to ISO 8601 (ISO standard for the representation of dates and times), a
1878 time interval is the intervening time between two time points. This type can allow operations
1879 like shift of the time interval, change of the starting/ending times, split of the interval,
1880 concatenation of contiguous intervals and so on (not necessarily all these operations are
1881 allowed in this VTL version).

1882 The type **date** is a subtype of the type *time* which denotes time points expressed at any
1883 precision, which are time intervals starting and ending in the same time point (i.e.
1884 intervals of zero duration). A value of type *date* includes all the parts needed to identify
1885 a time point at the desired precision, like the year, the month, the day, the hour, the
1886 minute and so on (for example, 2018-04-05 is the fifth of April 2018, at the precision of
1887 the day).

1888 The type **time_period** is a subtype of the type *time* as well and denotes non-
1889 overlapping time intervals having a regular duration (for example the years, the
1890 quarters of years, the months, the weeks and so on). A value of the type *time_period* is
1891 composite and must include all the parts needed to identify a regular time period at
1892 the desired precision; in particular, the *time-period* type includes the explicit indication
1893 of the kind of regular period considered (e.g., “day”, “week”, “month”, “quarter” ...). For
1894 example, the value 2018M04, assuming that “M” stands for “month”, denotes the
1895 month n.4 of the 2018 (April 2018). Moreover, 2018Q2, assuming that “Q” stands for
1896 “quarter”, denotes the second quarter of 2018. In these examples, the letters M and Q
1897 are used to denote the kind of period through its duration.

1898 A Scalar Value of type **duration** denotes the length of a time interval expressed with any
1899 precision and without connection to any particular time point (for example one year, half
1900 month, one hour and fifteen minutes). According to ISO 8601, in fact, a duration is the amount
1901 of intervening time in a time interval. The *duration* is the scalar type of possible Value
1902 Domains and Components representing the period (frequency) of periodical data.

1903 A Scalar Value of type **boolean** denotes a logical binary state, meaning either “true” or “false”.
1904 Boolean Values allow logical operations, such as: logical conjunction (and), disjunction (or),
1905 negation (not) and so on.

1906 All the scalar types are assumed by default to contain the conventional value “NULL”, which
1907 means “no value”, or “absence of known value” or “missing value” (in other words, the scalar
1908 types by default are “nullable”). Note that the “NULL” value, therefore, is the only value of
1909 multiple different types (i.e., all the nullable scalar types).

1910 The scalar types have corresponding non-nullable sub-types, which can be declared by adding
1911 the suffix “*not null*” to the name of the type. For example, **string not null** is a string that
1912 cannot be NULL, as well as **number not null** is a number that cannot be NULL.

1913 The VTL assumes that a basic scalar type has a unique internal representation and more
1914 possible external representations.

1915 The internal representation is the reference representation of a scalar type in a VTL system,
1916 used to process the scalar values. The use of a unique internal representation allows to
1917 operate on values possibly having different external formats: the values are converted in the
1918 reference representation and then processed. Although the unique internal representation
1919 can be very important for the operation of a VTL system, not necessarily users need to know
1920 it, because it can be hidden in the VTL implementation. The VTL does not prescribe any
1921 predefined internal representation for the various scalar types, leaving different VTL systems
1922 free to using they preferred or already existing ones. Therefore, the internal representations
1923 to be used for the VTL scalar types are left to the VTL implementations.

1924 The external representations are the ones provided by the Value Domains which refer to a
1925 certain scalar type (see also the following sections). These are also the representations used
1926 for the Values of the Components defined on such Value Domains. As obvious, the users have
1927 to know the external representations and formats, because these are used in the Data Point
1928 Values. Obviously, the VTL does not prescribe any predefined external representation, leaving
1929 different VTL systems free to using they preferred or already existing ones.

1930 Examples of possible different choices for external representations:

- 1931 • for the *strings*, various character sets can be used;
- 1932 • for the *numbers*, it is possible to use the dot or the comma as decimal separator, a fixed
1933 or a floating point representation; non-decimal or non-positional numeral systems and
1934 so on;
- 1935 • for the *time*, *date*, *time_period*, *duration* it can be used one of the formats suggested by
1936 the ISO 8601 standard or other possible personalized formats;
- 1937 • the “*boolean*” type can use the values like TRUE and FALSE, or 0 and 1, or YES and NO
1938 or other possible binary options.

1939 It is assumed that a VTL system knows how to convert an external representation in the
1940 internal one and vice-versa, provided that the format of the external representation is known.

1941 For example, the external representation of dates can be associated to the internal one
1942 provided that the parts that specify year, month and day are recognizable²¹.

1943

1944 Value Domain Scalar Types

1945 This is the class of the scalar Types corresponding to the scalar Values belonging to the same
1946 Value Domains (see also the section “Generic Model for Variables and Value Domains”).

1947 The super-type of all the Value Domain Scalar Types is *valuedomain*, which means any Value
1948 Domain Scalar Type. A specific Value Domain Scalar Type is identified by the name of the
1949 Value Domain.

1950 As said in the IM section, a Value Domain is the domain of allowed Values for one or more
1951 represented variables. In other words, a Value Domain is the space in which the abstractions
1952 of a certain category of the reality (population, age, country, economic sector, ...) are
1953 represented.

1954 A Value Domain refers to one of the Basic Scalar Types, which is the basic type of all the
1955 Values belonging to the Value Domain. A Value Domain provides an external representation of
1956 the corresponding Basic Scalar Type and can also restrict the possible (abstract) values of the
1957 latter. Therefore a Value Domain defines a customized scalar type.

1958 For example, assuming that the “population” is represented by means of numbers from zero
1959 to 100 billion, the (possible) “population” Value Domain refers to the “*integer*” basic scalar
1960 type, provides a representation for it (e.g., the number is expressed in the positional decimal
1961 number system without the decimal point) and allows only the integer numbers from zero up
1962 to 100 billion (and not all the possible numbers). Numeric operations are allowed on the
1963 population Values.

1964 As another example, assuming that the “classes of population” are represented by means of
1965 the characters from A to C (e.g. A for population between 0 and 1 million, B for population
1966 greater than 1million until 1 billion, C for population greater than 1 billion), the “classes of
1967 population” Value Domain refers to the “string” basic scalar type and allows only the strings
1968 “A”, “B” or “C”. String operations are possible on these values.

1969 As usual, even if many operations are possible from the syntactical point of view, they not
1970 necessarily make sense in semantics terms: the evaluation of the meaningfulness of the
1971 operations remains up to the users. In fact, the same abstractions, in particular if enumerated
1972 and coded, can be represented by using different possible Value Domains, also using different
1973 scalar types. For example, the *country* can be represented through the ISO 3166-1 numeric
1974 codes (type number), or ISO alpha-2 codes (type string), or ISO alpha-3 codes (type string), or
1975 other coding systems. Even if numeric operations are possible on ISO 3166-1 country numeric
1976 codes, as well as string operations are possible on ISO 3166-1 alpha-2 or alpha-3 country
1977 codes, not necessarily these operations make sense.

²¹ This can be achieved in many ways that depend on the data type and on the adopted internal and external representations. For example, there can exist a default correspondence (e.g., 0 means always False and 1 means always True for Boolean), or the parts of the external representation can be specified through a mask (e.g., for the dates, DD-MM-YYYY or YYYYMMDD specify the position of the digits representing year, month and day).

1978 While the Basic Scalar Types are the types on which VTL is founded and cannot be changed,
1979 all the Value Domains are user defined, therefore their names and their contents can be
1980 assigned by the users.

1981 Some VTL Operators assume that a VTL system have certain kinds of Value Domains which
1982 are needed to perform the correspondent operations²². In the VTL manuals, definite names
1983 and representations are assigned to such Value Domains for explanatory purposes; however
1984 these names and representations are not mandatory and can be personalised if needed. If
1985 VTL rules are exchanged between different VTL systems, the partners of the exchange must
1986 be aware of the names and representations adopted by the counterparties.

1987

1988 **Set Scalar Types**

1989 This is the class of the scalar types corresponding to the scalar Values belonging to the same
1990 Sets (see also the section “Generic Model for Variables and Value Domains”).

1991 The super-type of all the Set Scalar Types is *set*, which means any Set Scalar Type. A specific
1992 Set Scalar Type is identified by the name of the Set.

1993 A Set is a (proper or improper) subset of the Values belonging to a Value Domain (the Set of
1994 all the values of the Value Domain is an improper subset of it). A scalar Set inherits from its
1995 Value Domain the Basic Scalar Type and the representation and can restrict the possible
1996 Values of its Value Domain (as a matter of fact, except the Set which contains all the values of
1997 its Value Domain and can also be assumed to exist by default, the other Sets are defined just to
1998 restrict the Values of the Value Domain).

1999

2000 **External representations and literals used in the VTL Manuals**

2001 The Values of the scalar types, when written directly in the VTL definitions or expressions, are
2002 called *literals*.

2003 The literals are written according to the external representations adopted by the specific VTL
2004 systems for the VTL basic data types (i.e., the representations of their Value Domains). As
2005 already said, the VTL does not prescribe any particular external representation.

2006 In these VTL manuals, anyway, there is the need to write literals of the various data types in
2007 order to explain the behaviour of the VTL operators and give proper examples. The
2008 representation of these literals are not intended to be mandatory and are not part of the VTL
2009 standard specifications, these are only the representations used in the VTL manuals for
2010 explanatory purposes and many other representations are possible and legal.

2011 The representations adopted in these manuals are described below.

2012 The *string* values are written according the Unicode and ISO/IEC 10646 standards.

²² For example, at least one default Value Domain should exists for each basic scalar type, the Value Domains needed to represent the results of the checks should exist, and so on.

- 2013 The **number** values use the positional numeral system in base 10.
- 2014
- 2015
- 2016
- 2017
- 2018
- 2019
- 2020
- 2021
- a fixed-point *number* begins with the integer part, which is a sequence of numeric characters from 0 to 9 (at least one digit) optionally prefixed by plus or minus for the sign (no symbol means plus), a dot is always present in the end of the integer part and separates the (possible) fractional part, which is another sequence of numeric characters.
 - A floating point *number*, has a mantissa written like a fixed-point number, followed by the capital letter E (for “Exponent”) and by the exponent, written like a fixed-point integer;

2022 For example:

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

2025 The **integer** values are represented like the *number* values with the following differences:

2026

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

2031 For example:

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

2034 The **time** values are conventionally represented through the initial and final Gregorian dates of the time interval separated by a slash. The accuracy is reduced at the level of the day (therefore omitting the time units shorter than the day like hours, minutes, seconds, decimals of second). The following format is used (this is one of the possible options of the ISO 8601 standard):

2035

2036

2037

2038

2039 *YYYY-MM-DD/YYYY-MM-DD*

2040 Where *YYYY* indicates 4 digits for the year, *MM* indicates two digits for the month, *DD*

2041 indicates two digits for the day. For example:

2042	2000-01-01/2000-12-31	the whole year 2000
2043	2000-01-01/2009-12-31	the first decade of the XXI century

2044 The **date** values are conventionally represented through one Gregorian date. The accuracy is reduced at the level of the day (therefore omitting the time units shorter than the day like hours, minutes, seconds, decimals of second). The following format is used (this is one of the possible options of the ISO 8601 standard):

2045

2046

2047

2048 *YYYY-MM-DD*

2049 The meaning of the symbols is the same as above. For example:

2050	2000-12-31	the 31 st December of the year 2000
2051	2010-01-01	the first of January of the year 2010

2052 The **time_period** values are represented for sake of simplicity with accuracy equal to

2053 the day or less (week, month ...) and a periodicity not higher than the year. In the VTL

2054 manuals the following format is used (this is a personalized format not compliant with
2055 the ISO 8601 standard):

2056 *YYYYPppp*

2057 Where *YYYY* are 4 digits for the year, *P* is one character for specifying which is the
2058 duration of the regular period (e.g. D for day, W for week, M for month, Q for quarter, S
2059 for semester, Y for the whole year, see the codes of the *duration* data type below), *ppp*
2060 denotes from zero two three digits which contain the progressive number of the period
2061 in the year. For example:

2062	2000M12	the month of December of the year 2000
2063	2010Q1	the first quarter of the year 2010
2064	2020Y	the whole year 2010

2065 The ***duration*** values in these manuals are conventionally restricted to very few predefined
2066 durations which are codified through just one character as follows:

2067	<i>Code</i>	<i>Duration</i>
2068	D	Day
2069	W	Week
2070	M	Month
2071	Q	Quarter
2072	S	Semester
2073	A	Year (Annual)

2074 This is a very simple format not compliant with the ISO 8601 standard, which allows
2075 representing durations in a much more complete, even if more complex, way. As mentioned,
2076 the real VTL systems may adopt any other external representation.

2077 The ***boolean*** values used in the VTL manuals are *TRUE* and *FALSE* (without quotes).

2078 When a *literal* is written in a VTL expression, its basic scalar type is not explicitly declared
2079 and therefore is unknown.

2080 For ensuring the correctness of the VTL operations, it is important to assess the scalar type of
2081 the literals when the expression is parsed. For this purpose, there is the need for a mechanism
2082 for the disambiguation of the literals types, because often the same literal might in itself
2083 belong to many types, for example:

- 2084 • the word “true” may be interpreted as a *string* or a *boolean*,
- 2085 • the symbol “0” may be interpreted as a *string*, a *number* or a *boolean*,
- 2086 • the word “20171231” may be interpreted as a *string*, a *number* or a *date*.

2087 The VTL does not prescribe any predefined mechanism for the disambiguation of the scalar
2088 types of the literals, leaving different VTL systems free to using they preferred or already
2089 existing ones. The disambiguation mechanism, in fact, may depend also on the conventions
2090 adopted for the external representation of the scalar types in the VTL systems, which can be
2091 various.

2092 In these VTL manuals, anyway, there is the need to use a disambiguation mechanism in order
2093 to explain the behaviour of the VTL operators and give proper examples. This mechanism,
2094 therefore, is not intended to be mandatory and, strictly speaking, is not part of the VTL
2095 standard.

