

SDMX Technical Working Group

VTL Task Force

**VTL - version 1.1**  
**(Validation & Transformation Language)**

**Part 2 - Reference Manual**

*(DRAFT FOR PUBLIC REVIEW)*

*October 2016*



## Foreword

31

32

33 The Task force for the Validation and Transformation Language (VTL), created in 2012-2013  
34 under the initiative of the SDMX Secretariat, is pleased to present the draft version of VTL 1.1.

35 The SDMX Secretariat launched the VTL work at the end of 2012, moving on from the  
36 consideration that SDMX already had a package for transformations and expressions in its  
37 information model, while a specific implementation language was missing. To make this  
38 framework operational, a standard language for defining validation and transformation rules  
39 (operators, their syntax and semantics) had to be adopted, while appropriate SDMX formats  
40 for storing and exchanging rules, and web services to retrieve them, had to be designed. The  
41 present VTL 1.1 package is only concerned with the first element, i.e. a formal definition of  
42 each operator, together with a general description of VTL, its core assumptions and the  
43 information model it is based on.

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

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

64 For the reason described above, The VTL specifications are designed at a logical level,  
65 independently of any other standard, including SDMX. The VTL specifications, therefore, are  
66 self-standing and can be implemented either on their own or by other standards (including  
67 SDMX). In particular, the work for the SDMX implementation of VTL is going in parallel to the  
68 work for designing the VTL 1.1 version, and will entail a future update of the SDMX  
69 documentation.

70 The first public consultation on VTL (version 1.0) was held in 2014. Many comments were  
71 incorporated in the VTL 1.0 version, published in March 2015. Other suggestions for  
72 improving the language, received afterwards, fed the discussion for building the present draft  
73 version 1.1, which contains many new features.

74

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

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

84 The present document (part 2) contains the reference manual with the full library of  
85 operators ordered by category.

86 The latest version of VTL is freely available online at [https://sdmx.org/?page\\_id=5096](https://sdmx.org/?page_id=5096)

87

## 88 **Acknowledgements**

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

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

101 Feedback and suggestions for improvement are encouraged and should be sent to the SDMX  
102 Technical Working Group ([twg@sdmx.org](mailto:twg@sdmx.org)).

103

105		
106	<b>Foreword</b> .....	<b>3</b>
107	<b>Table of contents</b> .....	<b>5</b>
108	<b>Introduction</b> .....	<b>9</b>
109	<b>Structure of the document</b> .....	<b>10</b>
110	<b>Diagram of the Operators</b> .....	<b>11</b>
111	<b>List of the Operators/Functions</b> .....	<b>12</b>
112	VTL-DL Operators .....	12
113	VTL-ML Standard Library .....	13
114	Operators and functions applied on Datasets and scalar values .....	13
115	List of standard library operators and functions .....	14
116	VTL-ML - Evaluation order of the Operators .....	21
117	<b>Syntactical conventions</b> .....	<b>22</b>
118	<b>VTL-DL - Artefacts Definition</b> .....	<b>24</b>
119	defineValueDomain .....	24
120	defineValueDomainSubset .....	25
121	defineVariable .....	27
122	defineDataStructure .....	27
123	defineDataset .....	29
124	<b>VTL-DL - Rulesets</b> .....	<b>31</b>
125	define datapoint ruleset .....	31
126	define hierarchical ruleset .....	34
127	define mapping ruleset .....	39
128	<b>VTL-ML - General purpose operators and functions</b> .....	<b>41</b>
129	Parentheses ( ) .....	41
130	Assignment := .....	41
131	Membership .....	41
132	Alias as .....	42
133	alterDataset .....	43
134	get .....	45
135	put .....	49
136	eval .....	50

137	Join expression.....	50
138	join_clause.....	51
139	calc_clause.....	54
140	drop_clause.....	54
141	keep_clause.....	55
142	filter_clause.....	55
143	rename_clause.....	55
144	unfold_clause.....	56
145	fold_clause.....	57
146	Function Creation.....	57
147	<b>VTL-ML - String operators and functions.....</b>	<b>59</b>
148	length.....	59
149	String concatenation   .....	59
150	trim /rtrim/ltrim.....	61
151	upper/lower.....	62
152	substr.....	62
153	instr.....	64
154	date_from_string.....	65
155	replace.....	66
156	<b>VTL-ML - Numeric operators and functions.....</b>	<b>68</b>
157	unary plus +.....	68
158	unary minus -.....	68
159	addition and subtraction + -.....	69
160	multiplication and division * /.....	72
161	round/ceil/floor.....	73
162	abs.....	74
163	trunc.....	76
164	exp.....	77
165	ln.....	78
166	log.....	79
167	power.....	79
168	sqrt.....	80
169	nroot.....	81
170	mod.....	82

171	listsum .....	83
172	<b>VTL-ML - Boolean operators and functions.....</b>	<b>85</b>
173	equal to = .....	85
174	not equal to <>.....	86
175	greater than > >= .....	87
176	less than < <= .....	89
177	in, not in .....	90
178	between .....	91
179	isnull .....	93
180	exists_in, not_exists_in/in_all.....	94
181	match_characters .....	98
182	all.....	99
183	any.....	100
184	unique .....	102
185	func_dep .....	103
186	and .....	104
187	or .....	106
188	xor .....	107
189	not.....	109
190	<b>VTL-ML - Date operators and functions.....</b>	<b>111</b>
191	extract .....	111
192	string from date .....	112
193	current_date .....	113
194	<b>VTL-ML - Set functions .....</b>	<b>114</b>
195	union .....	114
196	intersect .....	117
197	symdiff.....	118
198	setdiff .....	119
199	subscript.....	121
200	transcode .....	122
201	aggregate .....	123
202	<b>VTL-ML - Statistical functions .....</b>	<b>127</b>
203	Aggregate functions .....	127
204	Time aggregate functions.....	130
205	Analytic functions.....	133

206	first_value .....	134
207	lag lead.....	135
208	last_value.....	135
209	ntile.....	136
210	percent_rank .....	137
211	rank.....	138
212	ratio_to_report.....	138
213	hierarchy .....	139
214	<b>VTL-ML - Data validation functions.....</b>	<b>149</b>
215	check .....	149
216	check (with datapoint rulesets).....	149
217	check (with hierarchical rulesets).....	151
218	check ( single rule) .....	152
219	check value domain subset.....	154
220	<b>VTL-ML - Time series functions.....</b>	<b>157</b>
221	fill_time_series.....	157
222	flow_to_stock.....	158
223	stock_to_flow.....	159
224	timeshift.....	160
225	<b>VTL-ML - Conditional operators.....</b>	<b>162</b>
226	if-then-else .....	162
227	nvl.....	164
228	<b>VTL-ML - Clause operators .....</b>	<b>166</b>
229	rename .....	166
230	filter.....	166
231	keep.....	168
232	calc .....	169
233	attrcalc .....	170
234		



236 The VTL 1.1 library of the Operators is described hereinafter. The operators included in this  
237 version of VTL are summarized in the diagrams and tables below.

238 VTL 1.1 is made of two main parts: the VTL Definition Language (VTL-DL) and the VTL  
239 Manipulation Language (VTL-ML).

240 The former (VTL-DL) did not exist in VTL 1.0, because at that time VTL was intended to work  
241 on top of existing standards, like SDMX, DDI, GSIM or others, and therefore the definition of  
242 the artefacts to be manipulated (Data and their structures, Variables, Value Domains and so  
243 on) was assumed to be made using the implementing standards and not VTL itself. In other  
244 words, VTL 1.0 was not meant to define its artefacts and therefore only contained a  
245 manipulation language.

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

254 Therefore, VTL 1.1 has been equipped also with a definition language for VTL artefacts.

255 As for the manipulation language, VTL 1.0 contains a flat list of operators, in principle not  
256 related one another. A main suggestion for VTL 1.1 was to identify a core set of primitive  
257 operators able to express all the other operators of the language. This is in order to specify  
258 more formally the semantics of the available operators, avoiding possible ambiguities about  
259 their behaviour and fostering coherent implementations. The distinction between the core  
260 and standard library is mainly of interest of the VTL technical implementers.

261 The suggestion above has been acknowledged, so that the VTL 1.1 manipulation language is  
262 made of a core set of primitive operators and a standard library of derived operators,  
263 definable in term of the primitive ones. The standard library contains VTL 1.0 operators  
264 (possibly enhanced) and the new operators introduced in VTL 1.1.

265 The VTL core includes a mechanism called FLWOR expressions (For-Let-Where-Order-  
266 Return), which allows to define derived operators and their behaviour, including custom  
267 operators (not existing in the standard library) for specific purposes.

269 This manual describes in detail the operators of VTL 1.1 and is organized as follows.

270 In this chapter, the following paragraph (Diagrams of the Operators) summarizes all the  
271 available operators (for the VTL-DL, VTL-ML - Core Operators, VTL-ML - Standard Library)  
272 through a diagram. Then, in the paragraph “List of the Operators/Functions”, two  
273 corresponding lists are given, specifying for each operator some basic information. In  
274 “Evaluation Order”, the precedence rules for evaluation of the VTL-ML operators are  
275 described. Finally, the “Syntactical conventions” section illustrates the meta-syntax used in  
276 the other chapters for describing formally the syntax of the operators.

277 The remainder of the document is structured in chapters, each one dedicated to the  
278 description of a category of Operators. For each Operator there is a specific section explaining  
279 the syntax, the semantics and giving some examples. Each of these sections has the following  
280 structure:

281

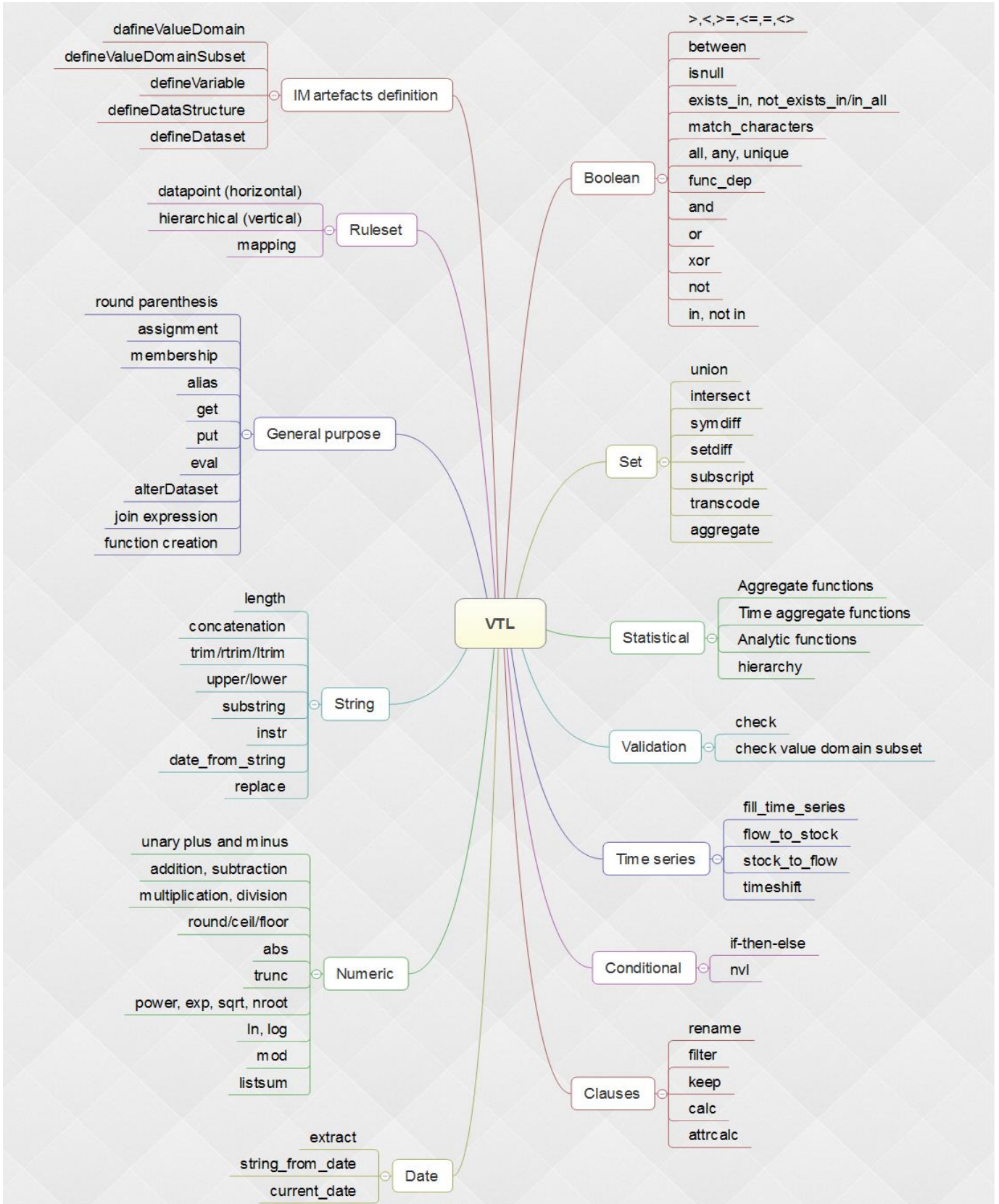
- 282 • **Semantics:** an informal extract of the behaviour;
- 283 • **Syntax:** a specification of the complete syntax of the operator at hand. It is expressed  
284 in terms of the types of the Core (cfr. part 1) by means of a specific meta-syntax;
- 285 • **Parameters:** the input parameters, described in detail, with respect to the types of the  
286 Core;
- 287 • **Returns:** the output parameters, described in detail, with respect to the types of the  
288 Core;
- 289 • **Constraints:** semantic constraints and syntactical constraints that cannot be specified  
290 with the meta-syntax but need a textual explanation; sometimes for the sake of clarity,  
291 even syntactical constraints are also repeated.
- 292 • **Semantic specification:** an extensive explanation of the behaviour of the operator in  
293 terms of the syntactical elements described in the sections Syntax, Parameters and  
294 Returns. Sometimes, when particularly complex, specific constraints are explained also  
295 in this section.
- 296 • **Examples:** a series of examples proving the behaviour of the operator.

297

298 The last chapter illustrates the use case of a real questionnaire and the possible use of VTL for  
299 defining validation rules.

300

# Diagram of the Operators



303

## List of the Operators/Functions

304 The following tables list the VTL Operators and describe their main characteristics. The tables are relevant to the  
 305 VTL-DL and the VTL-ML Standard Library. The operators of the Standard Library are ordered by category except  
 306 for the clauses, which are the operators having a postfix syntax that are shown all together in the end.

307 The VTL-ML Standard Library includes operators that may act on both Data Set and on Structure Components of  
 308 the Data Sets. The last column shows if the Operator acts on Dataset, Components or both, when meaningful. The  
 309 Component version takes as input and returns in output Component expressions. They are part of the syntax of  
 310 other operators or clauses, where specifically required for row-wise processing.

311

### VTL-DL Operators

Operator/Functions	Category	Syntax	Description	Operand Data Sets	Component version
defineValueDomain	Information Model artefacts definition	Functional	defines a ValueDomain in VTL information model.	-	-
defineValueDomainSubset	Information Model artefacts definition	Functional	defines a ValueDomainSubset in VTL information model	-	-
Define Variable	Information Model artefacts definition	Functional	defines a persistent Variable in the VTL information model	-	-
defineDataStructure	Information Model artefacts definition	Functional	defines a persistent DataStructure in the information model	-	-
defineDataset	Information Model artefacts definition	Functional	defines a persistent Dataset in the information model	-	-
define datapoint ruleset	Ruleset	Functional	defines a persistent object that contains Rules to be applied to the Data Points	-	-
define hierarchical ruleset	Ruleset	Functional	defines a persistent object that contains Rules to be applied to the code items of a Dataset component.	-	-
define mapping	Ruleset	Functional	defines a persistent object that contains Rules to be applied to recode codes of a component in a Dataset	-	-

312

## 314 VTL-ML Standard Library

315

## 316 Operators and functions applied on Datasets and scalar values

317

318 Most of the single data points operators and functions can be applied to both Datasets and scalar values. The  
 319 operands of the operators and functions can take the following forms:

320

- Scalar expression, e.g. 1+2.
- Dataset expression, with a single measure or attribute selected using the membership operator ".", e.g. ds\_bop.obs\_value. In this case the operator or function is applied to the specified measure or attribute.
- Dataset expression, with no measure or attribute selected, e.g. ds\_bop. In this case the operator or function is applied to all measures of the Dataset having the data type accepted by the operator.

326

327 When a VTL operator or function is applied to two or more Datasets then at least an Identifier Component must  
 328 appear in all Datasets with the same name and data type. In this case the function is applied on the measures  
 329 having the same name and data type (accepted by the operator) and for the matching data points, i.e., the data  
 330 points that have the same values of the common Identifier Components.

330

331 Assuming that  $f$  is a VTL function or operator,  $ds$ ,  $ds1$  and  $ds2$  are Datasets and  $c$  is a scalar value (constant), the  
 332 following table shows the VTL rules in the case of binary operators or functions:

332

Case	Result	Computation rule	Examples
$f(c, c)$	A scalar value	$f$ is applied to the scalar operands	1 + 1 round ( 10.52, 1) "abc"    "cde"
$f(ds, c)$	A Dataset having the same components (Identifiers and Attributes Components) of $ds$ . The Measure Components returned are only those having data type accepted by the operator. The other Measures will be discarded.	$f$ is applied to all measures of $ds$ having data type accepted by the operator. $f$ is applied to all data points of $ds$ . The cardinality of the resulting Dataset (number of data points) is the same of $ds$ .	$ds + 1$ round ( $ds$ , 1 )
$f(ds1, ds2)$	A Dataset having all the Identifier Components (without duplicates) and the common measures of $ds1$ and $ds2$ having data type accepted by the operator. The other Measures will be discarded.  The attributes of $ds1$ and $ds2$ are ignored (do not appear in the resulting Dataset).	$f$ is applied to the common numeric measures of $ds1$ and $ds2$ . $f$ is applied to all matching data points of $ds1$ and $ds2$ (those having the same values of the common Identifier Components) and to the Measures having data type accepted by the operator.  The cardinality of the resulting Dataset (number of data points) is the number of matching data points.	$ds1 + ds2$ mod ( $ds1$ , $ds2$ )
$f(ds.m, c)$	A Dataset having all the Identifier Components of $ds$ , the specified Measure Component $m$ and the Attribute Components of $ds$ .	$f$ is applied to the specified Measure Components of $ds$ . $f$ is applied to all data points of $ds$ .	round ( $ds$ .obs_value, 1) $ds$ .obs_comment    "."