2096 If VTL rules are exchanged between different VTL systems, the partners of the exchange must
2097 be aware of the external representations and the disambiguation mechanisms adopted by the
2098 counterparties.

2099 The disambiguation mechanism adopted in these VTL manuals is the following:

- 2100 • The *string* literals are written between double quotes, for example the literal “123456”
2101 is a string, even if its characters are all numeric, as well as “I am a string!”.
- 2102 • The *numeric* literals are assumed to have some pre-definite patterns, which are the
2103 numeric patterns used for the external representation of the numbers described above.
2104 A literal having one of these patterns is assumed to be a *number*.
- 2105 • The *boolean* literals are assumed to be the values *TRUE* and *FALSE* (capital letters
2106 without quotes).

2107 In these manuals, it is also assumed that the types *time*, *date*, *time_period* and *duration* do not
2108 directly support literals. Literal values of such types can be anyway built from literals of other
2109 types (for example they can be written as strings) and converted in the desired type by the
2110 cast operator (type conversion). In some cases the conversion can be made automatically,
2111 (i.e., without the explicit invocation of the cast operator – see the Reference Manual for more
2112 details).

2113 As mentioned, the VTL implementations may personalize the representation of the literals
2114 and the disambiguation mechanism of the basic scalar types as desired, provided that the
2115 latter work properly and no ambiguities in understanding the type of the literals arise. For
2116 example, in some cases the type of a literal can also be deduced from the context in which it
2117 appears. As already pointed out, the possible personalised mechanism should be
2118 communicated to the counterparties if the VTL rules are exchanged.

2119 Conventions for describing the scalar types

- 2120 • The keywords which identify the basic scalar types are the following: scalar, string,
2121 number, integer, time, date, time_period, duration, boolean.
- 2122 • By default, the basic scalar types are considered as nullable, i.e., allowing NULL values.
- 2123 • The keyword **not null** following the type (and the “literal” keyword if present), means that
2124 the scalar type does not allow the NULL value, for example:

2125 operand :: string literal not null

2126 means that the operand is a literal of *string* scalar type and cannot be NULL; if not null is
2127 omitted the NULL value is meant to be allowed.

- 2128 • An **expression within square brackets** following the previous keywords, means that the
2129 preceding scalar type is restricted by the expression. This is a VTL *boolean* expression²³
2130 (whose result can be TRUE or FALSE) which specifies a membership criterion, that is a
2131 condition that discriminates the values which belong to the restriction (sub-type) from the
2132 others (the value is assumed to belong to the sub-type only if the expression evaluates to
2133 TRUE). The keyword “value” stands for the generic value of the preceding scalar type and
2134 is used in the expression to formulate the restrictive condition. For example:

2135 integer [value <= 6]

²³ I.e., an expressions whose result is *boolean*

2136 is a sub-type of *integer* which contains only the integers lesser than or equal to 6.

2137 The following examples show some particular cases:

- 2138 ○ The generic expression [**between (value, x, y)**] ²⁴ restricts a scalar type by
2139 indicating a closed interval of possible values going from the value x to the value y,
2140 for example

2141 `integer [between (value, 1, 100)]`

2142 is the sub-type which contains the integers between 1 and 100.

- 2143 ○ The generic expression [**(value > x) and (value < y)**] restricts a scalar type by
2144 indicating an open interval of possible values going from the value x to the value y,
2145 for example

2146 `integer [(value > 1) and (value < 100)]`

2147 means integer greater than 1 and lesser than 100 (i.e., between 2 and 99).

- 2148 ○ By using **>=** or **<=** in the expressions above, the intervals can be declared as open
2149 on one side and closed on the other side, for example

2150 `integer [(value >= 1) and (value < 100)]`

2151 means integer greater than or equal to one and lesser than 100.

- 2152 ○ The generic expressions [**value >= x**] or [**value > x**] or [**value <= y**] or [**value**
2153 **< y**] restrict a scalar type by indicating an interval having one side unbounded, for
2154 example

2155 `integer [value >= 1]`

2156 means integer equal to or greater than 1, while “integer[value < 100]” means an
2157 integer lesser than 100.

- 2158 ○ The generic expression [**value in { v₁, ... , v_N }**] ²⁵ restricts a scalar type by
2159 specifying explicitly a set of possible values, for example

2160 `integer { 1, 2, 3, 4, 5, 6 }`

2161 means an integer which can assume only the integer values from 1 to 6. The same
2162 result is obtained by specifying [value in set_name], where in is the “Element of”
2163 VTL operator and set_name is the name of an existing Set (Value Domain Subset)
2164 of the VTL IM.

- 2165 ○ By using in the expression the operator **length** ²⁶ it is possible to restrict a scalar
2166 type by specifying the possible number of digits that the values can have, for
2167 example

2168 `integer [between (length (value), 1, 10)]`

²⁴ “between (x, y, z)” is the VTL operator which returns TRUE if x is comprised between y and z

²⁵ “in” is the VTL operator which returns TRUE if an element (in this case the value) belongs to a Set; the symbol { ..., ..., ... } denotes a set defined as the list of its elements (separated by commas)

²⁶ “length” is the VTL Operator that returns the length of a string (in the example, the *integer* operand of the length operator is automatically cast to a string and its length is determined)

2169 means an integer having a length from one to 10 digits;

2170 As obvious, other kinds of conditions are possible by using other VTL operators and more

2171 conditions can be combined in the restricting expression by using the VTL boolean

2172 operators (and, or, not ...)

2173 • Like in the general case, a restricted scalar type is considered by default as including the

2174 NULL value. If the NULL value must be excluded, the type specification must be followed

2175 by the symbol **not null**; for example

2176 integer [between (length (value), 1, 10)] not null

2177 means a not-null integer having from one to 10 digits

2178 Compound Data Types

2179 The Compound data types are the types defined in terms of more elementary types.

2180 The compound data types are relevant to artefacts like Components, Data Sets and to other

2181 compound structures. For example, the a type Component is defined in terms of the scalar

2182 type of its values, besides some characteristics of the Component itself (for example the role it

2183 assumes in the Data Set, namely Identifier, Measure or Attribute). Similarly, the type of a Data

2184 Set (i.e. of a mathematical function) is defined in terms of the types of its Components.

2185 The compound Data Types are described in the following sections.

2186 Component Types

2187 This is the class of the Component types, i.e. of the Components of the Data Structures (for

2188 example the “country of residence” in the role of Identifier, the “resident population” in the

2189 role of Measure ...).

2190 A Component is essentially a Variable (i.e. an unknown scalar Value with a certain meaning,

2191 e.g. the resident population) which takes Values in a Value Domain and plays a definite role in

2192 a Data Structure (e.g., Identifier, Measure, Attribute). A Component inherits the scalar type

2193 (e.g. *number*) from the relevant Value Domain.

2194 The main sub-types of the Component Type depend on the role of the Component in the data

2195 structure and are the *identifier*, *measure* and *attribute* types (if the automatic propagation of

2196 the Attributes is supported, another sub-type is the *viral attribute*). These types reflect the

2197 fact that the VTL behaves differently on Components of different roles. Their common super-

2198 type is *component*, which means “a Component having any role”.

2199 Moreover, a Component type can be restricted by an associated scalar type (e.g. *number*,

2200 *string*, ...), therefore the complete specification of a Component type takes the form

2201 role_type < scalar_type >

2202 where the scalar type included in angular parenthesis, restricts the specification of the

2203 preceding type (the role type); omitted angular parenthesis mean “any scalar type”, which is

2204 the same as writing <*scalar*>. Examples of Component types are the following:

- 2205 • component (or component<scalar>) any Component
- 2206 ○ component<number> any Component of scalar type *number*
- 2207 ○ identifier (or identifier<scalar>) any Identifier

- 2208 ▪ identifier<time not null> Identifier of scalar type *time not null*
- 2209 ○ measure (or measure<scalar>) any Measure
 - 2210 ▪ measure<boolean> Measure of scalar type *boolean*
 - 2211 ○ attribute (or attribute<scalar>) any Attribute
 - 2212 ▪ attribute<string> Attribute of scalar type *string*

2213 In the list above, the more indented types are sub-types of the less indented ones.

2214 According to the functional paradigm, the Identifiers cannot contain NULL values, therefore
2215 the scalar type of the Identifiers Components must be “not null”.

2216 In summary, the following conventions are used for describing Component types.

- 2217 • As already said, the more general type is “**component**” which indicates any component,
2218 for example

2219 operand :: component

2220 means that “operand” may be any component.

- 2221 • The main sub-types of the *component* type correspond to the roles that the Component
2222 may assume in the Data Set, i.e., **identifier**, **measure**, **attribute**; for example

2223 operand :: measure

2224 means that the operand must be a Measure.

2225 The additional role **viral attribute** exists if the automatic propagation of the Attributes is
2226 supported.²⁷ The type *viral_attribute* is a sub-type of *attribute*.

- 2227 • By default, a Component can be either specified directly through its name or indirectly
2228 through a sub-expression which calculates it.

- 2229 • The optional keyword **name** following the type keyword means that a component name
2230 must be specified and that the component cannot be obtained through a sub-expression;
2231 For example:

2232 operand :: measure name <string>

2233 means that the name of a *string* Measure must be specified and not a string sub-
2234 expression²⁸. If the **name** keyword is omitted the sub-expression is allowed.

- 2235 • The symbol < scalar type > means that the preceding type is restricted to the scalar type
2236 specified within the angular brackets”, for example

2237 operand :: component < string >

2238 means that the operand is a Component having any role and belonging to the *string* scalar
2239 type; if the restriction is not specified, then the scalar type can be any (for
2240 example operand:: attribute means that the operand is an Attribute of any scalar type).

- 2241 • In turn, the scalar type of a Component can be restricted; for example

²⁷ See the section “Behaviour for Attribute Components”

²⁸ I.e., a sub-expressions whose result is *string*

2242 operand:: measure < integer [value between 1 and 100] not null >

2243 means that the operand can be a not-null integer Measure whose values are comprised
2244 between 1 and 100;

2245 Data Set Types

2246 This is the class of the Data Sets types. The Data Sets are the main kind of artefacts
2247 manipulated by the VTL and their types depend on the types of their Components.

2248 The super-type of all the Data Set types is *dataset*, which means “any dataset” (according to
2249 the definition of Data Set given in the IM, as obvious).

2250 A sub-type of dataset is the Data Sets of time series, which fulfils the following restrictive
2251 conditions:

- 2252 • The Data Set structure must contain one Identifier Component that acts as the reference
2253 time, which must belong to one of the basic scalar types *time*, *date* or *time_period*.
- 2254 • The possible values of the reference time Identifier Component must be regularly spaced
 - 2255 ○ For the type *time*, the time intervals must start (or end) at a regular periodicity and
2256 have the same duration
 - 2257 ○ For the type *date*, the time values must have a regular periodicity
 - 2258 ○ For the type *time_period* there are no additional conditions to fulfil, because the
2259 *time_period* values comprise by construction the indication of the period and
2260 therefore are regularly spaced by default

- 2261 • It is assumed that it exist the information about which is Identifier Components that acts
2262 as the reference time and about which is the period (frequency) of the time series and that
2263 such information is represented in some way in the VTL system. The VTL does not
2264 prescribe any predefined representation, leaving different VTL systems free to using they
2265 preferred or already existing ones. It is assumed that the VTL operators acting on time
2266 series know which is the reference time Identifier and the period of the time series and
2267 use these information to perform correct operations.

2268 Usually, the information about which is the reference time is included in the data structure
2269 definition of the Data Sets or in the definition of the Data Set Components.

2270 Some commonly used representations of the periodicity are the following:

- 2271 ○ For the types *time* and *date*, the period is often represented through an additional
2272 Component of the Data Set (of any possible role) or an additional metadata relevant
2273 to the whole Data Set or some parts of it. This Component (or other metadata) is of
2274 the “duration” type and is often called “frequency”.
- 2275 ○ For the type *time_period*, the periodicity is embedded in the *time_period* values.

2276 In any case, if some periodical data exist in the system, it is assumed that a Value Domain
2277 representing the possible periods exists and refers to the *duration* scalar type.

2278 Within a Data Set of Time Series, a single Time Series is the set of Data Points which have the
2279 same values for all the Identifier Components except the reference time²⁹. A Data Set of time
2280 series can also contain more time series relevant to the same phenomenon but having
2281 different periodicities, provided that one or more Identifiers (other than the reference time)
2282 distinguish the Time Series having different periodicity.

²⁹ Therefore each combination of values of the Identifier Components except the reference time identifies a Time Series.

2283 The Data Sets of time series are the possible operands of the time series operators (they are
2284 described in the Reference Manual).

2285 More specific Data Set types can be defined by constraining the *dataset* type, for example by
2286 specifying the number and the type of the possible Components in the different roles
2287 (Identifiers, Measures and Attributes), and even their names if needed. Therefore the general
2288 syntax for specifying a Data Set type is

2289 `dataset { type_constraint } or dataset_ts { type_constraint }`

2290 where the *type_constraint* may assume many different forms which are described in detail in
2291 the following section. Examples of Data Set types are the following:

2292 `dataset` Any Data Set (according to the IM)

2293 `dataset { measure <number> _* }` A Data Set having one or more Measures of
2294 type *number*, without constraints on
2295 Identifiers and Attributes

2296 `dataset { measure <boolean> _ , attribute<string> _* }`
2297 A Data Set having one *boolean* Measure, one
2298 or more *string* Attributes and no constraints
2299 on Identifiers

2300 In summary, the following conventions are used for describing Data Set types.

- 2301 • The more general type is “**dataset**” which means any possible Data Set of the VTL IM (in
2302 other words, a Data Set having any possible components allowed by the IM integrity rules)
- 2303 • By default, a Data Set can be either specified directly through its name or indirectly
2304 through a sub-expression which calculates it.
- 2305 • The optional keyword **name** following **dataset** means that a Data Set name must be
2306 specified and that the Data Set cannot be obtained through a sub-expression; For
2307 example:

2308 `operand:: dataset name`

2309 means that a Data Set name must be specified and not a sub-expression. If the name
2310 keyword is omitted the sub-expression is allowed.