		The cardinality of the resulting Dataset (number of data points) is the same of ds.	
$f(ds1.m1, ds2.m1)$	A Dataset having all the Identifier Components (without duplicates) of $ds1$ and $ds2$ , and the Measure Component $m1$ . The same Measure must be selected in both Datasets.  The Attributes of $ds1$ and $ds2$ , and the other Measures (if any), are ignored (do not appear in the resulting Dataset).	$f$ is applied to the specified Measure of ds, or to the common measures of ds.  $f$ is applied to all matching data points of $ds1$ and $ds2$ (those having the same values of the common Identifier Components).  The cardinality of the resulting Dataset (number of data points) is the number of matching data points.	$ds1 + ds2$  $\text{mod}(ds1.\text{obs\_value}, ds2.\text{obs\_value})$

333

334 To apply the function  $f$  to measures having different names (in different Datasets) is possible using the operator  
335 **as**, e.g.:

336  $ds1.\text{obs\_value} + (ds2.\text{obsval as obs\_value})$

337 A Dataset contains a set of data points. A data point (statistical observation) can be thought of as a row in a  
338 relational table or as a cell in a hypercube.

339 Scalars are also supported. As we will show, many operators allow for a kind of hybrid combination, involving  
340 Datasets and scalars. In this case the scalar value is combined (according to the semantics of the operator at  
341 hand) with all the Data Points in the Dataset, and in particular with the respective values of the Measure  
342 Component.

343 For example:

344  $ds2 := ds1 + 1$

345 produces a Dataset  $ds2$  with the same structure as  $ds1$ , where the constant numeric value 1 has been added to  
346 the value of the Measure Component of every single Data Point in  $ds1$ . Seen in another perspective, with this  
347 behavior, we propose a kind of implicit “promotion” of a scalar value into a somehow special Dataset, with one  
348 single Data Point, having one Measure Component (with the constant value) and with no Identifier Components.

349 In such a case, this single Data Point will match with all the Data Points of the involved Data Set as a limit but  
350 straightforward case, since, indeed, there are no Identifier Components to be matched at all.

351

### 352 [List of standard library operators and functions](#)

353

Operator/ Functions	Category	Syntax	Description	Operand Data Sets	Component version	Core/ Standard
Round parenthesis ()	General purpose	Functional	Specifies the evaluation precedence	1	YES	Standard
assignment :=	General purpose	Infix	Assigns an Expression to a model artefact	2	NO	Standard
Membership .	General purpose	Infix	Identifies a Component within a Data Set	1	NO	Standard
Alias as	General purpose	Infix	Define an alias for a component or for the result of an expression	1	YES (ONLY)	Standard

alterDataset	General purpose	Functional	Modify the Dataset with all or a subset of input components having only the Identifier role	1	NO	Standard
get	General Purpose	Functional	Retrieves a Data Set	1..N	NO	Standard
put	General Purpose	Functional	Stores a Data Set	1	NO	Standard
eval	General Purpose	Functional	Evaluates an external routine	1	NO	Standard
join expression	General Purpose	Functional	Implements the FLWOR expression	1..N	YES	Core
Function creation	General Purpose	Functional	Creates a function	1..N	YES	Core
null	General Purpose	Functional	null literal	0	YES	Core
length	String	Functional	Returns the length of a string	1	YES	Standard
concatenation 	String	Functional	Concatenates two strings	2	YES	both
trim/ltrim/rtrim	String	Functional	Eliminates trailing or/and leading whitespace from a String	1	YES	Standard
upper/lower	String	Functional	Makes a string upper / lower case	1	YES	Standard
substr	String	Functional	Extracts a substring from a string	1	YES	Standard
instr	String	Functional	Returns the position of a String in another one	1	YES	Standard

date_from_string	String	Functional	Change a string into a date	1	YES	Standard
replace	String	Functional	Replace a string with another one into a string	1	YES	Standard
unary plus +	Numeric	Infix	Leaves the sign unaltered	1	YES	both
unary minus -	Numeric	Infix	Changes the sign	1	YES	both
addition + and subtraction -	Numeric	Infix	Sum or subtract two numbers	2	YES	both
multiplication * and division /	Numeric	Infix	Multiply or divide two numbers	2	YES	both
round/ceil/floor	Numeric	Functional	Rounds a number	1	YES	Standard
abs	Numeric	Functional	Calculates the absolute value	1	YES	Standard
trunc	Numeric	Functional	Truncates the values	1	YES	Standard
exp	Numeric	Functional	Calculates the exponential	1	YES	both
ln	Numeric	Functional	Calculates the natural logarithm	1	YES	Standard
log	Numeric	Functional	Calculates the a base b logarithm	1	YES	Standard
power	Numeric	Functional	Calculates the power	1	YES	Standard
sqrt	Numeric	Functional	Calculates the square root of a number	1	YES	Standard
nroot	Numeric	Functional	Calculates the	1	YES	Standard



			n-th root			
mod	Numeric	Functional	Calculates the modulo	1	YES	both
listsum	Numeric	Functional	Sums numbers replacing null with zero	1..N	YES	Standard
equal to =	Boolean	Infix	Compares the values	2	YES	both
not equal to <>	Boolean	Infix	Compares the values	2	YES	both
Greater than >, >=	Boolean	Infix	Compares the values	2	YES	both
Less than <, <=	Boolean	Infix	Compares the values	2	YES	both
in, not in	Boolean	Infix	Verify if a value belongs to a set of values	1	YES	Standard
between	Boolean	Infix	Verify if a value belongs to a range of values	1	YES	both
isnull	Boolean	Functional	Compares the values with the NULL literal	1	YES	both
exists_in, not_exists_in, exists_in_all, not_exists_in_all	Boolean	Infix	Checks the Identifiers and the foreign keys	2	NO	Standard
match_characters	Boolean	Functional	Checks if a value respects or not a pattern	1	YES	Standard
all	Boolean	Functional	Verifies that all values in the Dataset are true	1	YES	Standard
any	Boolean	Functional	Verifies that at least one value in the	1	YES	Standard

			Dataset are true			
unique	Boolean	Functional	Verifies that at only one value in the Dataset are true	1	YES	Standard
func_dep	Boolean	Functional	Checks the functional dependency between components of a Dataset	1	YES	Standard
and	Boolean	Infix	Calculates the logical AND	2	YES	both
or	Boolean	Infix	Calculates the logical OR	2	YES	both
xor	Boolean	Infix	Calculates the logical XOR	2	YES	both
not	Boolean	Infix	Calculates the logical NOT	1	YES	both
extract	Date operator	Functional	Returns an integer that is part of a given date	1	YES	Standard
string_from_date	Date operator	Functional	Converts a date value into a string	1	YES	Standard
current_date	Date operator	Functional	returns the current date and time	0	YES	Standard
union	Set	Functional	Computes the union of datasets	1..N	NO	Standard
intersect	Set	Infix	Computes the intersection of datasets	1..N	NO	Standard
symdiff	Set	Functional	Computes the symmetric difference of 2 datasets	2	NO	Standard
setdiff	Set	Infix	Computes the difference of 2 datasets	2	NO	Standard

subscript	Set	Postfix	Assigns a fixed value to the identifiers and remove them	1	NO	Standard
transcode	Set	Functional	Recodes the identifiers values using a mapping ruleset	1	YES	Standard
aggregate	Set	Functional	Aggregates data using a hierarchical ruleset.	1	YES	Standard
aggregateFunctions	Statistical function	Functional	Set of statistical functions used to aggregate data	1	YES	Standard
time_aggregate	Statistical function	Functional	Set of statistical functions used to aggregate data using time constraints	1	YES	Standard
analytic function	Statistical function	Functional	Allows to specify operations on groups of Data Points	1	YES	Standard
hierarchy	Statistical function	Functional	Applies a hierarchical aggregation	1	YES	Standard
check (with datapoint ruleset)	Validation	Functional	Applies one or more datapoint Ruleset on a Dataset.	1	YES	Standard
check (with hierarchical ruleset)	Validation	Functional	Applies one or more hierarchical ruleset on a Dataset.	1	YES	Standard
check (single rule)	Validation	Functional	Checks if an expression verifies a condition	1	NO	Standard
check value domain subset	Validation	Functional	Checks if the Value Domain Subset is respected	1	NO	Standard
fill_time_series	Time series	Functional	Replaces each missing data point in the input Dataset	1	YES	Standard

flow_to_stock	Time series	Functional	Transforms from a flow interpretation of a Dataset to stock	1	YES	Standard
stock_to_flow	Time series	Functional	transforms from a stock interpretation of a Dataset to flow	1	YES	Standard
timeshift	Time series	Functional	Shifts the time component of a specified range of time	1	YES	Standard
if-then-else	Conditional	Functional	Makes different calculations according to a condition	1	YES	Standard
nvl	Conditional	Functional	Replaces the null value with a value	1	YES	Standard
rename	Clause	Clause (Postfix Operator)	change the name and the role of Measures or Attributes component	1	YES (ONLY)	Standard
filter	Clause	Clause (Postfix Operator)	Filters the Data Points	1	YES (ONLY)	Standard
keep	Clause	Clause (Postfix Operator)	Alters the Data Structure	1	YES (ONLY)	Standard
calc	Clause	Clause (Postfix Operator)	Calculates the values of a Structure Component	1	YES (ONLY)	Standard
attrcalc	Clause	Clause (Postfix Operator)	Calculates the values of an Attribute	1	YES (ONLY)	Standard

354

355

356 VTL-ML - Evaluation order of the Operators

357 Within a single expression of the manipulation language, the operators are applied in sequence, according to the  
 358 precedence order. Operators with the same precedence level are applied according to associativity rules.  
 359 Precedence and associativity orders are reported in the following table.  
 360

Order	Operator	Description	Associativity
I	()	Round parenthesis. To alter the default order.	Left-to-right
II	All VTL functional operators	The majority of the operators of the VTL	Left-to-right
III	Clauses and membership		Left-to-right
IV	unary plus unary minus not	Unary minus Unary plus Logical negation	Right-to-left
V	*, /	Multiplication Division	Left-to-right
VI	+, -	Addition Subtraction	Left-to-right
VII	> >= < <= in, not in between	Greater than Less than In (not in) a value list In a range	Left-to-right
VIII	exists_in not_exists_in exists_in_all not_exists_in_all	Identifiers matching	Left-to-right
IX	= <>	Equal-to Not-equal-to	Left-to-right
X	and	Logical AND	Left-to-right
XI	or xor	Logical OR Logical XOR	Left-to-right
XII	if-then-else	Conditional (if-then-else)	Right-to-left
XIII	:=	Assignment	Right-to-left

361

363 In the remainder of the document, and in the Syntax sections in particular, a meta-syntax is  
 364 used to describe the syntax of the operators. The meta-syntax is described in this section and  
 365 is **not part** of the VTL language, but has only presentation purposes.