- 2311 • The symbol **dataset { type_constraint }** indicates that the *type_constraint* included in
2312 curly parenthesis restricts the specification of the preceding *dataset type* without giving a
2313 complete type specification, but indicating only the constraints in respect to the general
2314 structure of the artefact of the Information Model corresponding to such *type*. For
2315 example, given that the generic structure of a Data Set in the IM may have any number of
2316 Identifiers, Measures and Attributes and that these Components may be of any scalar type,
2317 the declaration

2318 `operand :: dataset { measure<string> _ }`

2319 means that the operand is of type Data Set having any number of Identifiers (like in the
2320 IM), just one Measure of string type (as declared in the type declaration) and any number
2321 of Attributes (like in the IM).

- 2322 • Some or all the Data Set Components can also be predetermined. For example writing

2323 operand:: dataset { identifier<st_Id₁> Id₁, ..., identifier<st_Id_N> Id_N,
2324 measure<st_Me₁> Me₁, ... , measure<st_Me_L> Me_L, attribute<st_At₁>
2325 At₁, ... , attribute<st_At_K> At_K }

2326 means that the operand is of Data Set type and has the identifier, measure and attribute
2327 types and names specified within the curly brackets (in the example, <st_Id₁> stands for
2328 the scalar type of the Component named Id₁ and so on). This is the example of an
2329 extremely specific Data Set type in which all the component types and names are fixed in
2330 advance.

2331 • If a certain role (i.e. identifier, measure, attribute) is not specified, it means that there are
2332 no restrictions on it, for example

2333 operand:: dataset { me<st_Me₁> Me₁, ... , me<st_Me_L> Me_L }

2334 means that the operand is of Data Set type and has the measure types and names specified
2335 within the curly brackets, while the Identifier and Attribute components have no
2336 restrictions and therefore can be any.

2337 Product Types

2338 This is the class of the Cartesian products of other types; a product type is written in the form
2339 $t_1 * t_2 * \dots * t_n$ where t_i ($i=1 \dots n$) is another arbitrary type; the elements of a Product type are
2340 n-tuples whose i-th element belongs to the type t_i . For instance, the product type

2341 *string * integer * boolean*

2342 includes elements like³⁰ ("PfgTj", 7, true), ("kj-o", 80, false), ("", 4, false) but does not include
2343 for example ("qwe", 2017-12-31, true), ("kj-o", 80, 92).

2344 The superclass is *product*, which means any product type

2345 Product types can be used in practice for several reasons. They allow:

- 2346 i. the natural expression of exclusion or inclusion criteria (i.e., constraints) over
2347 values of two or more dataset components,
- 2348 ii. the definition of the domain of the Operators in term of types of their Parameters
- 2349 iii. the definition of more complex data types.

2350 Operator Types

2351 This is the class of the Operators' types, i.e., the higher-levels functions that allow
2352 transformations from the type t_1 (the type of the input Parameters), to the type t_2 (the type of
2353 the output Parameter). An Operator Type is written in the form ' $t_1 \rightarrow t_2$ ', where t_1 and t_2 are
2354 arbitrary types. For example, the type of the following operator says that it takes as input two
2355 integer Parameters and returns a number.

2356 $Op_1 :: integer * integer \rightarrow number$

2357 The superclass is *operator*, which means any operator type

2358

³⁰ In the VTL syntax the symbol () allows to define a tuple in-line by enumeration of its elements.

2359 **Ruleset Types**

2360 The class of the Ruleset types, i.e. the set of Rules that are used by some operators like
2361 “check_hierarchy”, “check_hatapoint”, “hierarchy”, “transcode”. The general syntax for
2362 specifying a Ruleset type is `main_type_of_ruleset {type_constraint}`.

2363 The main Rulesets types are the *datapoint* and the *hierarchical* Rulesets. Their super-type is
2364 *ruleset* which means “any Ruleset”. Moreover, Rulesets can be defined either on Value
2365 domains or on Variables, therefore the `main_type_of_rulesets` are:

- 2366 • *ruleset*
 - 2367 ○ *datapoint*
 - 2368 ▪ *datapoint_on_value_domains*
 - 2369 ▪ *datapoint_on_variables*
 - 2370 ○ *hierarchical*
 - 2371 ▪ *hierarchical_on_value_domains*
 - 2372 ▪ *hierarchical_on_variables*

2373 In the list above, the more indented types are sub-types of the less indented ones.

2374 The `type_constraint` is optional and may assume many different forms which depends on the
2375 `main_type_of_ruleset`. If the `type_constraint` is present, the `main_type_of_ruleset` must
2376 specify if the ruleset is defined on Value Domains or Variables (i.e., it must be one of the more
2377 indented types above).

2378 A *datapoint* Ruleset is defined on a Cartesian product of Value Domains or Variables,
2379 therefore the `type_constraint` can contain such a list. Examples of constrained *datapoint* types
2380 are:

```
2381     datapoint on value domains {(geo_area * sector * time_period * numeric_value)}  
2382     datapoint on variables {(ref_date * import_currency * import_country)}  
2383     datapoint on value domains {date * _+}
```

2384 The last one is the type of the Data Point Rulesets that are defined on the “date” Value Domain
2385 and on one to many other Value Domains (“_+” means “one or more”).

2386 A hierarchical Ruleset is defined on one Value Domain or Variable and can contain conditions
2387 referred to other Value Domains or Variables, therefore the `type_constraint` for hierarchical
2388 Rulesets can take one of the following forms:

```
2389     {value_domain * (conditioningValueDomain1 * ... * conditioningValueDomainN)}  
2390     {variable * (conditioningVariable1 * ... * conditioningVariableN)}.
```

2391 Examples of *hierarchical* types are:

```
2392     hierarchical on value domains {geo_area * ( time_period ) }  
2393     hierarchical on variables { currency * ( date * country ) }  
2394     hierarchical on value domains { _ }  
2395     hierarchical on value domains { _ * ( reference_date ) }
```

2396 The last one is the type of the Hierarchical Rulesets that are defined on any Value Domain and
2397 are conditioned by the reference date Value Domain.

2398

2399 **Universal Set Types**

2400 The Universal Sets are unordered collections of other objects that belong to the same type *t*
2401 and do not have repetitions (each object can belong to a Set just once). The Universal Sets are
2402 denoted as *set* < *t* >, where *t* is another arbitrary type. If < *t* > is not specified it means any
2403 universal set type.

2404 Possible examples are the Sets of product types. For instance, the Universal Set Type:

2405 *set* < *string* * *integer* * *boolean* >

2406 includes the sets³¹ :

2407 { ("PfgTj", 7, true), ("kj-o", 80, false), ("", 4, false) }

2408 { ("duo9", 67, true), ("io/p", 540, true) }

2409 But does not includes the sets:

2410 { ("PfgTj", 7, true), 80, ("", 4, false) } in fact 80 is not a *product type*

2411 { ("duo9", 67, true), (50, true) } in fact (50, true) is not the right *product type*

2412 { ("", 4, false), ("F", 8, true), ("", 4, false) } in fact ("", 4, false) is repeated

2413 **Universal List Types**

2414 The Universal Lists are ordered collections of other objects that belong to the same type *t* and
2415 can have repetitions (an object can appear in a list more than once). The Universal Lists are
2416 denoted as *list* < *t* >, where *t* is an arbitrary type. If < *t* > is not specified it means any
2417 universal list type.

2418 For instance, the following Universal List type:

2419 *list* < *component* >

2420 includes elements like³² [reference date, import, export] but does not include elements like
2421 [dataset1, country, sector] and [import, "text"] because dataset1 and "text" are not
2422 Components.

³¹ In the VTL syntax, the symbol {...} denotes a set defined as the list of its elements (separated by commas)

³² In the VTL syntax, the symbol [] allows to define a List in-line by enumeration of its elements.

2424 This section describes the key concepts, assumptions and characteristics of the VTL which are
 2425 needed to a VTL user to define Transformations. As mentioned in the section about the
 2426 general characteristics above, the language is oriented to users without deep information
 2427 technology (IT) skills, who should be able to define calculations and validations
 2428 independently, without the intervention of IT personnel. Therefore, the VTL has been
 2429 designed to make the definition of the Transformations as intuitive as possible and to reduce
 2430 the chances of errors.

2431 As already said, a Transformation consists of a statement which assigns the outcome of the
 2432 evaluation of an Expression to an Artefact of the Information Model. Then, transformations
 2433 are made of the following components:

- 2434 • A right-hand side, which contains the expression to be evaluated, whose inputs are the
- 2435 operands of the Transformation
- 2436 • An assignment operator
- 2437 • A left-hand side, which specifies the Artefact which the outcome of the expression is
- 2438 assigned to (this is the result of the Transformation)

2439 Examples of assignments are (assuming that D_i ($i=1...n$) are Data Sets):

- 2440 • $D_1 := D_2$
- 2441 • $D_3 := D_4 + D_5$

2442 Assuming that E is the expression, R is the result and IO_i ($i=1,...,n$) the input Operands, the
 2443 mathematical form of a Transformation based on E can be written as follows:

$$2444 \quad R := E (IO_1, IO_2, \dots, IO_n)$$

2445 The expression uses any number of VTL operators in combination to specify a compound
 2446 operation. Because all the VTL operators are functional, the whole expression is functional
 2447 too.

2448 Transformations are properly chained for their execution, in fact the result R_i of a
 2449 Transformation T_i can be referenced as the operand of another Transformation T_j . In this case,
 2450 the former Transformation is evaluated first in order to provide the input for the latter. To
 2451 enforce the consistency of the results, cycles are not allowed, therefore in the case above the
 2452 result R_j of the Transformation T_j cannot be operand of the Transformation T_i and cannot
 2453 contribute to the calculation of any operand of T_i , even indirectly through a chain of other
 2454 Transformations.

2455 The order in which the user defines the Transformations may be important for a better
 2456 understanding but cannot override the order of execution determined according their input-
 2457 output relationships.

2458 For the rules for the Transformation consistency, see also the section “Generic Model for
 2459 Transformation” above.

2460 A VTL program is a set of Transformations executed in the same run, which is defined as a
 2461 Transformation Scheme.

2462

2463 The Expression

2464 A VTL expression constitutes the right-hand side of a Transformation. It takes one or more
2465 input operands and returns one output artefact.

2466 An expression is the invocation of one or more operators in combination, in which the result
2467 of an operator is passed as input parameter to another operator, and so on, in a tree structure.
2468 The root of the tree structure is last operator to be applied and gives the final result.

2469 For example, for the expression $a + b - c$ the result of the addition $a + b$ is passed to the
2470 following subtraction, which gives the final result.

2471 An expression is built from the following ingredients:

- 2472 • **Operators**, which specify the operation to be performed (e.g. +, - and so on). As
2473 mentioned, the standard VTL operators are described in detail in the Reference
2474 Manual, moreover the VTL allows defining and then invoking “user defined operators”
2475 (see the Reference Manual). Each operator envisages a certain number of input
2476 parameters of definite data types and produces an outcome having a definite data type
2477 (the types parameter are described in detail in the Reference Manual for each
2478 operator).
- 2480 • **Operands**, which are the actual arguments passed to the invoked Operators, for
2481 example in the expression $D_1 + D_2$ the Operator “+” is invoked and the Operands D_1
2482 and D_2 are passed to it. The Operands can be:
 - 2483 ○ **Named artefacts**, which are VTL artefacts specified through their names. Their
2484 actual values are obtained either referring to an external persistent source
2485 (persistent artefacts) or as result of previous Transformations (non-persistent
2486 artefacts) of the same Transformation Scheme; they are identified by means of a
2487 symbolic name (e.g. in $D_1 + D_2$ the Operands D_1 and D_2 are identified by the
2488 names D_1 and D_2). Examples of identified artefacts are the Data Sets (like D_1
2489 and D_2 above) and the Data Set Components (like $D_1\#C_1, D_1\#C_2, D_1\#C_3$, where #
2490 means that C_j is a Component of the Data Set D_i).
 - 2491 ○ **Literals**, which are VTL artefacts whose actual values are directly written in the
2492 expression; for example, in the invocation $D_1 + 7$ the second operand (7) is a
2493 literal, in this case a scalar literal. Also other kind of artefacts can be written in
2494 the expressions, for example the curly brackets denote the value of a Set (for
2495 example $\{1, 2, 3, 4, 5, 6\}$ is the set of the integers from 1 to 6) and the square
2496 brackets denote a list (for example $[7, 5, 3, 6, 3]$ is a list of numbers).
- 2498 • **Parentheses**, which specify the order of evaluation of the operators; for example in
2499 the expression $D_1 * (D_2 + D_3)$ first the sum $D_2 + D_3$ is evaluated and then their
2500 product for D_1 . In case the parenthesis are not used, the default order of evaluation
2501 (described in the Reference Manual) is applied (in the example, first the product and
2502 then the sum).

2503 An expression implies different steps of calculation, for example the expression:

$$2504 \quad R := O_1 + O_2 / (O_3 - O_4 / O_5)$$

2505 Can be calculated in the following steps:

$$2506 \quad \text{I. } (O_4 / O_5)$$

- 2507 II. $(O_3 - I)$
- 2508 III. (O_2 / II)
- 2509 IV. $(O_1 + III)$

2510 The intermediate and final outputs (I, II, III, IV) of the expression are assumed to be non-
 2511 persistent (temporary). The persistency of the result Data Set R is controlled by the
 2512 assignment operator, as described below.

2513 An intermediate result within the expression can be only the input of other operators in the
 2514 same expression.

2515 In general, unless differently specified in the Reference Manual, in the invocation of an
 2516 operator any operand can be the result of a sub-expression which calculates it. For example,
 2517 taking the exponentiation whose syntax is

2518 $power(base, exponent),$

2519 the invocation $power(D_1 + D_2, 2)$ is allowed and means that first $D_1 + D_2$ is calculated and then
 2520 the result is squared. As usual, the data type of the calculated operand must comply with the
 2521 allowed data types of the corresponding Parameter (in the example above, $D_1 + D_2$ must have
 2522 a numeric data type, otherwise it cannot be squared).