- 366 • *For denoting the type of a Variable Parameter, we refer to the – **VTL types** (See User*  
 367 **Manual, Section “Objects and Types”**).
- 368 • *Operator names and parameters are **case sensitive**.*
- 369 • *In general, some operators have infix style, others have functional style and the*  
 370 *clauses have postfix style.*

371 The syntax of the operators is defined by *meta-expressions*, which denotes the signature of an  
 372 operator, that is, its **name**, the list of **the input parameters**, the possible special **keywords**  
 373 and the respective **types**. For readability reasons, a meta-expression is often partitioned into  
 374 concatenated sub-meta-expressions (or simply sub-expressions), as follows:

```
375     meta-expression ::= sub-expr1 sub-expr2 ... sub-exprN
376                   sub-expr1 ::= sub-meta-expression ...
377                   ...
378                   sub-exprN ::= sub-meta-expression ...
```

379 In this representation:

- 380 • The *sub-expr1, ... sub-exprN* are meta-variables, that is, placeholders for sub-  
 381 expressions. In the text, they are in italic.
- 382 • The symbol ::= means “defined as” and denotes the assignment of a sub-expression to  
 383 a meta-variable.
- 384 • The operator names and the special keywords that appear in the various sub-  
 385 expressions are in **bold**.

386 Sub-expressions can be composed into the meta-expression adopting a particular restriction  
 387 of **regular expression patterns** as follows:

- 388 • *{optional}, {optional}?,[optional]? : alternative ways to denote an optional sub-*  
 389 *expression*
- 390 • *{one-or-more}+: a sub-expression that is repeated from 1 to many occurrences*
- 391 • *{zero-or-more}\*: a sub-expression that is repeated from 0 to many occurrences*
- 392 • *[part1|part2|part3]: alternative sub-expressions*
- 393 • *[part1|part2|part3]+: alternative sub-expressions, from 1 to many occurrences*
- 394 • *[part1|part2|part3]\*: alternative sub-expressions, from 0 to many occurrences*

### 395 Example

```
396 [trim | ltrim | rtrim ] ( ds )
```

```
397
398 ds : dataset {identifier <IDENT> as scalar-type}+
399         {measure <IDENT> as string-literal}+
400         {attribute <IDENT> as scalar-type}*
401
```

402 The meta-expression above synthesizes:

- 403
- **trim, ltrim, rtrim**, “(”, “)” are the operator names (reserved keywords);
  - They take *s* input an expression *ds*, which is a meta-sub-expression and defined accordingly;
  - the type of *ds* is constrained to be a *Dataset* with one or more Identifier Components and one or more string Measure Component. No particular constraints are introduced for attributes.
  - *ds* is the only parameter of the operators in the example and denotes a Dataset. Specifically, <IDENT> is a placeholder for any identifier (measure or attribute, in the different cases).
- 404
- 405
- 406
- 407
- 408
- 409
- 410
- 411

412

413

414 From this template, it is possible to infer some valid instances of the operators:

415

416 `ds_1 := ltrim(ds_2)`

417 `ds_1 := rtrim(ds_3)`

418

419 The two examples above are compliant with the template. In facts, **ltrim** and **rtrim** are  
420 recognized as VTL operators of the library and *ds\_2* and *ds\_3* are two Datasets. Also observe  
421 that the example implies a previous definition of *ds\_2* and *ds\_3*, for example importing the  
422 Datasets from the database (as we will see, with the GET operator). The restrictions on the  
423 specific structure of the input Datasets, in terms of allowed Identifier and Measure  
424 Components, are also checked, but do not have effects on the syntax.

425

426

427

428

429

431 

## defineValueDomain

432 

### Semantics

433 The operator **defineValueDomain** defines a ValueDomain in the VTL information model.

434

435 

### Syntax

```
436 defineValueDomain valueDomainId (
437     {valueDomainDescription, isEnumerated}
438     dimensionType { [ inLineCodeList | dataTypeRestriction] }
439 )
```

440

441 

### Parameters

442 *valueDomainId* : ident443 *valueDomainDescription* : string444 *isEnumerated* : boolean445 *dimensionType* : scalar446 *inLineCodeList* : **list**( { **record**({@codeItemId as ident; { #codeDescr as constant; }})\* )447 *codeItemId* : ident448 *codeDescr* : constant449 *dataTypeRestriction*

```
450     : restrict [YYYY | MM | DD | YYYY-MM | maxLength n | regexp regexp | between a and b | > b | < n |
451     <= n | >= n]
```

452 *n, a, b* : numeric453 *regexp* : string

454

- 455 • *valueDomainId* – is the identifier of the new ValueDomain.
- 456 • *valueDomainDescription* – is a string that describes the new ValueDomain.
- 457 • *isEnumerated* – is a Boolean that denotes whether the new ValueDomain is enumerated.
- 458 • *dimensionType* – is the data type of the Identifier Component.
- 459 • *inLineCodeList* – is an in line specification of a CodeList. It is a list of records (pairs, in particular). The first element of the record is the *codeItemId* (which identifies the code item, is the identifier of the record "@"), the second, optional, is the codeDescription, that is, the actual value for the code item. An *in-Line CodeList* cannot be reused.
- 463 • *regexp* – is a regular expression.
- 464 • *dataTypeRestriction* – constrains the allowed values by restricting *dimensionType*.

465

466 

### Constraints

467 • The scalar-type of the constant codeDescr must be dimensionType.

468 • *regexp* is a POSIX regular expression.469 • If the ValueDomain is enumerated, an *inLineCodeList* must be specified.470 • The particular restriction for *dataTypeRestriction* must be coherent with *dimensionDataType*. In particular:

471 ○ date: [YYYY | MM | DD | YYYY-MM | &gt; YYYY-MM-DD | &lt; YYYY-MM-DD | &gt;= ...]

472 ○ string: maxLength n, regexp regexp

473 ○ number: [between a and b | &lt;a | &gt;a | ...]

474

475

476 

### Returns

477 This operator defines persistent ValueDomain artefacts that can be referenced by a reference to *valueDomainId*.478 References valueDomainRef to *valueDomainId* are implicitly created in the VTL information model.

479

480 

### Semantic specification

481 This operator takes as input an identifier for this ValueDomain and the specification for its dimension. The dimension is in turn a component ValueDomain.

482 Only mono-dimensional ValueDomains can be defined, the multi-dimensional ValueDomain are implicitly defined in the VTL information model as the Cartesian product of the mono-dimensional ones. The allowed

484



485 values are directly those specified by the criteria. Otherwise, in case of n-dimensional ValueDomains, all the  
 486 combinations of values of the mono-dimensional ValueDomains are possible, which means that the ValueDomain  
 487 contains the Cartesian product of the values of the single mono-dimensional ValueDomain.  
 488 The definition of a ValueDomain comports also the implicitly definition of the respective full  
 489 ValueDomainSubset, that is the subset of the ValueDomain that allows the same values of the ValueDomain,  
 490 without further restrictions to its domain.  
 491 The dimension defines a set of allowed values by means of one among different criteria: 1) with an in-line  
 492 definition of a codeList; 2) by restricting the *dimensionType* to a subset of allowed values by means of a criterion  
 493 out of a set of pre-defined ones; 3) by allowing all the values of the specified *dimensionType*.  
 494 Notice that a CodeList can only be defined within a *ValueDomain* or a *ValueDomainSubset*, using the in-line mode.  
 495 After the application of the operator, the information model is modified as follows.  
 496 A *ValueDomain* identified by *valueDomainId* is created. Its description *ValueDomainDescr* is set to the value of  
 497 *valueDomainDescription*, when specified, NULL otherwise. Property *isEnumerated* is set according to parameter  
 498 *isEnumerated*. Property *DataType* is set to *dimensionType*.  
 499 Anytime a ValueDomain is specified, are implicitly specified with him all the possible combination between the  
 500 new ValueDomain and the others. Therefore, the definition of a ValueDomain defined also multi-dimensional  
 501 ValueDomain. In addition, a ValueDomainSubset full is defined and its ValueDomainRef is set to *valueDomainId*.  
 502 For the dimension, a corresponding **component** ValueDomain identified by the respective *compValueDomainId*  
 503 is created in the information model and its properties are set as follows. The property *valueDomainDescription*  
 504 contains the string "component ValueDomainSubset of *id*", where *id* amounts to *valueDomainId*. Property  
 505 *DataType* is set to *dimensionType*.