2523 The nesting capabilities allow writing from very simple to very complex expressions. Users
 2524 can manage the complexity of the expressions by splitting or merging transformations. For
 2525 example, taking again the example above, the following two options would give the same
 2526 result:

2527 Option 1:

2528 $D_r := power(D_1 + D_2, 2)$

2529 Option 2:

2530 $D_3 := D_1 + D_2$

2531 $D_r := power(D_3, 2)$

2532 In both cases, in fact, first $D_1 + D_2$ is evaluated and then the $power$ operator is applied to obtain
 2533 D_r .

2534 In general, it is possible either to have simpler expressions splitting and chaining
 2535 Transformations or to have a minor number of Transformations writing more complex
 2536 expressions.

2537 The Assignment

2538 The assignment of an expression to an artefact is done through an assignment operator. The
 2539 VTL has two assignment operators, the persistent and the non-persistent assignment:

2540 $<-$ persistent assignment

2541 $:=$ non-persistent assignment

2542 The former assigns the outcome of the expression on the left side to a persistent artefact, the
 2543 latter to a non-persistent one; therefore the choice of the assignment operator allows to
 2544 control the persistency of the artefact which is result of the Transformation.

2545 The only artefact that can be made persistent is the result (the left side artefact). In fact, as
2546 already mentioned, the intermediate and final results of the right side expression are always
2547 considered as non-persistent.

2548 For example, taking again the example of Transformation above:

2549
$$D_r := power(D_1 + D_2, 2)$$

2550 The result D_r can be declared as persistent by writing:

2551
$$D_r <- power(D_1 + D_2, 2)$$

2552 Instead to make persistent also the intermediate result of $D_1 + D_2$ it is necessary to split the
2553 Transformation like in the option 2 above:

2554
$$D_3 <- D_1 + D_2$$

2555
$$D_r <- power(D_3, 2)$$

2556 The persistent assignment operator is also called *Put*, because it is used to specify that a result
2557 must be put in a persistent store. The *Put* has two parameters, the first is the final result of the
2558 expression on the right side that has to be made persistent, the second is the reference to the
2559 persistent Data Set which will contain such a result.

2560 The Result

2561 The left side artefact, i.e., the result of the Transformation, is always a named Data Set (i.e. a
2562 Data Set identified by means of a symbolic name like explained in the previous section).

2563 The data type and structure of the left side Data Set coincide with the data type and structure
2564 of the outcome of the expression, which must be a Data Set as well.

2565 Almost all VTL operators act on Data Sets. Many VTL operators can act also on Data Set
2566 Components to produce other Data Set Components, however even in this case the outcome of
2567 the expression is a new Data Set which contains the calculated Components.

2568 An expression can result also in scalar Value, because many VTL operators can act on scalar
2569 Values to obtain other scalar Values, furthermore some particular operations on Data Sets can
2570 eliminate Identifiers, Measures and Attributes and obtain scalar Values (see the Reference
2571 Manual). The result of such expressions is considered as a named Data Set which does not
2572 have Components (Identifiers, Measures and Attributes) and therefore contains just one
2573 scalar Value. The Data Sets without Components can be manipulated and possibly stored like
2574 any other Data Set. Because the VTL notion of Data Set is logical and not physical, more Data
2575 Set without Components can be stored in the same physical Data Set if appropriate.

2576 The current VTL version does not include operators which produce other output data types,
2577 for example there are not operators which manipulate Sets (however this is a possible future
2578 development).

2579 As a matter of fact, the Data Set at the moment is the only type of Artefact that can be
2580 produced and stored permanently through a command of the language.

2581 The Names

2582 The artefact names

2583 The names are the labels which identify the “named” artefacts which are operands or result of
2584 the transformations.

2585 For ensuring the correctness of the VTL operations, it is important to distinguish the names
2586 from the scalar literals when the expression is parsed. For this purpose, the disambiguation
2587 mechanism that distinguishes the types of the scalar literals must also be able of
2588 distinguishing names and scalar literals.

2589 As already mentioned in the section about the scalar literals, the VTL does not prescribe any
2590 predefined disambiguation mechanism, leaving different VTL systems free to using they
2591 preferred or already existing ones. In these VTL manuals, anyway, there is the need to use
2592 some disambiguation mechanisms in order to explain the behaviour of the VTL operators and
2593 give proper examples. These mechanisms are not intended to be mandatory and therefore,
2594 strictly speaking, they are not part of the VTL standard specifications. If no drawbacks exist,
2595 however, their adoption is encouraged to foster the convergence between possible different
2596 practices. If VTL rules are exchanged, the disambiguation mechanisms should be
2597 communicated to the counterparties, at least if they are different from the one suggested
2598 hereinafter.

2599 The general rules for the names are given below. As said above, these rules can be
2600 personalized (for example restricted) in some implementations (e.g. a particular
2601 implementation can require that an name starts with a letter).

2602 The names are strings of characters no more than 128 characters long and are classified in
2603 regular and non-regular names.

2604 The **regular names**:

- 2605 • can contain alphabetic and numeric characters and the special characters underscore
2606 (`_`) and dot (`.`),
- 2607 • must begin with an alphanumeric character and not with a special character
- 2608 • must contain at least one alphabetic character
- 2609 • cannot be a VTL reserved word

2610 Examples of regular names are *abcdef*, *1ab_cde*, *a.b.c_d_e*, *1234_5*.

2611 The regular names are:

- 2612 • written in the Transformations / Expressions without quoting them
- 2613 • case insensitive

2614 The non-regular names:

- 2615 • can contain alphanumeric characters and, in addition to the underscore and the dot,
2616 any other Unicode character
- 2617 • can contain blanks
- 2618 • can begin with special characters
- 2619 • can contain only numeric characters
- 2620 • can be equal to the VTL reserved words

2621 The non-regular names are:

- 2622 • written in the Transformations / Expressions surrounded by single quotes
2623 • case sensitive

2624 Examples of non-regular names, which therefore are enclosed in single quotes, are '*_abcdef*',
2625 '*1ab-cde*', '*12345*', '*power*' (the first begins with a special character, the second contains the "-"
2626 character that is not allowed, the third contains only numeric characters, the fourth coincides
2627 to a VTL reserved word (the name of the exponentiation operator). These names would not
2628 be recognized by VTL if not enclosed between single quotes.

2629 The **VTL reserved words** (and symbols) are:

- 2630 • the keywords of the VTL-ML and VTL-DL operators and of their parameters (e.g. <- ,
2631 := , # , inner_join, as, using, filter, apply, rename, to, + , - , power, and, or, not, group by,
2632 group except, group all, having ...)
2633 • the names of the classes of VTL artefacts of the VTL-IM (e.g., value, value domain, value
2634 domain subset, set, variable, component, data set, data structure, operator, operand
2635 parameter, transformation ...)
2636 • additional keywords for possible future use like get, put, join, map, mapping, merge,
2637 transcode and the names of commonly used mathematical and statistical functions.
2638

2639 **The environment name**

2640 In order to ensure non-ambiguous definitions and operations, the names of the artefacts must
2641 be unique, meaning that an name cannot be assigned to more than one artefact.

2642 In practice, the unicity of the names is ensured in a certain environment, that can be also
2643 called namespace (i.e. the space in which the names are assigned without ambiguities). For
2644 examples, Institutions (agencies) that operate independently can assign the same name to
2645 different artefacts, therefore they cannot be considered as part of the same environment.

2646 The artefacts which input of a Transformation can come also from other environments than
2647 the one in which the Transformation is defined. In these cases the artefact name must be
2648 accompanied by a **Namespace**, which specifies the Data Set environment, to univocally
2649 identify the artefact to retrieve (for example the Data Set).

2650 Therefore, the reference to an artefact belonging to a different environment assumes the
2651 following form:

2652 *Namespace\Name*

2653 *Namespace* is the name of the environment and *Name* is the name of the artefact within the
2654 environment. The separator is the backslash (\).

2655 When the Namespace is not specified, the artefact is assumed to belong to the same
2656 environment as the Transformation.

2657 The result of a Transformation is always assumed to belong to the same environment as the
2658 Transformation, therefore the specification of the namespace of the result is not allowed.

2659 Within a given environment, the names of all the VTL artefacts (such as Value Domains, Sets,
2660 Variables, Components, Data Sets) are assigned by the users.

2661 Some VTL Operators assume that a VTL environment have certain default names for some
2662 kinds of Variables or Value Domains which are needed to perform the correspondent
2663 operations (for example, the operators which transform the data type of the Measure of the

2664 input Data Sets assign a default name to the resulting Measure, the check operators assign
2665 default names to Components and Value Domains needed to represent the results of the
2666 checks). In the VTL manuals, some definite default names are adopted for explanatory
2667 purposes, however these names are not mandatory and can be personalised if needed. If VTL
2668 rules are exchanged between different VTL systems, the partners of the exchange must be
2669 aware of the names adopted by the counterparties.

2670

2671 **The connection to the persistent storage**

2672 As described in the VTL IM, the Data Set is considered as an artefact at logical level, equivalent
2673 to a mathematical function. A VTL Data Set contains the set of Data Points which are the
2674 instances of the function. Each Data Point is interpreted as an association between a
2675 combination of values of the independent variables (the Identifiers) and the corresponding
2676 values of the dependent variables (the Measures and Attributes).

2677 Therefore, the VTL statements reference the conceptual/logical Data Sets and not the objects
2678 in which they are persistently stored. As already mentioned, there can be any relationships
2679 between the VTL logical Data Sets and the corresponding persistent objects (one VTL Data Set
2680 in one storage object, more VTL Data Sets in one storage object, one VTL Data Set in more
2681 storage objects, more VTL Data Sets in more storage objects). The mapping between the VTL
2682 Data Sets and the storage objects is out of the scope of the VTL and is left to the
2683 implementations.

2684

2686 As mentioned, the VTL is made of Operators, which are the basic operations that the language
 2687 can do. For example, the VTL has mathematical operators (e.g. sum (+), subtraction (-),
 2688 multiplication (*), division (/)...), string operators (e.g. string concatenation, substring ...),
 2689 comparison operators (e.g. equal (=), greater than (>), lesser than (<) ...), logical operators
 2690 (e.g. and, or, not ...) and so on.

2691 An Operator has some input and output Parameters, which are its a-priori unknown operands
 2692 and result, have a definite role in the operation (e.g. dividend, divisor or quotient for the
 2693 division) and correspond to a certain type of artefact (e.g. a “Data Set”, a “Data Set
 2694 Component”, a “scalar Value” ...).

2695 The VTL operators are considered as functions (higher-order functions³³), which manipulate
 2696 one or more input first-order functions (the operands) to produce one output first-order
 2697 function (the result).

2698 Assuming that F is the function corresponding to an operator, that P_o is its output parameter
 2699 and that P_i ($i=1, \dots, n$) are its input parameters, the mathematical form of an operator can be
 2700 written as follows:

$$2701 \quad P_o = F (P_1, \dots, P_n)$$

2702 The function F composes the Parameters P_i to obtain P_o (as mentioned, P_i ($i=1, \dots, n$) and P_o must
 2703 be first order functions). In the common case in which the Parameters are Data Sets, F
 2704 composes the Data Points of the input Data Sets D_i ($i=1, \dots, n$) to obtain the Data Points of the
 2705 output Data Set D_o .

2706 When an Operator is invoked, for each input Parameter an actual argument (operand) is
 2707 passed to the Operator, which returns an argument (result) for the output Parameter.

2708 Each parameter has a data type, which is the data type of the possible arguments that can be
 2709 passed or returned for it. For example, the parameters of a multiplication are of type *number*,
 2710 because only the numbers can be multiplied (in fact for example the strings cannot). For a
 2711 deeper explanation of the data types see the corresponding section.

2712

2713 The categories of VTL operators

2714 The VTL operators are classified according to the following categories.

2715 1. The **VTL standard library** contains the standard VTL operators: they are described in
 2716 detail in the Reference Manual.

2717 On the technical perspective, the standard VTL operations can be divided into the
 2718 following two sub-categories:

³³ A higher-order function is a function which takes one or more other functions as arguments and/or provides another function as result.

- 2719 a. The **core set of operations**; they are the primitive operations, in the sense that all
 2720 the other operations can be defined in their terms. The core operations are:
 2721 i. The operations that accept scalar arguments as operands and return a scalar
 2722 value (for example the sum between numeric scalar values, the
 2723 concatenation between *string* scalar values, a logical operation between
 2724 *boolean* scalar values ...).
 2725 ii. The various kinds of Join operators, which allow to lift the scalar operations
 2726 to the Data Set level, i.e., to compose Data Sets with scalar values or with
 2727 other Data Sets.
 2728 iii. Other special operators which cannot be defined by means of the previous
 2729 two categories (for example the analytical functions).
 2730 b. The **non-core standard operations**; they are standard VTL operations as well but
 2731 are not “primitive” and can be derived from the core operations. Examples of these
 2732 operations are the ones that allow to compose Data Sets and scalar values or Data
 2733 Sets and other Data Sets (besides the various kinds of Join operators and the
 2734 special operators mentioned above). Examples of non-core operations are the sum
 2735 between numeric Data Sets, the concatenation between *string* Data Sets, the logical
 2736 operations between *boolean* Data Sets, the *union* operator, some postfix operators
 2737 like *calc*, *filter*, *rename* (see the Reference Manual).

2738 Most VTL Operators of the standard library (for example numerical, string, logical
 2739 operators and others) can operate both on scalar Values and on Data Sets, and thus
 2740 they have two variants: a scalar and a Data Set variant. The scalar variant is part of the
 2741 VTL core, while the Data Set variant is usually not.

2742 Anyway, VTL users do not need to distinguish between core and non-core operators,
 2743 because in the practice, the use of either these categories of Operators is the same.

- 2744 2. The **user-defined operators** are non-standard VTL operators that can be defined by the
 2745 users in order to enhance and personalize the language if needed. VTL provides a special
 2746 operator, called “*define operator*” (see the Reference Manual), for the creation of user-
 2747 defined operators as well as a special syntax to invoke them.