#### 506 *Examples*

507 1) This example defines the ValueDomain TimeYears as a restriction of the date type where only the digits  
 508 representing the years are considered.

```
509 define ValueDomain TimeYears ("Time values", date restrict YYYY)
```

510

511 2) This example defines the ValueDomain GeoAreas with an in-line CodeList, that is the enumeration of all the  
 512 allowed values.

```
513 define ValueDomain GeoAreas("Geographic areas", string list(record(@IT, "Italy"), record(@LU,  
514 "Luxembourg"),",", "...))
```

515  
516

## 517 **defineValueDomainSubset**

#### 518 *Semantics*

519 The operator **defineValueDomainSubset** defines a ValueDomainSubset in the VTL information model.

520

#### 521 *Syntax*

```
522 defineValueDomainSubset valueDomainSubsetId (  
523     {valueDomainSubsetDescription, isEnumerated}  
524     valueDomainRef { [ inLineSubCodeList | dataTypeRestriction ] }  
525 )
```

526

#### 527 *Parameters*

528 *valueDomainSubsetId* : ident

529 *valueDomainSubsetDescription* : string

530 *isEnumerated* : Boolean

531 *valueDomainRef* : valueDomain-ref

532 *inLineSubCodeList*: list{ { **record**{{@codeItemId as ident; { #codeDescr as constant; } } }\* }

533 *dataTypeRestriction*

534 : **restrict** [YYYY | MM | DD | YYYY-MM | **maxLength** n | **regexp** regexp | **between** a **and** b | > b | < n | <= n |  
535 >= n]

536 *n*, *a*, *b* : numeric

537 *regexp* : string

538

- 539 • *valueDomainSubsetId* – is the identifier of the new ValueDomainSubset.

- 540 • *valueDomainSubsetDescription* – is a string that describes the new ValueDomainSubset.

- 541 • *isEnumerated* – specifies whether the ValueDomainSubset is enumerated.
- 542 • *valueDomainRef* – is the reference to an existing ValueDomain.
- 543 • *inLineCodeList* – is an in line specification of a CodeList. It is a list of records (pairs, in particular). The first
- 544 *element* of the record is the codeItemId (which identifies the code item, is the identifier of the record "@"),
- 545 the second, optional, is the codeDescription, that is, the actual value for the code item. An in-line CodeList
- 546 cannot be reused.
- 547 • *regexp* – is a POSIX regular expression.
- 548 • *dataTypeRestriction* – constrains the allowed values by restricting dimensionType of the referred
- 549 ValueDomain.

550

### 551 *Constraints*

- 552 • *regexp* is a POSIX regular expression.
- 553 • The possible restrictions on the values of the dimension must be coherent with the type of the dimension in
- 554 the *ValueDomain* referred to by *valueDomainRef*.
- 555 • The criteria according to which the values of the dimension is defined must be the same as in the referred
- 556 *ValueDomain*, that is: 1) if an *in-Line CodeList* is used in the *ValueDomain*, then in the *ValueDomainSubset* an
- 557 *in-Line CodeList* containing a subset of the values must be used; 2) if a *dataTypeRestriction* has been used in
- 558 the *ValueDomain*, then a *dataTypeRestriction* must be used in the *ValueDomainSubset*.
- 559 • If the ValueDomainSubset is enumerated, an inLineSubCodeList must be specified.
- 560 • Independently of the way in which the values of the dimension are defined, the allowed values for the
- 561 dimension of the *ValueDomainSubset* must be a subset of the allowed values in the referred *ValueDomain* for
- 562 the respective dimension.
- 563 • The particular restriction for *dataTypeRestriction* must be coherent with *dimensionDataType* of the referred
- 564 *ValueDomain*. In particular:
  - 565 ○ date: [YYYY | MM | DD | YYYY-MM | > YYYY-MM-DD | < YYYY-MM-DD | >= ...]
  - 566 ○ string: maxLength n, regexp
  - 567 ○ number: [between a and b | <a | >a | ...]

568

### 569 *Returns*

570 This operator defines persistent *ValueDomainSubset* artefacts that can be referenced by a reference to

571 *valueDomainSubsetId*. References *valueDomainRef* to *valueDomainId* are implicitly created in the VTL

572 information model.

573

### 574 *Semantic specification*

575 This operator takes as input an identifier for this ValueDomainSubset, a reference to an existing ValueDomain

576 and the specification for its dimension in terms of subsets of the dimension of the referred ValueDomain.

577 If no further constraints are posed, all the values that are allowed in the dimension of the ValueDomain are

578 allowed in the ValueDomainSubset as well; alternatively, restrictions on the dimension can be specified

579 according to a set of criteria.

580 Only mono-dimensional ValueDomainSubsets can be defined, the multi-dimensional ValueDomainSubset are

581 implicitly defined in the VTL information model as the Cartesian product of the mono-dimensional ones. The

582 allowed values are directly those specified by the criteria. Otherwise, in case of n-dimensional

583 ValueDomainSubsets, all the combinations of values of the mono-dimensional ValueDomainSubsets are possible,

584 which means that the ValueDomainSubset contains the Cartesian product of the values of the single mono-

585 dimensional ValueDomainSubset.

586 The general rule is that the restrictions for the dimension must produce a subset of the values that are present in

587 the ValueDomain for that dimension.

588 The allowed criteria are the following: 1) with an in-line definition of a sub CodeList; 2) by restricting the

589 dimensionType to a subset of allowed values by means of a criterion out of a set of pre-defined ones; 3) by

590 allowing all the values that are allowed in the referred ValueDomain.

591 If in the ValueDomain no restriction is applied, in the ValueDomainSubset any restriction that is coherent with

592 the type of the respective dimension can be applied (hence no restriction, CodeList specification, data type

593 restriction). If in the ValueDomain a CodeList (defining it in an in-line fashion) is specified, the

594 ValueDomainSubset can either inherit all the values (no restriction) or restrict such CodeList specifying an in-

595 line subset CodeList. If in the ValueDomain a *dataTypeRestriction* is adopted, the ValueDomainSubset can either

596 inherit all the values (no restriction) or use another *dataTypeRestriction* that produces a subset of the parent one

597 when applied to the original *dimensionType*.

598 After the application of the operator, the information model is modified as follows.

599 A ValueDomainSubset, identified by ValueDomainSubsetId, is created (the value of the property SetId is set to

600 ValueDomainSubsetId). Its description SetDescr is set to the value of *valueDomainSubsetDescription*, if present,

601 NULL otherwise. Property `isEnumerated` is set according to parameter `isEnumerated` and coherently with the  
602 Property `SetCriterion`. Property `Criterion` is set to “IN\_LINE\_CODELIST” (in-line CodeList), “RESTRICTION” (type  
603 restriction), “FULL” (all the values of the referenced dimension) depending on how the allowed values have been  
604 specified.

605 For the dimension, a corresponding **component** `ValueDomainSubset` is created and its properties are set as  
606 follows.

607 The identifier `SetId` is set to `ValueDomainSubsetId` concatenated to the string “\_REF\_” concatenated to the  
608 `compValueDomainId` of the referred component `ValueDomain`. The property `SetDescr` contains the string  
609 “component `ValueDomainSubset` of `id`”, where `id` amounts to `compValueDomainId` of the referred component  
610 `ValueDomain`. Property `SetCriterion` is set to “IN\_LINE\_CODELIST” (in-line CodeList), “RESTRICTION” (type  
611 restriction), “FULL” (all the values of the referenced dimension) depending on how the allowed values have been  
612 specified.

613 Note that unlike in the `ValueDomains`, the identifiers for the component `ValueDomainSubsets` are statically  
614 specified and cannot be overridden. They are unique for a given `ValueDomain` and `ValueDomainSubsets`, so that  
615 a `ValueDomain` can be restricted in different ways. Moreover, note that there is no need for an artefact  
616 memorizing the relationship between the component `ValueDomainSubsets` and the compound ones, since it can  
617 be directly inferred from the identifiers conventions.

618 This operator also allows to alter existing `ValueDomainSubset` in a basic way. If a `ValueDomainSubset` with the  
619 same `valueDomainSubsetId` already exists in the information model, it is replaced by the newly defined one. The  
620 same holds for the respective component.

621

### 622 *Examples*

623 1) This example defines a `ValueDomainSubset` of positive numbers as a restriction of a `ValueDomain` allowing  
624 any integer number.

```
625 define ValueDomain Numbers(“Integer Numbers”, integer);
```

```
626 define ValueDomainSubset PositiveNumbers ( “Number greater than 0”, Numbers, restrict > 0)
```

627

628 2) This example defines a `ValueDomainSubset` for email addresses, as a restriction of a `ValueDomain` allowing  
629 any string.

```
630 define ValueDomain EmailAddresses(“E-mail addresses”, string-literal);
```

```
631 define ValueDomainSubset validEmailAddress(“Valid e-mail addresses”, EmailAddresses, restrict “[a-z]+@[a-  
632 z].[a-z]+”
```

633

## 634 **defineVariable**

### 635 *Semantics*

636 The operator **defineVariable** defines a persistent `Variable` in the VTL information model.

637

### 638 *Syntax*

```
639 defineVariable variableId
```

640

### 641 *Parameters*

```
642 variableId : ident
```

## 643 **defineDataStructure**

### 644 *Semantics*

645 The operator **defineDataStructure** defines a persistent `DataStructure` in the VTL information model

646

### 647 *Syntax*

```
648 defineDataStructure dataStructureId (
```

```
649     {dataStructureDescr}
```

```
650     { [ componentType ( componentName [Identifier | Measure | Attribute] ) |
```

```
651     valueDomainSubsetRef ( componentName [Identifier | Measure | Attribute] ) ] ;
```

```
652 )
```

653

654

655 *Parameters*  
656 *dataStructureId* : ident  
657 *dataStructureDescr* : string  
658 *componentType* : scalar  
659 *componentName* : ident  
660 *valueDomainSubsetRef* : valueDomainSubset-ref

- 661 • *dataStructureId* – is the identifier of the new *DataStructure*.
- 662 • *dataStructureDescr* – is a string that describe the new *DataStructure*.
- 663 • *componentType* – is the type of a Component in the new *DataStructure*.
- 664 • *componentName* – is a string that represents the name of the Component in the new *DataStructure*.
- 665 • *valueDomainSubsetRef* – is a reference to an existing *ValueDomainSubset*, used to assign a specific type to a
- 666 Component.
- 667

#### 668 *Constraints*

- 669 • At least one IdentifierComponent must be defined.
- 670 • At least one MeasureComponent must be defined.
- 671 • There cannot be two components with the same *componentName*.
- 672

#### 673 *Returns*

674 This operator defines a persistent *DataStructure* artefact that can be referenced by a reference to  
675 *dataStructureId*, in the VTL information model.