2748 The input parameters

2749 The input parameters may have various goals and in particular:

- 2750 • identify the model artefacts to be manipulated
- 2751 • specify possible options for the operator behaviour
- 2752 • specify additional scalar values required to perform the operator’s behaviour

2753 For example, in the “Join” operator, the first N parameters identify the Data Sets to be joined
 2754 while the “using” parameter specifies the components on which the join must operate.

2755 Depending on the number of the input parameters, the Operators can be classified in:

- | | | |
|------|---------------|---------------------------------|
| 2756 | Unary | having just one input parameter |
| 2757 | Binary | having two input parameters |
| 2758 | N-ary | having more input parameters |

2759 Examples of unary Operators are the change of sign, the minimum, the maximum, the absolute
 2760 value. Examples of binary Operators are the common arithmetical operators (+, -, *, /).

2761 Examples of N-ary operators are the substring, the string replacement, the Join. It is also
2762 possible the extreme case of operators having zero input parameters (e.g., an operator
2763 returning the current time).

2764 The invocation of VTL operators

2765 Operators have different invocation styles :

2766 ○ **Prefix**, only for unary operators. The operator appears before the operand; the general
2767 forms of invocation is:

2768 *Operator Operand* (e.g. $-D_2$ which changes the sign of D_2)

2769 ○ **Infix**, only for binary operators. The operator symbol appears between the operands;
2770 the general form of invocation is:

2771 *FirstOperand Operator SecondOperand* (e.g. $D_1 + D_2$)

2772 ○ **Postfix**, only for unary operators. The operator appears in square brackets and follows
2773 its operand; the general forms of invocation is:

2774 *Operand [Operator]*

2775 (e.g. DS_2 [**filter** $M_1 > 0$] which selects from Data Set DS_2 only the Data Points having
2776 values greater than zero for measure M_1 and returns such values in the result Data
2777 Set.

2778 Postfix operators are also called “clause operators” or simply “clauses”.

2779 ○ **Functional**, for N-ary operators. The operator is invoked using a functional notation;
2780 the general form of invocation is:

2781 *Operator(IO_1, \dots, IO_N)* where IO_1, \dots, IO_N are the input operands;

2782 For example, the syntax for the exponentiation is *power(base, exponent)* and a possible
2783 invocation to calculate the square of the numeric Data Set D_1 is *power($D_1, 2$)*.

2784 The comma (“,”) is the separator between the operands. Parameter binding is fully
2785 positional: in the invocation, actual parameters are passed to the Operator in the same
2786 positional order as the corresponding formal parameters in the Operator syntax.
2787 Parameters can be mandatory or optional: usually the mandatory ones are in the first
2788 positions and the optional ones in the last positions. An underscore (“_”) must be used
2789 to denote that optional operand is omitted in the invocation; for example, this is a
2790 possible invocation of *Operator1(P_1, P_2, P_3)*, where P_2, P_3 are optional and P_2 is omitted:

2791 ***Operator1** ($IO_1, _ , IO_3$)*.

2792 One or more unspecified operands in the last positions can be simply omitted
2793 (including the relevant commas); for example, if both P_2, P_3 are omitted, the invocation
2794 can be simply:

2795 ***Operator1** (IO_1)*.

2796 ○ **Functional with keywords** (a functional syntax in which some parameters are
2797 denoted by special keywords); in this case, each operator has its own form of
2798 invocation, which is described in the Reference Manual. For example, a possible
2799 invocation of the Join operator is the following:

2841 to the calculated Data Set. The behaviour at the Data Set level deserves a general explanation
2842 which is given in the following sections. Examples of operations at the Component level are:

2843 $D_r := D_1 [\text{calc } C_3 := C_1 + C_2]$ C_1 and C_2 are numeric Components of D_1
2844 $D_r := D_1 [\text{calc } C_3 := C_1 + 7]$ C_1 is a numeric Component of D_1 , 7 is a scalar
2845 $D_r := D_3 [\text{calc } C_6 := C_4 || C_5]$ C_4 and C_5 are string Components of D_3
2846 $D_r := D_3 [\text{calc } C_6 := C_4 || \text{"fghij"}]$ C_4 is a string Component of D_3 , "fghij" is a scalar
2847 $D_r := D_3 [\text{calc } C_6 := C_4 || \text{"fghij"}]$ C_4 is a string Component of D_3 , "fghij" is a scalar
2848 $D_r := D_3 [\text{calc } C_6 := C_4 || \text{"fghij"}]$ C_4 is a string Component of D_3 , "fghij" is a scalar
string

2849 In these examples, the postfix operator *calc* is applied to the input Data Sets D_1 and D_3 , takes
2850 in input some of their components and produces in output the components C_3 and C_6
2851 respectively, which become part of the result Data Set D_r .

2852 The operations at a component level are performed row by row and in the context of one
2853 specific Data Set, so that one input Data point results in no more than one output Data Point.

2854 In these last examples the assignment is used both at the Data Set level (when the outcome of
2855 the expression is assigned to the result Data Set) and at the Component level (when the
2856 outcome of the operations at the Component level is assigned to the resulting Components).
2857 The assignment at Data Set level can be either persistent or non-persistent, while the
2858 assignment at the Component level can be only non-persistent, because a Component exists
2859 only within a Data Set and cannot be stored on its own.

2860 The Operators' behaviour

2861 As mentioned, the behaviour of the VTL operators is always functional, i.e., they behave as
2862 higher-order mathematical functions, which manipulate one or more input first-order
2863 functions (the operands) to produce one output first-order function (the result).

2864 The Join operators

2865 The more general and powerful behaviour is supplied by the Join operators, which operates at
2866 Data Set level and allow to compose one or more Data Sets in many possible ways.

2867 In particular, the Join operators allow to:

- 2868 • match the Data Points of the input Data Sets by means of various matching options
2869 (inner/left/full/cross join) and by specifying the Components to match ("using"
2870 clause). For example the sentence

2871 $\text{inner_join } D_1, D_2 \text{ using } [\text{reference_date}, \text{geo_area}]$

2872 matches the Data Points of D_1 , D_2 which have the same values for the Identifiers
2873 *reference_date* and *geo_area*.

- 2874 • filter the result of the match according to a condition, for example the sentence

2875 $\text{filter } D_1 \# M_1 > 0$

2876 maintains the matched Data Points for which the Measure M_1 of D_1 is positive.

- 2877 • aggregate according to the values of some Identifier, for example the sentence

2878 $\text{group by } [Id_1, Id_2]$

2879 eliminates the Identifiers save than Id_1 and Id_2 and aggregate the Data Points having
2880 the same values for Id_1 and Id_2

2881 • combine homonym measures of the matched Data Points according to a formula, for
2882 example the sentence

2883 *apply $D_1 + D_2$*

2884 sums the homonymous Measures of the matched Data Points of D_1 and D_2

2885 • calculate new Components (or new values for existing Components) according to the
2886 desired formulas, also assigning or changing the Component role (Identifier, Measure,
2887 Attribute), for example:

2888 *calc measure $M_3 := M_1 + M_2$, attribute $A_1 := A_2 // A_3$*

2889 calculates the Measure M_3 and the Attribute A_1 according to the formulas above

2890 • keep or drop the specified Measures or Attributes, for example the sentence

2891 *keep $[M_1, M_3, A_1]$*

2892 maintains only the specified measures and attributes, instead the sentence

2893 *drop $[M_2, A_2, A_3]$*

2894 drops only the specified measures and attributes

2895 • rename the specified Components, for example:

2896 *rename $[M_1$ to M_{10} , I_1 to $I_{10}]$*

2897 As shown above, the Join operator, together with the other operators applied at scalar or at
2898 Component level, allows to reproduce the behaviour of the other operators at a Data Set level
2899 (save than some special operator) and also to achieve many other behaviours which are
2900 impossible to achieve otherwise.

2901 Anyway, even if the *join* would cover most of the VTL manipulation needs, the VTL provides
2902 for a number of other Operators which are designed to support the more common
2903 manipulation needs in a simpler way, in order to make the use of the VTL simpler in the more
2904 recurrent situations. Their features are naturally more limited than the ones of the *join* and a
2905 number of default behaviours are assumed.

2906 The following sections explain the more common default behaviours of the Operators other
2907 than the Join.

2908 **Other operators: default behaviour on Identifiers, Measures and Attributes**

2909 The default behaviour of the operators other than the Join, when they operate at Data Set
2910 level, is different for Identifiers, Measures and Attributes.

2911 In fact, unless differently specified, the Operators at Data Set level act only on the Values of
2912 the Measures. The Values of Identifiers are usually left unchanged, except for few special
2913 operators specifically aimed at manipulating Identifiers (for example the operators which
2914 make aggregations, either dropping some Identifiers or according the hierarchical links
2915 between the Code Items of an Identifier). The Values of the Attributes, instead, are
2916 manipulated by default through specific Attribute propagation rules explained in a following
2917 section.

2918 For example, considering the Transformation $D_r := ln(D_1)$, the operation is applied for each
2919 Data Point of D_1 , the values of the Identifiers are left unchanged and the values of all the

2920 Measures are substituted by their natural logarithm (it is assumed that the Measures of D_1
2921 are all numerical).

2922 Similarly, considering the simple operation $D_r := D_1 + 7$, the addition is done for each Data
2923 Point of D_1 , the values of the Identifiers are left unchanged and the number 7 is added to the
2924 values of all the Measures (it is assumed that the Measures of D_1 are all numerical).

2925 As for the structure, like in the examples above, the Identifiers of the result Data Set D_r are the
2926 same as the Identifiers of the input Data Set D_1 (save for the special operators specifically
2927 aimed at manipulating Identifiers), and by default also the Measures of D_r remain the same as
2928 D_1 (save for the operator which change the basic scalar type of the operand, this case is
2929 described in a following section). The Attribute Components of the result depend instead on
2930 the Attribute propagation rule.

2931 In the previous examples, only one Data Set is passed in input to the Operator (other possible
2932 operands are not Data Sets). The operations on more Data Sets, like $D_r := D_1 + D_2$, behave in
2933 the same way than the operations on one Data Set, save that there is the additional need of a
2934 preliminary matching of the Identifiers of the Data Points of the input Data Sets: the operation
2935 applies on the matched Data Points.

2936 For example, the addition $D_1 + D_2$ above happens between each couple of Data Points, one
2937 from D_1 and the other from D_2 , whose Identifiers match according to a default rule (which is
2938 better explained in a following section). The values of the homonymous Measures of D_1 and D_2
2939 are added, taken respectively from the D_1 and D_2 Data Points (the default rule for composing
2940 the measure is better explained in a following section).

2941 **The Identifier Components and the Data Points matching**

2942 This section describes the default Data Points matching rules for the Operators which operate
2943 at Data Set level and which do not manipulate the Identifiers (for example, the behaviour of
2944 the Operators which make aggregations is not the same, and is described in the Reference
2945 Manual).

2946 As shown in the examples above, the actual behaviour depends also on the number of the
2947 input Data Sets.

2948 If just one input Data Set is passed to the operator, the operation is applied for each input
2949 Data Point and produces a corresponding output Data Point. This case happens for all the
2950 unary operators, which have just one input parameter and therefore cannot operate on more
2951 than one Data Set (e.g. $\ln(D_1)$), and for the invocations of Nary operators in which just one
2952 Data Set is passed to the operator (e.g. $D_1 + 7$).

2953 If more input Data Sets are passed to the operator (e.g. $D_1 + D_2$), a preliminary match
2954 between the Data Points of the various input Data Sets is needed, in order to compose their
2955 measures (e.g. summing them) and obtain the Data Points of the result (i.e. D_r). The default
2956 matching rules envisage that the **Data Points are matched when the values of their
2957 homonimous Identifiers are the same.**

2958 For example, let us assume that D_1 and D_2 contain the population and the gross product of the
2959 United States and the European Union respectively and that they have the same Structure
2960 Components, namely the Reference Date and the Measure Name as Identifier Components,
2961 and the Measure Value as Measure Component:

2962

$D_1 =$ United States Data

2963

2964

2965

2966

2967

<i>Ref.Date</i>	<i>Meas.Name</i>	<i>Meas.Value</i>
2013	Population	200
2013	Gross Prod.	800
2014	Population	250
2014	Gross Prod.	1000

2968

$D_2 =$ European Union Data

2969

2970

2971

2972

2973

2974

<i>Ref.Date</i>	<i>Meas.Name</i>	<i>Meas.Value</i>
2013	Population	300
2013	Gross Prod.	900
2014	Population	350
2014	Gross Prod.	1000

2975

The desired result of the sum is the following:

2976

$D_r =$ United States + European Union

2977

2978

2979

2980

2981

2982

<i>Ref.Date</i>	<i>Meas.Name</i>	<i>Meas.Value</i>
2013	Population	500
2013	Gross Prod.	1700
2014	Population	600
2014	Gross Prod.	2000

2983

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).

2984

2985

2986

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.

2987

2988

2989

However, various Data Set operations are possible also under a more **relaxed constraint**, which is when the Identifier Components of one Data Set are a superset of those of the other Data Set.³⁴

2990

2991

2992

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):

2993

2994

³⁴ This corresponds to the "outer join" form of the join expressions, explained in details in the Reference Manual.

2995

$D_1 = \text{European Countries}$

2996

<i>Ref.Date</i>	<i>Country</i>	<i>Population</i>
2012	U.K.	60
2012	Germany	80
2013	U.K.	62
2013	Germany	81

2997

2998

2999

3000

3001

$D_2 = \text{Europe}$

3002

<i>Ref.Date</i>	<i>Population</i>
2012	480
2013	500

3003

3004

3005

3006 In order to calculate the percentage of the population of each single country on the total of
3007 Europe, the Transformation will be:

3008

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

3009 The Data Points will be matched according to the Identifier Components common to D_1 and D_2
3010 (in this case only the *Ref.Date*), then the operation will take place.

3011 The result Data Set will have the Identifier Components of both the operands:

3012

$D_r = \text{European Countries} / \text{Europe} * 100$

3013

<i>Ref.Date</i>	<i>Country</i>	<i>Population</i>
2013	U.K.	12.5
2013	Germany	16.7
2014	U.K.	12.4
2014	Germany	16.2

3014

3015

3016

3017

3018

3019 When the relaxed constraint is applied, therefore, the Data Points are matched when the
3020 values of their **common** Identifiers are the same.