#### 676 *Semantic specification*

677 This operator defines a persistent *DataStructure* in the information model, allowing to specify its name and the  
678 description, along with the characteristics of its Components. It takes as input the identifier for this  
679 *DataStructure*, according to the conventions for it, optionally a description, and the specification for one or more  
680 Components. The Components can be defined in two ways: in a simplified form where there is a  
681 *componentName*, and a scalar-type for it is directly specified (*componentType*); in a fully-fledged form, where  
682 there is a *componentName*, and a *ValueDomainSubset* (mono-dimensional) is specified to restrict the allowed  
683 values.

684 Although in the VTL information model, a Component is always characterized by a *ValueDomainSubset*, the  
685 simplified form is particularly useful, since it prevents the need to define a *ValueDomain* and a  
686 *ValueDomainSubset* that are the mere renaming of a scalar data type. Let us now consider the fully-fledged form.

687 The *ValueDomainSubset* is mono-dimensional, it restricts the allowed values for a single Component;  
688 For each Component, a role must be declared by using one keyword among **Identifier**, **Measure** and **Attribute**.

689 After the application of the operator, the information model is modified as follows.

690 A *DataStructure* identified by *dataStructureId* is created. Its description *DataStructureDescr* is set to the value of  
691 *dataStructureDescr*, when specified, NULL otherwise.

692 For each Component, a *DataStructureComponent* is created. Its identifier, *componentId*, takes its value from the  
693 parameter *componentName*, which is unique within a single *DataStructure*. The *DataStructureComponent* is  
694 linked to the referred *DataStructure* by assigning the *DataStructureId* property.

695 If the Component is specified in the simplified form (only the data type), the created *DataStructureComponent* is  
696 linked (by the property *SetId*) to a conventional *ValueDomainSubset* for that type. Notice that a conventional  
697 *ValueDomainSubset* that simply renames each scalar type and the corresponding *ValueDomain* are assumed to  
698 be present in the information model, or created when needed and then reused.

699 If the *DataStructureComponent* is specified in the fully-fledged form (with its *ValueDomainSubset*), the single  
700 *ValueDomainSubset* is referred to by the property *SetId*.

701 In all the cases the property *VariableRole* is set to “Identifier”, “Measure” and “Attribute” depending on the used  
702 keyword.

703 For each component a new *RepresentedVariable* is created (or an existing one is reused). Its identifier,  
704 *VariableId*, is automatically and the respective property of *DataStructureComponent* is assigned accordingly. The  
705 description of the variable is automatically generated as “*RepresentedVariable* for <*componentId*>”. The  
706 *RepresentedVariable* is linked to the *ValueDomain* it takes its values from (being restricted by a specific  
707 *ValueDomainSubset* when assigned to a *DataStructureComponent*).

#### 708 *Examples*

709 1) Definition of a *DataStructure* where scalar types are used.

```
710 define DataStructure dstr_1(  
711     string ID identifier;
```

```

715         string NAME Identifier;
716         integer AGE Measure;
717     )
718 2) The example below allows to define a data structure using a ValueDomainSubset:
719 define ValueDomain Numbers("Integer Numbers", integer);
720 define ValueDomainSubset PositiveNumbers ("Number greater than 0", Numbers, restrict > 0)
721
722 define DataStructure dstr_1(
723     string ID Identifier;
724     string NAME Measure;
725     PositiveNumbers AGE Measure;
726 )
727

```

## 728 defineDataset

### 729 *Semantics*

730 The operator **defineDataset** defines a persistent Dataset in the VTL information model.

731

### 732 *Syntax*

```

733 defineDataset datasetId {
734     {datasetDescr,} {IsCollected},
735     [ dataStructureRef |
736         { [ componentType ( componentName [Identifier | Measure | Attribute] ) |
737             valueDomainSubsetRef ( componentName [Identifier | Measure | Attribute] ) ] };
738     ]
739 }

```

740

### 741 *Parameters*

```

742 datasetId : ident
743 datasetDescr : string
744 dataStructureRef: dataStructure-ref
745 componentType : scalar-type
746 componentName : string
747 valueDomainSubsetRef: valueDomainSubset-ref

```

748

- 749 • *datasetId* - is the identifier of the new Dataset.
- 750 • *datasetDescr* - is a string that describes the new Dataset.
- 751 • *isCollected* - if present this Dataset is an elementary one, otherwise it is meant to be the result of a calculation.
- 752 • *dataStructureRef* - is a reference to an existing DataStructure, used to assign a specific structure to the new Dataset. Optionally the DataStructure for the new Dataset can be defined in-line.
- 753 • *componentType* - is the type of a Component in the new Dataset.
- 754 • *componentName* - is a string that represents the name of the Component in the new Dataset.
- 755 • *valueDomainSubsetRef* - is a reference to an existing ValueDomainSubset, used to assign a specific type to a Component.

756

### 760 *Constraints*

- 761 • At least one Identifier Component must be defined.
- 762 • At least one Measure Component must be defined.
- 763 • There cannot be two components with the same *componentName*.

764

### 765 *Returns*

766 This operator defines a persistent Dataset artefact that can be referenced by a reference to *datasetId*, in the VTL information model.

767

### 768 *Semantic specification*

769



770 This operator defines a persistent Dataset in the information model, allowing to specify its name and the  
771 description, along with the characteristics of its Components (either specifying an existing DataStructure or  
772 defining the Components in an in-line fashion mode).  
773 It takes as input the identifier for this Dataset, according to the conventions for it, optionally a description, and  
774 the reference to an existent DataStructure or alternatively the specification for one or more Components. The  
775 Components can be defined in two ways: in a simplified form where there is a *componentName*, and a scalar-type  
776 for it is directly specified (*componentType*); in a fully-fledged form, where there is a *componentName*, and a  
777 ValueDomainSubset is specified to restrict the allowed values.  
778 Although in the VTL information model, a Component is always characterized by a ValueDomainSubset, the  
779 simplified form is particularly useful, since it prevents the need to define a ValueDomain and a  
780 ValueDomainSubset that are the mere renaming of a scalar data type. Let us now consider the fully-fledged form.  
781 In case of in-line definition of the Components: for each Component, a role must be declared by using one  
782 keyword among **Identifier**, **Measure** and **Attribute**.  
783 After the application of the operator, the information model is modified as follows.  
784 In case of reference to an existing DataStructure.  
785 A Dataset identified by *datasetId* is created. Its description *DatasetDescr* is set to the value of *datasetDescr*, when  
786 specified, NULL otherwise. The Dataset is linked to the DataStructure referred using *dataStructureRef* by  
787 assigning the *DataStructureId* property of the DataStructure identifier to the *DataStructureId* property of the  
788 new Dataset.  
789 In case of definition of a new DataStructure (not reusable).  
790 A Dataset identified by *datasetId* is created. Its description *DatasetDescr* is set to the value of *datasetDescr*, when  
791 specified, NULL otherwise. A DataStructure identified by an auto-generated *DataStructureId* is created for the  
792 new Dataset) and linked to it by assigning the generated identifier to the *DataStructureId* property of the  
793 Dataset. The description of the DataStructure is also generated automatically and set to  
794 "dataStructure\_of\_datasetId\_description" (if the DataStructure is reused, this convention for the description will  
795 be violated).  
796 For each Component, a DataStructureComponent is created (or the ones in the existing DataStructure are  
797 reused). Its identifier, *componentId*, takes its value from the parameter *componentName* (notice that for a  
798 DataStructure to be reused, these identifiers must be coherent), which is unique within a single DataStructure.  
799 The DataStructureComponent is linked to the referred DataStructure by assigning the *DataStructureId* property.  
800 If the Component is specified in the simplified form (only the data type), the created DataStructureComponent is  
801 linked (by the property *setId*) to a conventional ValueDomainSubset for that type. Notice that a conventional  
802 ValueDomainSubset that simply renames each scalar type and the corresponding ValueDomain are assumed to  
803 be present in the information model, or created when needed and then reused.  
804 If the StructureComponent is specified in the fully-fledged form (with its ValueDomainSubset, the single  
805 ValueDomainSubset is referred to by the property *setId*. The property *VariableRole* is set to "Identifier",  
806 "Measure" and "Attribute" depending on the used keyword.  
807 For each component a new RepresentedVariable is created (or an existing one is reused). Its identifier,  
808 *VariableId*, is automatically and the respective property of DataStructureComponent is assigned accordingly. The  
809 description of the variable is automatically generated as "RepresentedVariable for <componenId>". The  
810 RepresentedVariable is linked to the ValueDomain it takes its values from (being restricted by a specific  
811 ValueDomainSubset when assigned to a StructureComponent).

812

### 813 *Examples*

814 1) Definition of a Dataset, using an existing DataStructure.

```
815 define Dataset d_1("Dataset with the same structure of dstr_1", dstr_1)
816 dstr_1 is a DataStructure previously defined.
```

817

818 2) Definition of a Dataset with an in-line DataStructure definition where scalar types are used.

```
819 define Dataset d_1(
820     string ID identifier;
821     string NAME Identifier;
822     integer AGE Measure;
823 )
824
```

827 **define datapoint ruleset**828 *Semantics*

829 **define datapoint ruleset** defines a persistent object that contains Rules to be applied to each individual Data  
 830 Point of a given Dataset. These rulesets are also called “horizontal” taking into account the tabular  
 831 representation of a Dataset (considered as a mathematical function), in which each (vertical) column represents  
 832 a Variable and each (horizontal) row represents a Data Point: these rulesets are applied on individual Data  
 833 Points (rows), i.e. horizontally on the tabular representation.

834

835

836 *Syntax*

837 **define datapoint ruleset** *rulesetId* ( *RulesetSignature* ) **is**

838     { *Rule* } { ; *Rule* }\*

839 **end datapoint ruleset**

840 *Rule*

841     ::={ *ruleId*: } { **when** *antecedentCondition* **then** } *consequentCondition*

842     { **errorCode** ( *errorCode* ) }

843     { **errorlevel** ( *errorLevel* ) }

844 *RulesetSignature*

845     ::= *variable-signature* { , *variable-signature* }\*

846 *variable-signature*

847     ::= *variable-ref* { **as** *constant-string* }?

848

849 *Parameters*

850 *rulesetId* : identifier

851 *ruleId* : identifier

852 *antecedentCondition* : Boolean-scalar-expression

853 *consequentCondition* : Boolean-scalar-expression

854 *errorCode* : string

855 *errorLevel*: integer-literal

856 *constant-string*: string

857

- 858 • *rulesetId* – the identifier of the datapoint ruleset to be defined.
- 859 • *rulesetSignature* – the signature of the Ruleset. It specifies the Represented Variables (see the information  
860 model) on which the Ruleset is defined.
- 861 • *variable-signature* – it specifies a single Represented Variable on which the Ruleset is defined
- 862 • *variable-ref* - the reference to a Variable on which the Ruleset is defined. The Variable name can be aliased  
863 for the sake of compactness in writing the Rules. If the alias is not specified, the complete name of the  
864 Variable must be used in the body of the rules.
- 865 • *rule* – the complete specification of a single rule, as defined in the following parameters.
- 866 • *ruleId* – the identifier of the specific rule within the Ruleset. The *ruleId* is optional and, if not specified, is  
867 assumed to be the progressive order number of the Rule in the Ruleset (please note that this practice may  
868 cause changes of the rule identifiers in case the Ruleset is maintained, e.g. if new rules are added or existing  
869 rules are deleted)
- 870 • *antecedentCondition* - a Boolean scalar expression. It can contain references to the Variables declared for  
871 this Ruleset and Constants. All the Component level operators are allowed.
- 872 • *consequentCondition* - a Boolean scalar expression. It is evaluated when the *antecedentCondition* evaluates  
873 to true (missing antecedent conditions are assumed as true). It can contain references to the Variables  
874 declared for this Ruleset and Constants. All the Component level operators are allowed.
- 875 • *errorCode* – a string denoting the error code associated to the rule, respecting VTL conventions, in case the  
876 rule is used for validation.
- 877 • *errorLevel* - an integer containing the error level (severity) associated to the rule, in case the rule is used for  
878 validation.

879 • *constant-string*: the name assigned to the Variable within the ruleset

880

### 881 *Constraints*

882 • *antecedentCondition* and *consequentCondition* cannot use Variables that are not defined in the  
883 *RulesetSignature*

884 • A *Variable* can appear only once in the *RulesetSignature*

885 • Either the *ruleId* is specified for all the rules of the Ruleset or for none.

886 • If specified, the *ruleId* must be unique within the Ruleset.

887

### 888 *Returns*

889 A persistent DataPoint Ruleset identified by *rulesetId*, which can be referenced and used both for validation and  
890 data filtering (within a filter clause) purposes.

891

### 892 *Semantic specification*

893 A DataPoint Ruleset (also “horizontal ruleset”) is a persistent object that contains Rules to be applied to the Data  
894 Points of a given Dataset<sup>1</sup>. When used for validation, the Rules are aimed at checking the combinations of values  
895 of the Data Point Variables, assessing if these values fulfill the conditions expressed by the Rules themselves. The  
896 Rules are evaluated independently for each data point, returning a Boolean scalar value (see the **check** operator  
897 and the relevant options). When used for data filtering, the Rules are aimed at filtering the Data Points,  
898 maintaining only the ones that fulfill (or, as an option, that do not fulfill) the Rules themselves (see the **filter**  
899 operator and the relevant options).

900 Each rule contains an *antecedentCondition* Boolean expression followed by a *consequentCondition* Boolean  
901 expression and expresses a logical implication. Each condition states that when the *antecedentCondition*  
902 evaluates to **true**, for a given Data Point, then the *consequentCondition* must evaluate to **true** as well. In case the  
903 *antecedentCondition* is absent then it is assumed to be always true, therefore the *consequentCondition* must  
904 evaluate to true for all the Data Points. See the example below:

905

<i>Rule</i>	<i>Meaning</i>
<b>when</b> flow = "CREDIT" or "DEBIT" <b>then</b> obs_value >= 0	When the Variable named “flow” takes the value “CREDIT” or the value “DEBIT”, then the Variable named “obs_value” has a zero or positive value.
<b>when</b> flow = "BALANCE" <b>then</b> obs_value between -1.000.000 and +1.000.000	When the Variable named “flow” takes the value “BALANCE, then the Variable named “obs_value” has a value between -1.000.000 and +1.000.000

906

907 The definition of a Ruleset comprises a **signature** (*RulesetSignature*), which specifies the Represented Variables  
908 on which the Ruleset is defined and a number of **rules**, that are the Boolean expressions to be applied for each  
909 Data Point. The Rules can refer only to the Variables of the Ruleset signature, and must refer to all of them (in  
910 either the *antecedentCondition* or the *consequentCondition*, or both).

911 In regard to the Information Model, the Variables of the Ruleset signature identify a multi-dimensional space (i.e.  
912 a multi-dimensional Represented Variable), while each Rule provides for a criterion that demarcates a Set of  
913 values belonging to this space (i.e. a Set of combinations of values of these Variables).

914 A Ruleset can be applied on any Dataset which includes, among its Structure Components, the Variables of the  
915 Ruleset signature. More Rulesets having different signatures may be applied on the same Dataset, provided that  
916 the previous condition is satisfied.

---

<sup>1</sup> In order to apply the Ruleset to more Datasets, these Datasets must be joined together using the appropriate VTL operators in order to obtain a single Dataset.



917 Rules are uniquely identified by a *ruleId*. When the Ruleset is used for validation, two new Variables (the  
 918 RULESET and the RULE Variables) are added in the Dataset that contains the validation result and filled with  
 919 *rulesetId* and *ruleId* respectively, in order to document to which rules the results are referred. If not explicitly  
 920 declared, the *ruleId* is assumed by default to be the progressive order number of the Rule in the Ruleset (please  
 921 note that using the default mechanism the Rules identifiers can change if the Ruleset is maintained, e.g. if new  
 922 Rules are added or existing Rules are deleted, and therefore the users that interpret the validation results must  
 923 be aware of these changes).

924 As said, every **rule** is applied in a row-wise fashion to each individual Data Point of a Dataset. The references to  
 925 the Variables defined in the *antecedentCondition* and *consequentCondition* are replaced with the values of the  
 926 respective Variables of the Data Point under evaluation.

927 The semantics of each rule is the typical logical implication:

928 *antecedentCondition* and *consequentCondition*

929 The rule evaluates to true if: *antecedentCondition* evaluates to FALSE or *consequentCondition* evaluates to TRUE.  
 930 In practice, the *consequentCondition* must be evaluated only if the *antecedentCondition* succeeds and therefore  
 931 the former can be also interpreted as the precondition to apply the latter.

932 In the case of validation, the outcome is a Dataset (the validation output) having a Boolean measure (TRUE or  
 933 FALSE) and broken down at least by the Variables RULESET and RULE containing respectively the *rulesetId* and  
 934 the *ruleId* of the applied rule (for more details see the **check** operator). The variables ERRORCODE and  
 935 ERRORLEVEL are also added in the output Dataset and valued with the parameters *errorCode* and *errorLevel* of  
 936 the applied Rule in case of validation failure (i.e. FALSE value as outcome of the Rule).

937 These Rulesets can be also used to filter Datasets. In particular, the **filter** operator can apply a  
 938 Horizontal|DataPoint Ruleset to all the Data Points of the Dataset to be filtered. The result will be a new Dataset,  
 939 having the same data structure as the input Dataset and containing only the Data Points for which the Rules of  
 940 the Ruleset evaluates to TRUE or optionally to FALSE (for more details see the filter operator).

941

#### 942 *Examples*

943 1) Input Dataset:

ds_bop					
TIME	REF_AREA	PARTNER	FLOW	OBS_VALUE	OBS_STATUS
2010	EU25	CA	AVERAGE	20	
2010	BG	CA	NET	1	
2010	RO	CA	NET	1	M
2010	EU27	CA	CREDIT	12	C

944

```
945 define datapoint ruleset ruleset_1 (FLOW as x, OBS_STATUS as y) (  

  946     flow_dr : when x = "CREDIT" or x = "AVERAGE" then y <> "C" errorCode ( -XXXXX )  

  947 )
```

948

Meaning:

949

Once ruleset\_1 is defined, it is usable to perform validations or apply filters.