3021 More formally, let F be a generic n-ary VTL Data Set Operator, D_r the result Data Set and D_i
3022 ($i=1, \dots, n$) the input Data Sets, so that: $D_r := F(D_1, D_2, \dots, D_n)$

3023 The “strict” constraint requires that the Identifier Components of the D_i ($i=1, \dots, n$) are the same.
3024 The result D_r will also have the same Identifier components.

3025 The “relaxed” constraint requires that at least one input Data Set D_k exists such that for each
3026 D_i ($i=1, \dots, n$) the Identifier Components of D_i are a (possibly improper) subset of those of D_k . The
3027 output Data Set D_r will have the same Identifier Components than D_k .

3028 The n-ary Operator F will produce the Data Points of the result by matching the Data Points of
3029 the operands that share the same values for the common Identifier Components and by
3030 operating on the values of their Measure Components according to its semantics.

3031 The actual constraint for each operator is specified in the Reference Manual.

3032 Naturally, it is possible that not all the Data Sets contain the same combinations of values of
3033 the Identifiers to be matched. In these cases the match does not happen, the operation is not
3034 performed and no output Data Point is produced. In other words, the measures
3035 corresponding to the missing combinations of Values of the Identifiers are assumed to be
3036 unknown and to have the value NULL, therefore the result of the operation is NULL as well
3037 and the output Data Point is not produced.

3038 This default matching behaviour is the same as the one of the *inner join* Operator, which
3039 therefore is able to perform the same operation. The join operation equivalent to $D_1 + D_2$ is:

3040
$$\text{inner_join} (D_1, D_2 \text{ apply } D_1 + D_2)$$

3041 Different matching behaviours can be obtained through the use of the other *join* Operators,
3042 for example writing:

3043
$$\text{full_join} (D_1, D_2 \text{ apply } D_1 + D_2)$$

3044 the *full join* returns in the output also the combination of Values of the Identifiers which are
3045 only in one Data Set, the operation is applied considering the missing value of the Measure as
3046 the neutral element of the operation to be done (e.g. 0 for the sum, 1 for the product, empty
3047 string for the string concatenation ...) and the output Data Point is produced.

3048 The operations on the Measure Components

3049 This section describes the default composition of the Measure Components for the Operators
3050 which operate at Data Set level and which do not change the basic scalar type of the input
3051 Measure (for example, the behaviour of the Operators which convert one type in another, say
3052 for example a *number* in a *string*, is not the same and is described in a following section).

3053 As shown in the examples below, the actual behaviour depends also on the number of the
3054 input Data Sets and the number of their Measures.

3055 An **Operator applied to one mono-measure Data Set** is intended to be applied to the only
3056 Measure of the input Data Set. The result Data Set will have the same Measure Component,
3057 whose values are the result of the operation.

3058 For example, let us assume that D_1 contains the salary of the employees (the only Identifier is
3059 the Employee ID and the only Measure is the Salary):

3060

3061 $D_1 = \text{Salary of Employees}$

3062

<i>Employee ID</i>	<i>Salary</i>
A	1000
B	1200
C	800
D	900

3063

3064

3065

3066

3067

3068 The Transformation $D_r := D_1 * 1.10$ applies to the only Measure (the salary)
3069 and calculates a new value increased by 10%, so the result will be:

3070

3071

D_r = Increased Salary of Employees

3072

<i>Employee ID</i>	<i>Salary</i>
A	1100
B	1320
C	880
D	990

3073

3074

3075

3076

3077

3078 In case of **Operators applied to one multi-measure Data Set**, by default the operation is
3079 performed on all its Measures. The result Data Set will have the same Measure Components as
3080 the operand Data Set.

3081 For example, given the import and export and number of operations by reference date:

3082

D_1 = Import, Export, Operations

3083

<i>Ref.Date</i>	<i>Import</i>	<i>Export</i>	<i>Operations</i>
2011	1000	1200	5000
2012	1300	1100	6400
2013	1200	1300	4800

3084

3085

3086

3087 The Transformation $D_r := D_1 * 0.80$ applies to all the Measures (e.g. to the
3088 Import, the Export and the Balance) and calculates their 80%:

3089

D_r = 80% of Import & Export

3090

<i>Ref.Date</i>	<i>Import</i>	<i>Export</i>	<i>Operations</i>
2011	800	960	4000
2012	1040	880	5120
2013	960	1040	3840

3091

3092

3093

3094

3095 An Operator can be applied only on Measures of a certain basic data type, corresponding to its
3096 semantics³⁵. For example, *the multiplication* requires the Measures to be of type *number*,
3097 while the *substring* will require them to be *string*. Expressions which violate this constraint
3098 are considered in error.

3099 In general, all the Measures of the Operand Data Set must be compatible with the allowed data
3100 types of the Operator, otherwise (i.e. if at least one Measure is incompatible) the operation is
3101 not allowed. The possible input data types of each operator are specified in the Reference
3102 Manual.

³⁵ As obvious, the data type depends on the parameter for which the Data Set is passed

3103 Therefore, the operation of the previous example ($D_r := D_1 * 0.80$), which is assumed to act on
 3104 all the Measures of D_1 , would not be allowed and would return an error if D_1 would contain
 3105 also a Measure which is not *number* (e.g. *string*).

3106 In case of inputs having Measures of different types, the operation can be done either using
 3107 the *join* operators, which allows to calculate each measure with a different formula (see the
 3108 *calc* operator) or, in two steps, first keeping only the Measures of the desired type and then
 3109 applying the desired compliant operator; the explanation, as explained in the following cases.

3110 If there is the need to **apply an Operator only to one specific Measure**, the membership (#)
 3111 operator can be used, which allows keeping just one specific Components of a Data Set. The
 3112 syntax is: *dataset_name#component_name* (for a better description see the corresponding
 3113 section in the Part 2).

3114 For example, in the Transformation $D_r := D_1\#Import * 0.80$
 3115 the operation keeps only the Import and then calculates its 80%):

3116 $D_r = 80\%$ of the Import

<i>Ref.Date</i>	<i>Import</i>
2011	800
2012	1040
2013	960

3122 If there is the need to **apply an Operator only to some specific Measures**, the *keep*
 3123 operator (or the *drop*)³⁶ can be used, which allows keeping in the result (or dropping) the
 3124 specified Measures (or also Attributes) of the input Data Set. Their invocations are:

3125 *dataset_name [keep component_name , component_name ...]*
 3126 *dataset_name [drop component_name , component_name ...]*

3127 For example, in the Transformation $D_r := D_1[keep Import, Export] * 0.80$
 3128 the operation keeps only the Import and the Export and then calculates its 80%):

3129 $D_r = 80\%$ of the Import

<i>Ref.Date</i>	<i>Import</i>	<i>Export</i>
2011	800	960
2012	1040	880
2013	960	1040

3135 If there is the need to **perform some operations on some specific Measures and keep the**
 3136 **others measures unchanged**, the *calc* operator can be used, which allows to calculate each

³⁶ to preserve the functional behaviour *keep* and *drop* can be applied only on Measures and Attributes, for a deeper description of these operators see the corresponding section in the Reference Manual

3137 Measure with a dedicated formula leaving the other Measures as they are. A simple kind of
3138 invocation is³⁷:

3139 $dataset_name [calc \ component_name ::= cmp_expr, \ component_name ::= cmp_expr \dots]$

3140 The component expressions (*cmp_expr*) can reference only other components of the input
3141 Data Set.

3142 For example, in the Transformation $D_r := D_1 [calc \ Import * 0.80, \ Export * 0.50]$

3143 the operations apply only to Import and Export (and calculate their 80% and 50%
3144 respectively), while the Operations values remain unchanged:

3145 $D_r = 80\%$ of the Import, 50% of the Export and Operations

3146

<i>Ref.Date</i>	<i>Import</i>	<i>Export</i>	<i>Operations</i>
2011	800	1200	5000
2012	1040	1100	6400
2013	960	1300	4800

3147
3148
3149

3150

3151 In case of **Operators applied on more Data Sets**, by default **the operation is performed**
3152 **between the Measures having the same names** (in other words, on the same Measures). To
3153 avoid ambiguities and possible errors, the input Data Sets must have only these Measures and
3154 the result Data Set is assumed to have only those Measures.

3155 For example, let us assume that D_1 and D_2 contain the births and the deaths of the United
3156 States and the European Union respectively.

3157 $D_1 =$ Births & Deaths of the United States

3158

<i>Ref.Date</i>	<i>Births</i>	<i>Deaths</i>
2011	1000	1200
2012	1300	1100
2013	1200	1300

3159
3160
3161

3162 $D_2 =$ Birth & Deaths of the European Union

3163

<i>Ref.Date</i>	<i>Births</i>	<i>Deaths</i>
2011	1100	1000
2012	1200	900
2013	1050	1100

3164
3165
3166

3167

3168 The Transformation $D_r := D_1 + D_2$ will produce:

3169 $D_r =$ Births & Deaths of United States + European Union

³⁷ The *calc* Operator can be used also to calculate Attributes: for a more complete description of this operator see the corresponding section in the Reference Manual

3170
3171
3172
3173
3174

<i>Ref.Date</i>	<i>Births</i>	<i>Deaths</i>
2011	2100	2200
2012	2500	2000
2013	2250	2400

3175 The Births of the first Data Set will be summed up with the Births of the second to calculate
3176 the Births of the result (and the same for the Deaths).

3177 If there is the need to **apply an Operator on Measures having different names**, the
3178 “rename” operator can be used to make their names equal (for a complete description of the
3179 operator see the corresponding section in the Part 2).

3180 For example, given these two Data Sets:

3181 D_1 (Residents in the United States)

3182
3183
3184
3185

<i>Ref.Date</i>	<i>Residents</i>
2011	1000
2012	1300
2013	1200

3186
3187

D_2 (Inhabitants of the European Union)

3188
3189
3190
3191

<i>Ref.Date</i>	<i>Inhabitants</i>
2011	1100
2012	1200
2013	1050

3192

3193 A Transformation for calculating the population of United States + European Union is:

3194 $D_r := D_1[\text{rename Residents to Population}] + D_2[\text{rename Inhabitants to Population}]$

3195 The result will be:

3196 D_r (Population of United States + European Union)

3197
3198
3199
3200

<i>Ref.Date</i>	<i>Population</i>
2011	2100
2012	2500
2013	1250

3201

3202 Note again that the number and the names of the Measure Components of the input Data Sets
3203 are assumed to match (following their possible renaming), otherwise the invocation of the
3204 Operator is considered in error.

3205 To avoid a potentially excessive renaming, and only when just one component is explicitly
3206 specified for each dataset by using the *membership* notation, the VTL allows the operation
3207 even if the names are different. For instance, this operation is allowed:

3208 $D_r := D_1\#Residents + D_2\#Inhabitants$

3209 The result Data Set would have a single Measure named like the Measure of the leftmost
3210 operand (i.e. *Residents*), which in turn can be renamed, if convenient:

3211 $D_r := (D_1\#Residents + D_2\#Inhabitants)[rename Residents to Population]$

3212 The following options and prescription, already described for the operations on just one
3213 multi-measure Data Sets, are valid also for operations on two (or more) multi-measure Data
3214 Sets and are repeated here for convenience:

- 3215 • If there is the need to **apply an Operator only to specific Measures**, it is possible first to
3216 apply the *membership*, *keep* or *drop* operators to the input Data Sets in order to maintain
3217 only the needed Measures, and then the desired operation can be performed.
- 3218 • If there is the need to **apply some Operators to some specific Measures and keep the**
3219 **other ones unchanged**, one of the *join* operators can be used (the choice depends on the
3220 desired matching method). The *join* operations, in fact, provides also for a *calc* option
3221 which can be invoked and behaves exactly like the *calc* operator explained above.
- 3222 • Even in the case of operations on more than one Data Set, all the Measures of the input
3223 Data Sets must be compatible with the allowed data types of the Operator³⁸, otherwise (i.e.
3224 even if only one Measure is incompatible) the operation is not allowed.

3225 In conclusion, the operation is allowed if the input Data Sets have the same Measures and
3226 these are all compliant with the input data type of the parameter which the Data Sets are
3227 passed for.

3228 **Operators which change the basic scalar type**

3229 Some operators change the basic data type of the input Measure (e.g. from *number* to *string*,
3230 from *string* to *date*, from *number* to *boolean* ...). Some examples are the *cast* operator which
3231 converts the data types, the various *comparison* operators whose output is always *boolean*,
3232 the *length* operator which returns the length of a string.

3233 When the basic data type changes, also the Measure must change, because a Variable (in this
3234 case used with the role of Measure in a Data Structure) has just one type, which is the same
3235 wherever the Variable is used³⁹.

3236 Therefore, when an operator which changes the basic scalar type is applied, the output
3237 Measure cannot be the same as the input Measure. In these cases, the VTL systems must
3238 provide for a default Measure Variable for each basic data type to be assigned to the output
3239 Data Set, which in turn can be changed (renamed) by the user if convenient.

3240 The VTL does not prescribe any predefined name or representation for the default Measure
3241 Variable of the various scalar types, leaving different organisations free to using they

³⁸ As obvious, the data type depends on the parameters for which the Data Set are passed

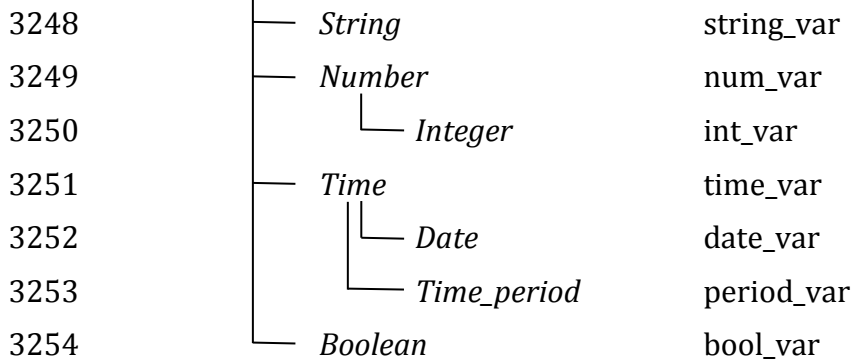
³⁹ In fact according to the IM, a Variable takes values in one Value Domain which represents just one basic data type, independently of where the Variable or the Value Domain are used (e.g. if they have the same type everywhere)

3242 preferred or already existing ones. Therefore the definition of the default Measure Variables
 3243 corresponding to the VTL basic scalar types is left to the VTL implementations.

3244 In the VTL manuals, just for explanatory purposes, the following default Measures will be
 3245 used:

3246

3247 **Basic Scalar Types** **Default Measure Variable**



3255 In some cases, in the examples of the Manuals, the default Boolean variable is also called
 3256 “condition”,

3257 When the operators which change the basic data type of the input Measure are applied
 3258 directly at Data Set level, the VTL does not allow to perform multi-Measure operations. In
 3259 other words, the input Data Set cannot have more than one Measure. In case it has more
 3260 Measures, a single Measure must be selected, for example by means of the *membership*
 3261 operator (e.g. dataset_name#measure_name).

3262 The multi-measure operations remain obviously possible when the operators which change
 3263 the basic data type of the input Measure are applied at Component Level, for example by using
 3264 the *calc* operator.

3265 For example, taking again the example of import, export and number of operations by
 3266 reference date:

3267 $D_1 = \text{Import_Export_Operations}$

<i>Ref.Date</i>	<i>Import</i>	<i>Export</i>	<i>Operations</i>
2011	1000	1200	5000
2012	1300	1100	6400
2013	1200	1300	4800

3272

3273 and assuming that the conversion from number to string of all the Measure Variables is
 3274 desired, the following statement expressed at Data Set level *cast (D₁, string)* is not allowed
 3275 because the Data Set D_1 is multi-measure, while the following one, which makes the
 3276 conversion at the Component level, is allowed:

```

  3277 D1 [ calc
  3278     import_string := cast (import, string)
  3279     , export_string := cast (export, string)
  3280     , operations_string := cast ( operations, string )
  
```



```

3281         ]
3282 For completeness, it is worth saying that also the various Join operators allow the same
3283 operation that, for example, for the inner join would be written as:
3284         inner_join ( D1 calc
3285                     import_string := cast (import, string)
3286                     , export_string := cast (export, string)
3287                     , operations_string := cast ( operations, string )
3288                 )

```

3289 The join operators is designed primarily to act on many Data Sets and allow applying these
3290 operations also when more Data Sets are joined.

3291 **Boolean operators**

3292 The Boolean operators (*and, or, not ...*) take in input boolean Measures and return boolean
3293 Measures. The VTL Boolean operators behave like the operators which change the basic scalar
3294 type: if applied at the Data Set level they are allowed only on mono-measure Data Sets, if
3295 applied at the Component level they are allowed on mono and multi-measure Data Sets.

3296 **Set operators**

3297 The Set operators (*union, intersection ...*) apply the classical set operations (union,
3298 intersection, difference, symmetric differences) to the input Data Sets, considering them as
3299 mathematical functions (sets of Data Points).

3300 These operations are possible only if the Data Sets to be operated have the same data
3301 structure, i.e. the same Identifiers, Measures and Attributes.

3302 For these operators the rules for the Attribute propagation are not applied and the Attributes
3303 are managed like the Measures.

3304 The Data Points common (or not common) to the input Data Sets are determined by taking
3305 into account only the values of the Identifiers: the common Data Points are the ones which
3306 have the same values for all the Identifiers.

3307 If for a common Data Point one or more dependent variables (Measures and Attributes) have
3308 different values in different Data Sets, the Data Point of the leftmost Data Set are returned in
3309 the result.

3310 **Behaviour for Missing Data**

3311 The awareness of missing data is very important for correct VTL operations, because the
3312 knowledge of the Data Points of the result depends on the knowledge of the Data Points of the
3313 operands. For example, assume $D_r := D_1 + D_2$ and suppose that some Data Points of D_2
3314 are unknown, it follows that the corresponding Data Points of D_r cannot be calculated and
3315 are unknown too.

3316 Missing data are explicitly represented when some Measures or Attributes of a Data Point
3317 have the value "NULL", which denotes the absence of a true value (the "NULL" value is not
3318 allowed for the Identifier Components, in order to ensure that the Data Points are always
3319 identifiable).

3320 Missing data may also show as the absence of some expected Data Point in the Data Set. For
3321 example, given a Data Set containing the reports to an international organization relevant to
3322 different countries and different dates, and having as Identifier Components the Country and
3323 the Reference Date, this Data Set may lack the Data Points relevant to some dates (for example
3324 the future dates) or some countries (for example the countries that didn't send their data) or
3325 some combination of dates and countries.

3326 The absence of Data Points, however, does not necessarily denote that the phenomenon under
3327 measure is unknown. In some cases, in fact, it means that a certain phenomenon did not
3328 happen.

3329 The handling of missing Data Points in VTL operations can be handled in several ways. One
3330 way is to require all participating Data Points used in a computation to be present and known,
3331 this is the correct approach if the absence of a Data Point means that the phenomenon is
3332 unknown and corresponds with the matching method of the *inner join* operator. Another way
3333 is to allow some, but not all, Data Points to be absent, when the absence does not mean that
3334 the phenomenon is unknown; this corresponds to the behaviour of the left and full join
3335 Operator.

3336 On the basic level, most of the scalar operations (arithmetic, logical, and others) return `NULL`
3337 when any of their arguments is `NULL`.

3338 The general properties of the `NULL` are the following ones:

- 3339 • **Data type:** the `NULL` value is the only value of multiple different types (i.e., all the
3340 nullable scalar types).
- 3341 • **Testing.** A built-in Boolean operator **is null** can be used to test if a scalar value is `NULL`.
- 3342 • **Comparisons.** Whenever a `NULL` value is involved in a comparison (`>`, `<`, `>=`, `<=`, `in`, `not`
3343 `in`, `between`) the result of the comparison is `NULL`.
- 3344 • **Arithmetic operations.** Whenever a `NULL` value is involved in a mathematical
3345 operation (`+`, `-`, `*`, `/`, `...`), the result is `NULL`.
- 3346 • **String operations.** In operations on Strings, `NULL` is considered an empty String (`""`).
- 3347 • **Boolean operations.** VTL adopts 3VL (three-value logic). Therefore the following
3348 deduction rules are applied:
3349 `TRUE or NULL → TRUE`
3350 `FALSE or NULL → NULL`
3351 `TRUE and NULL → NULL`
3352 `FALSE and NULL → FALSE`
- 3353 • **Conditional operations.** The `NULL` is considered equivalent to `FALSE`; for example in
3354 the control structures of the type (*if (p) -then -else*), the action specified in *-then* is
3355 executed if the predicate *p* is `TRUE`, while the action *-else* is executed if the *p* is `FALSE`
3356 or `NULL`;
- 3357 • **Filter clauses.** The `NULL` is considered equivalent to `FALSE`; for example in the filter
3358 clause [*filter p*], the Data Points for which the predicate *p* is `TRUE` are selected and
3359 returned in the output, while the Data Points for which *p* is `FALSE` or `NULL` are
3360 discarded.
- 3361 • **Aggregations.** The aggregations (like *sum*, *avg* and so on) return one Data Point in
3362 correspondence to a set of Data Points of the input. In these operations, the input Data
3363 Points having a `NULL` value are in general not considered. In the average, for example,

3364 they are not considered both in the numerator (the sum) and in the denominator (the
3365 count). Specific cases for specific operators are described in the respective sections.
3366 • **Implicit zero.** Arithmetic operators assuming implicit zeros (+,-,*,/) may generate
3367 NULL values for the Identifier Components in particular cases (superset-subset relation
3368 between the set of the involved Identifier Components). Because NULL values are in
3369 general forbidden in the Identifiers, the final outcome of an expression must not
3370 contain Identifiers having NULL values. As a momentary exception needed to allow
3371 some kinds of calculations, Identifiers having NULL values are accepted in the partial
3372 results. To avoid runtime error, possible NULL values of the Identifiers have to be fully
3373 eliminated in the outcome of the expression (through a selection, or other operators),
3374 so that the operation of “assignment” (:=) does not encounter them.
3375

3376 If a different behaviour is desired for NULL values, it is possible to **override** the default
3377 behaviour. This can be achieved with the combination of the *calc* clauses and *is null*
3378 operators.

3379 For example, suppose that in a specific case the NULL values of the Measure Component M_1 of
3380 the Data Set D_1 have to be considered equivalent to the number 1, the following
3381 Transformation can be used to multiply the Data Sets D_1 and D_2 , preliminarily converting
3382 NULL values of $D_1.M_1$ into the number 1. For detailed explanations of *calc* and *is null* refer to
3383 the specific sections in the Reference Manual.

3384
$$D_r := D_1 [M_1 := \text{if } M_1 \text{ is NULL then } 1 \text{ else } M_1] * D_2$$

3385 Behaviour for Attribute Components

3386 Given an invocation of one Operator F , which can be written as $D_r := F(D_1, D_2, \dots, D_n)$, and
3387 considering that the input Data Sets D_i ($i=1, \dots, n$) may have any number of Attribute
3388 Components, there can be the need of calculating the desired Attribute Components of D_r .
3389 This Section describes the general VTL assumptions about how Attributes are handled (the
3390 specific behaviours of the various operators are described in the Reference Manual).

3391 It should be noted that the Attribute Components of a Data Set are dependent variables of the
3392 corresponding mathematical function, just like the Measures. In fact, the difference between
3393 Attribute and Measure Components lies only in their meaning: it is implicitly intended that the
3394 Measures give information about the real world and the Attributes about the Data Set itself
3395 (or some part of it, for example about one of its measures), however the real uses of the
3396 Attribute Components are very heterogeneous.

3397 The VTL has different default behaviours for Attributes and for Measures, to comply as much
3398 as possible with the relevant manipulation needs.

3399 At the Data Set level, the VTL Operators manipulate by default only the Measures and not the
3400 Attributes.

3401 At the Component level, instead, Attributes are calculated like Measures, therefore the
3402 algorithms for calculating Attributes, if any, can be specified explicitly in the invocation of the
3403 Operators. This is the behaviour of clauses like *calc*, *keep*, *drop*, *rename*, and so on, either
3404 inside or outside the *join* (see the detailed description of these operators in the Reference
3405 Manual).

3406 **The Attribute propagation rule**

3407 The users which want also to automatize the propagation of the Attributes' Values when no
3408 operation is explicitly defined can optionally enforce a mechanism, called Attribute
3409 Propagation rule, whose behaviour is explained here. The adoption of this mechanism is
3410 optional, users are free to allow the attribute propagation rule or not. The users that do not
3411 want to allow Attribute propagation rules simply will not implement what follows.

3412 The **Attribute propagation rule** is made of two main components, namely the “virality” and
3413 the “default propagation algorithm”.

3414 The “**virality**” is a characteristic to be assigned to the Attributes Components which
3415 determines if the Attribute is propagated automatically in the result or not: a “**viral**” Attribute
3416 is propagated while a “**non-viral**” Attribute is not (being a default behaviour, the virality is
3417 applied when no explicit indication about the keeping of the Attribute is provided in the
3418 expression). If the virality is not defined, the Attribute is considered as non-viral.

3419 The virality is also assigned to the Attribute propagated in the result Data Set. By default, a
3420 viral Attribute in the input generates an homonymous viral Attribute also in the result. Vice-
3421 versa, by default a non-viral Attribute in the input generates a non-viral Attribute also in the
3422 result (this happens when the Attribute in the result is calculated through an explicitly
3423 expression but without specifying explicitly its virality). The default assignation of the virality
3424 can be overridden by operations at Component level as mentioned above, for example *keep*
3425 (i.e., to keep a *non-viral* Attribute or not to keep a *viral* one) and *calc* to alter the virality in the
3426 result Data Set, (from *viral* to *non-viral* or vice-versa).⁴⁰

3427 The “**default propagation algorithm**” is the specification of the calculus to be performed to
3428 propagate a viral Attribute when no explicit calculation is defined, always in the context of the
3429 Data Set level operations. A default propagation algorithm should be associated to each
3430 Variable that can assume the role of viral Attribute Component in a Data Set. The default
3431 propagation algorithm is an aggregation function which produces the Attribute's value for a
3432 generic output Data Point starting from the Attribute's values of the input Data Points that
3433 contribute to it. If the Attribute is viral and no default propagation algorithm is provided for it,
3434 the invocation of the Operators at Data Set level is considered in error.

3435 Hence, the **Attribute propagation rule** behaves as follows:

- 3436 • the non-viral Attributes are not kept in the result and their values are not considered;
- 3437 • the viral Attributes of the operands are kept and are considered viral also in the result;
3438 in other words, if an operand has a viral Attribute V, the result will have V as viral
3439 Attribute too;
- 3440 • The Attributes, like the Measures, are combined according to their names, e.g. the
3441 Attributes having the same names in more input Data Sets are combined, while the
3442 Attributes having different names are considered as different Attributes;
- 3443 • Whenever in the application of a VTL operator the input Data Points are not combined
3444 as for their Measures (i.e., one input Data Point can result in no more than one output
3445 Data Point), the values of the viral Attributes are simply copied from the input Data

⁴⁰ In particular the *keep* clause allows the specification of whether or not an attribute is kept in the result while the *calc* clause make it possible to define calculation formulas for specific attributes. They can be used both for Measures and for Attributes and operate on Components of just one Data Set to obtain new Measures / Attributes.

- 3446 Point to the (possible) output Data Point (obviously, this applies always in the case of
 3447 unary Operators which do not make aggregations);
- 3448 • Whenever in the application of a VTL operator two or more Data Points (belonging to
 3449 the same or different Data Sets) are combined as for their Measures to give one output
 3450 Data Point, the default propagation algorithm associated to the viral Attribute is
 3451 applied, producing the Attribute value of the output Data Point. This happens for
 3452 example for the unary Operators which aggregate Data Points and for Operators which
 3453 combine the Data Points of more input Data Sets; in the latter case, the Attributes
 3454 having the same names in such Data Sets are combined.

3456 Extending an example already given for unary Operators, let us assume that D_1 contains the
 3457 salary of the employees of a multinational enterprise (the only Identifier is the Employee ID,
 3458 the only Measure is the Salary, and there are two other Components defined as viral
 3459 Attributes, namely the Currency and the Scale of the Salary):

3461 $D_1 =$ Salary of Employees

3462

<i>Employee ID</i>	<i>Salary</i>	<i>Currency</i>	<i>Scale</i>
A	1000	U.S. \$	Unit
B	1200	€	Unit
C	800	yen	Thousands
D	900	U.K. Pound	Unit

3467

3468 The Transformation $D_r := D_1 * 1.10$ applies only to the Measure (the salary)
 3469 and calculates a new value increased by 10%, the viral Attributes are kept and left unchanged,
 3470 so the result will be:

3471 $D_r =$ Increased Salary of Employees

3472

<i>Employee ID</i>	<i>Salary</i>	<i>Currency</i>	<i>Scale</i>
A	1100	U.S. \$	Unit
B	1320	€	Unit
C	880	yen	Thousands
D	990	U.K. Pound	Unit

3477

3478 The Currency and the Scale of D_r will be considered viral too and therefore would be kept also
 3479 in case D_r becomes operand of other Transformations.

3480 Another example can be given for operations involving more input Data Sets (e.g. $D_r := D_1 +$
 3481 D_2). Let us assume that D_1 and D_2 contain the births and the deaths of the United States and
 3482 the Europe respectively, plus a viral Attribute that qualifies if the Value is estimated or not
 3483 (having values *True* or *False*).

3484

3485 $D_1 =$ Births & Deaths of the United States

3486
3487
3488
3489

<i>Ref.Date</i>	<i>Births</i>	<i>Deaths</i>	<i>Estimate</i>
2011	1000	1200	False
2012	1300	1100	False
2013	1200	1300	True

$D_2 = \text{Birth} \ \&$

3491

Deaths of the European Union

3492
3493
3494
3495

<i>Ref.Date</i>	<i>Births</i>	<i>Deaths</i>	<i>Estimate</i>
2011	1100	1000	False
2012	1200	900	True
2013	1050	1100	False

3496

3497 Suppose that the default propagation algorithm associated to the “Estimate” variable works as
3498 follows:

- 3499 • each value of the Attribute is associated to a default weight;
- 3500 • the result of the combination is the value having the highest weight;
- 3501 • if multiple values have the same weight, the result of the combination is the first in
3502 lexicographical order.

3503 Assuming the weights 1 for “false” and 2 for “true”, the Transformation $D_r := D_1 + D_2$
3504 will produce:

3505

$D_r = \text{Births} \ \& \ \text{Deaths of United States} + \text{European Union}$

3506
3507
3508
3509

<i>Ref.Date</i>	<i>Births</i>	<i>Deaths</i>	<i>Estimate</i>
2011	2100	2200	False
2012	2500	2000	True
2013	2250	2400	True

3510 Note also that:

- 3511 • if the attribute *Estimate* was non-viral in both the input Data Sets, it would not be kept
3512 in the result
- 3513 • if the attribute *Estimate* was viral only in one Data Set, it would be kept in the result
3514 with the same values as in the viral Data Set

3515 In an expression, the default propagation of the Attributes is performed always in the same
3516 order of execution of the Operators of the expression, which is determined by their
3517 precedence and associativity rules, as already explained in the relevant section.

3518 For example, recalling the example already given example:

3519
$$D_r := D_1 + D_2 / (D_3 - D_4 / D_5)$$

3520 The evaluation of the Attributes will follow the order of composition of the Measures:

- 3521 I. $A(D_4 / D_5)$ (default precedence order)
- 3522 II. $A(D_3 - I)$ (explicitly defined order)

- 3523 III. $A(D_2 / II)$ (default precedence order)
 3524 IV. $A(D_1 + III)$ (default precedence order)
 3525

3526 **Properties of the Attribute propagation algorithm**

3527 An Attribute default propagation algorithm is a user-defined operator which has a group of
 3528 Values of an Attribute as operands and returns just one Value for the same Attribute.

3529 An Attribute default propagation algorithm (here called A) must ensure the following
 3530 properties (in respect to the application of a generic Data Set operator “ \S ” which applies on
 3531 the measures):

3532 **Commutative law (1)**

3533 $A(D_1 \S D_2) = A(D_2 \S D_1)$

3534 The application of A produces the same result (in term of Attributes) independently of
 3535 the ordering of the operands. For example, $A(D_1 + D_2) = A(D_2 + D_1)$. This may seem
 3536 quite intuitive for “sum”, but it is important to point out that it holds for every
 3537 operator, also for non-commutative operations like difference, division, logarithm and
 3538 so on; for example $A(D_1 / D_2) = A(D_2 / D_1)$

3539 **Associative law (2)**

3540 $A(D_1 \S A(D_2 \S D_3)) = A(A(D_1 \S D_2) \S D_3)$

3541 Within one operator, the result of A (in term of Attributes) is independent of the
 3542 sequence of processing.

3543 **Reflexive law (3)**

3544 $A(\S(D_1)) = A(D_1)$

3545 The application of A to an Operator having a single operand gives the same result (in
 3546 term of Attributes) that its direct application to the operand (in fact the propagation
 3547 rule keeps the viral attributes unchanged).

3548 With these properties in place, it is always possible to avoid ambiguities and circular
 3549 dependencies in the determination of the Attributes’ values of the result. Moreover, it is
 3550 sufficient without loss of generality to consider only the case of binary operators (i.e. having
 3551 two Data Sets as operands), as more complex cases can be easily inferred by applying the
 3552 Attribute propagation rule recursively (following the order of execution of the operations in
 3553 the VTL expression).

3554 Governance, other requirements and future work

3555 The SDMX Technical Working Group, as mandated by the SDMX Secretariat, is responsible for
3556 ensuring the technical maintenance of the Validation and Transformation Language through a
3557 dedicated VTL task-force. The VTL task-force is open to the participation of experts from
3558 other standardisation communities, such as DDI and GSIM, as the language is designed to be
3559 usable within different standards.

3560 The governance of the extensions and personalisations

3561 According to the requirements, it is envisaged that the language can be enriched and made
3562 more powerful in future versions according to the evolution of the business needs. For
3563 example, new operators and clauses can be added, and the language syntax can be upgraded.

3564 The VTL governance body will take care of the evolution process, collecting and prioritising
3565 the requirements, planning and designing the improvements, releasing future VTL versions.

3566 The release of new VTL versions is considered as the preferred method of fulfilling the
3567 requirements of the user communities. In this way the possibility of exchanging standard
3568 validation and transformation rules would be preserved to the maximum extent possible.

3569 In order to fulfil specific calculation features not yet supported, the VTL provides for an
3570 operator which allows to define new custom operators by means of the existing ones and
3571 another operator (Evaluate) whose purpose is to invoke an external calculation function
3572 (routine), provided that this is compatible with the VTL IM, basic principles and data types.

3573 As already mentioned, because the user-defined operators does not belong to the standard
3574 library, they are not standard VTL operators and are applicable only in the context in which
3575 they have been defined. In particular, if there is the need of applying user-defined operators
3576 in other contexts, their definitions need to be pre-emptively shared.

3577 The operator “Evaluate” (also “Eval”) allows defining and making customized calculations
3578 (also reusing existing routines) without upgrading or extending the language, because the
3579 external calculation function is not considered as an additional operator. The expressions
3580 containing Eval are standard VTL expressions and can be parsed through a standard parser.
3581 For this reason, when it is not possible or convenient to use other VTL operators, Eval is the
3582 recommended method of customizing the language operations.

3583 However, as explained in the section “Extensibility and Customizability” of the “General
3584 Characteristics of VTL” above, calling external functions has some drawbacks in respect to the
3585 use of the proper VTL operators. The transformation rules would be not understandable
3586 unless such external functions are properly documented and shared and could become
3587 dependent on the IT implementation, less abstract and less user oriented. Moreover, the
3588 external functions cannot be parsed (as if they were built through VTL operators) and this
3589 could make the expressions more error-prone. External routines should be used only for
3590 specific needs and in limited cases, whereas widespread and generic needs should be fulfilled
3591 through the operators of the language.

3592 While the “Eval” operator is part of VTL, the invoked external calculation functions are not.
3593 Therefore, they are considered as customized parts under the governance, and are
3594 responsibility and charge of the organizations which use it.

3595 Organizations possibly extending VTL through non-standard operators/clauses would
3596 operate on their own total risk and responsibility, also for any possible maintenance activity
3597 deriving from VTL modifications.

3598 As mentioned, whilst an Organisation adopting VTL can extend its own library by defining
3599 customized parts and by implementing external routines, on its own total responsibility, in
3600 order to improve the standard language for specific purposes (e.g. for supporting possible
3601 algorithms not permitted by the standard part), it is important that the customized parts
3602 remain compliant with the VTL IM and the VTL fundamentals. Adopting Organizations are
3603 totally in charge of any activity for maintaining and sharing their customized parts. Adopting
3604 Organizations are also totally in charge of any possible maintenance activity to maintain the
3605 compliance between their customized parts and the possible standard VTL future evolutions

3606 Relations with the GSIM Information Model

3607 As explained in the section "VTL Information Model", VTL 1.0 is inspired by GSIM 1.1 as much
3608 as possible, in order to provide a formal model at business level against which other
3609 information models can be mapped, and to facilitate the implementation of VTL with
3610 standards like SDMX, DDI and possibly others.

3611 GSIM faces many aspects that are out of the VTL scope; the latter uses only those GSIM
3612 artefacts which are strictly related to the representation of validations and transformations.
3613 The referenced GSIM artefacts have been assessed against the requirements for VTL and, in
3614 some cases, adapted or improved as necessary, as explained earlier. No assessment was made
3615 about those GSIM artefacts which are out of the VTL scope.

3616 In respect to GSIM, VTL considers both unit and dimensional data as mathematical functions
3617 having a certain structure in term of independent and dependent variables. This leads to a
3618 simplification, as unit and dimensional data can be managed in the same way, but it also
3619 introduces some slight differences in data representation. The aim of the VTL Task Force is to
3620 foster the adoption of this adjustment for the next GSIM versions.

3621 The VTL IM allows defining the Value Domains (as in GSIM) and their subsets (not explicitly
3622 envisaged in GSIM), needed for validation purposes. In order to be compliant, the GSIM
3623 artefacts are used for modelling the Value Domains and a similar structure is used for
3624 modelling their subsets. Even in this case, the VTL task force will propose the explicit
3625 introduction of the Value Domain Subsets in future GSIM versions.

3626 VTL is based on a model for defining mathematical expressions which is called
3627 "Transformation model", while GSIM does not have a Transformation model. The VTL IM has
3628 been built on the SDMX Transformation model, with the intention of suggesting its
3629 introduction in future GSIM versions.

3630 Some misunderstanding may arise from the fact that GSIM, DDI, SDMX and other standards
3631 also have a Business Process model. The connection between the Transformation model and
3632 the Business Process model has been neither analysed nor modelled in VTL 1.0. One reason is
3633 that the business process models available in GSIM, DDI and SDMX are not yet fully
3634 compatible and univocally mapped.

3635 It is worth nothing that the Transformation and the Business Process models address
3636 different matters. In fact, the former allows defining validation and calculation rules in the
3637 form of mathematical expressions (like in a spreadsheet) while the latter allows defining a

3638 business process, made of tasks to be executed in a certain order. The two models may
3639 coexist and be used together as complementary. For example, a certain task of a business
3640 process (say the validation of a data set) may require the execution of a certain set of
3641 validation rules, expressed through the Transformation model used in VTL. Further progress
3642 in this reconciliation can be part of the future work on VTL.

3644 The VTL language is also expressed in EBNF (Extended Backus-Naur Form).

3645 EBNF is a standard⁴¹ meta-syntax notation, typically used to describe a Context-Free grammar
3646 and represents an extension to BNF (Backus-Naur Form) syntax. Indeed, any language
3647 described with BNF notation can also be expressed in EBNF (although expressions are
3648 typically lengthier).

3649 Intuitively, the EBNF consists of terminal symbols and non-terminal production rules.
3650 Terminal symbols are the alphanumeric characters (but also punctuation marks, whitespace,
3651 etc.) that are allowed singularly or in a combined fashion. Production rules are the rules
3652 governing how terminal symbols can be combined in order to produce words of the language
3653 (i.e. legal sequences).

3654 More details can be found at http://en.wikipedia.org/wiki/Extended_Backus-Naur_Form

3655 Properties of VTL grammar

3656 VTL can be described in terms of a Context-Free grammar⁴², with productions of the form $V \rightarrow$
3657 w , where V is a single non-terminal symbol and w is a string of terminal and non-terminal
3658 symbols.

3659 VTL grammar aims at being unambiguous. An ambiguous Context-Free grammar is such that
3660 there exists a string that can be derived with two different paths of production rules,
3661 technically with two different leftmost derivations.

3662 In theoretical computer science, the problem of understanding if a grammar is ambiguous is
3663 undecidable. In practice, many languages adopt a number of strategies to cope with
3664 ambiguities. This is the approach followed in VTL as well. Examples are the presence of
3665 *associativity* and *precedence* rules for infix operators (such as addition and subtraction), and
3666 the existence of compulsory *else* branch in *if-then-else* operator.

3667 These devices are reasonably good to guarantee the absence of ambiguity in VTL grammar.
3668 Indeed, real parser generators (for instance YACC⁴³), can effectively exploit them, in particular
3669 using the mentioned associativity and precedence constrains as well as the relative ordering
3670 of the productions in the grammar itself, which solves ambiguity by default.

⁴¹ ISO/IEC 14977

⁴² http://en.wikipedia.org/wiki/Context-free_grammar

⁴³ <http://en.wikipedia.org/wiki/Yacc>