950

2)

951

```
ds := check(ds_bop, ruleset_1, with measures , only failures)
```

952

TIME	REF_AREA	PARTNER	FLOW	RULE_ID	OBS_VALUE	OBS_STATUS	ERRORCODE
2010	EU27	CA	CREDIT	ruleset1_flow_dr	12	C	-XXXXX

953

3)

954

```
ds := ds_bop[filter ruleset_1]
```

955

956

TIME	REF_AREA	PARTNER	FLOW	OBS_VALUE	OBS_STATUS
2010	EU25	CA	AVERAGE	20	
2010	BG	CA	NET	1	
2010	RO	CA	NET	1	M

957

## 958 define hierarchical ruleset

959

960

### Semantics

961

962

963

964

965

966

967

968

969

970

971

972

973

974

975

976

977

978

979

980

981

982

983

984

**define hierarchical ruleset** defines a persistent object that contains Rules to be applied to individual Components of a given Dataset in order to make validations or calculations according to hierarchical relationships between the relevant Code Items. These rulesets are also called “vertical” taking into account the tabular representation of a Data Set (considered as a mathematical function), in which each (vertical) column represents a Variable and each (horizontal) row represents a Data Point: these Rulesets are applied on Variables (columns), i.e. vertically on the tabular representation of a Data Set.

A first purpose of these Rules is to express some more aggregated Code Items (e.g. the continents) in terms of less aggregated ones (e.g., their countries). This kind of relations can be applied to aggregate data, for example to calculate an additive measure (e.g., the population) for the aggregated Code Items (e.g. the continents) as the sum of the corresponding measures of the less aggregated ones (e.g. their countries). If a certain information is available for both, the more and the less aggregated Code Items, these rules can be used for validating their mutual coherence, for example to check if the additive measures relevant to the aggregated Code Items (e.g. the continents) match the sum of the corresponding measures of their component Code Items (e.g. their countries).

Another purpose of these Rules is to express the relationships in which a Code Item represents some part of another one, (e.g., “Africa” and “Five largest countries of Africa”, being the latter a detail of the former). This kind of relationships can be used only for validation, for example to check if a positive and additive measure (e.g. the population) relevant to the more aggregated Code Item (e.g., Africa) is greater than the corresponding measure of the other one more detailed (e.g. “5 largest countries of Africa”).

The name “hierarchical” comes from the fact that this kind of Ruleset is able to express the hierarchical relationships between Code Items at different levels of detail, in which each (aggregated) Code Item is expressed as a partition of (disaggregated) ones.

As a first simple example, the following Hierarchical Ruleset named “BeneluxCountries” contains a single rule that asserts that, in the Value Domain “Geo\_Area”, the Code Item BENELUX is the aggregation of the Code Items BELGIUM, LUXEMBOURG and NETHERLANDS:

```
define hierarchical ruleset BeneluxCountriesHierarchy ( ValueDomain=Geo_Area ) is  
    BENELUX = BELGIUM + LUXEMBOURG + NETHERLANDS  
end hierarchical ruleset
```

988

989

### Syntax

990

991

992

993

```
define hierarchical ruleset rulesetId ( RulesetSignature ) is  
    { Rule } { ; Rule } *  
end hierarchical ruleset
```

994

995

996

997

998

999

1000

1001

1002

1003

1004

1005

1006

1007

1008

1009

1010

1011

1012

1013

1014

1015

1016

*RulesetSignature*

*::=* { *antecedentSignature*, } *codeItemRelationSignature*

*antecedentSignature*

*::=* **antecedentvariables**= *variable-signature* { , *variable-signature* } \*

*variable-signature*

*::=* *variable-ref* { **as** *constant-string* } ?

*codeItemRelationSignature*

*::=* [ **variable**= *variable-ref* | **valuedomain**= *valuedomain-ref* ]

*Rule*

*::=* { **ruleId** : } ? { **when** *antecedentCondition* **then** } ? *codeItemRelation*

{ **errorCode** ( *errorCode* ) } ?

{ **errorlevel** ( *errorLevel* ) } ?

*antecedentCondition*

*::=* *boolean-scalar-expression*

*codeItemRelation*

*::=* *codeItem-reference* [ = | > | < | >= | <= ] [ + | - ] ? *codeItemReference* { [ + | - ] *codeItemReference* } \*

*codeItemReference*

*::=* *codeItem-ref* [ **from** *time-ref* ] ? [ **to** *time-ref* ] ?

### Parameters

*rulesetId* : identifier

*ruleId* : identifier

1017 *codeItem-ref*: identifier  
 1018 *variable-ref*: identifier  
 1019 *valueDomain-ref*: identifier  
 1020 *antecedentCondition*: boolean-scalar-expression  
 1021 *errorCode*: string  
 1022 *errorLevel*: integer  
 1023 *time-ref*: time-literal  
 1024

- 1025 • *rulesetId* – the identifier of the Hierarchical Ruleset to be defined.
- 1026 • *rulesetSignature* – the signature of the Ruleset. It specifies the space on which the Ruleset is defined.
- 1027 • *antecedentSignature* - the signature of the antecedent conditions of the Ruleset. It specifies the Represented Variables (see the information model) on which the antecedent conditions of the Rules are defined.
- 1028 • *codeItemRelationSignature* - the signature of the Code Item Relations of the Ruleset. It specifies either the Represented Variable or the ValueDomain (see the information model) on which the Code Item Relations of the Rules are defined. When a Represented Variable is specified, the Ruleset is meant to be applicable to DataSets having such Variable as a Component. When a Value Domain is specified, the Ruleset is meant to be applicable to Datasets having a Component which takes values on it.
- 1029 • *variable-signature* – It specifies a single Represented Variable on which the Ruleset is defined
- 1030 • *variable-ref* – It references a Represented Variable by its name. The Variable name can be aliased for the sake of compactness in writing the Rules. If the alias is not specified, the complete name of the Variable must be used in the body of the Rules.
- 1031 • *constant-string*: the name assigned to the Variable within the ruleset
- 1032 • *valueDomain-ref* – It specifies a Value Domain
- 1033 • *Rule* – the complete specification of a single rule, as defined in the following parameters.
- 1034 • *ruleId* – the identifier of the specific rule within the Ruleset. The ruleId is optional and, if not specified, is assumed to be the progressive order number of the Rule in the Ruleset (please note that this practice may cause changes of the rule identifiers in case the Ruleset is maintained, e.g. if new rules are added or existing rules are deleted)
- 1035 • *antecedentCondition* – a Boolean scalar expression. All the Component level operators are allowed.
- 1036 • *CodeItemRelation* – the specification of a Code Item Relation to be evaluated only when the *antecedentCondition* evaluates to true (missing antecedent conditions are assumed as true). It expresses a logical relationship between Code Items belonging to the Value Domain referenced by the Ruleset. The relation is expressed by one of the symbols “=”, “>”, “>=”, “<”, “<=”, which in this case denote special logical relationships typical of Code Items (see below). The first member of the relationship is a single Code Item. The second member of the relationship is the composition of one or more Code Items expressed by the symbols “+” or “-”, which in their turn also denote special logical operators typical of Code Items (see below). The meaning of these symbols is explained below.
- 1037 • *codeItemReference* –the reference to an existing *Code Item* of the VTL information model, that is a Value of a ValueDomain.
- 1038 • *errorCode* – a string denoting the error code associated to the rule, respecting VTL conventions, in case the rule is used for validation.
- 1039 • *errorLevel* – an integer containing the error level (severity) associated to the rule, in case the rule is used for validation.

#### 1061 *Constraints*

- 1062 • *valueDomainReference* must be enumerated.
- 1063 • *antecedentCondition* must refer only to identifiers specified in *antecedentConditionIds*
- 1064 • *errorCode* must respect the conventions of user-defined error codes.

#### 1066 *Returns*

1067 A persistent Hierarchical (or vertical) Ruleset identified by *rulesetId*, which can be referenced and used both for validation and aggregation purposes.

#### 1070 *Semantic specification*

1071 This operator defines a Hierarchical Ruleset, which is a collection of Rules expressing logical relationships between the Values (Code Items) of a Variable or a Value Domain.

1072 Each rule contains an optional antecedent condition, which defines the cases in which the Rule has to be applied (if not declared the Rule is applied ever) and a mandatory code item relation, which expresses the **relation**

1075 **between Code Items** to be enforced. In the relation, one Code Item (the first member of the relation) is put in  
1076 relation to a combination of other Code Items.

1077 As for the mathematical meaning of the relation, please note that each Value (Code Item) is the representation of  
1078 an event belonging to a space of events (i.e. the relevant Value Domain), according to the notions of “event” and  
1079 “space of events” of the probability theory (see also the section on the Generic Models for Variables and Value  
1080 Domains in the VTL IM). Therefore the relations between Values (Code Items) express logical implications  
1081 between events.

1082 The envisaged types of relations are: “coincides” (=), “implies” (<), “implies or coincides” (<=), “is implied by” (>),  
1083 “is implied by or coincides” (>=)<sup>2</sup>. For example:

1084 UnitedKingdom < Europe means UnitedKingdom implies Europe

1085 In other words, this means that if a point of space belongs to United Kingdom it also belongs to Europe.

1086 January 2000 < year 2000 means January of the year 2000 implies the year 2000

1087 In other word, if a time instant belong to “January 2000” it also belongs to the “year 2000”

1088 The first member of a Relation is a single Code Item. The second member can be either a single code item, like in  
1089 the example above, or a **logical composition of Code Items** giving another Code Item as result. The logical  
1090 composition can be defined by means of Code Item Operators, whose goal is to compose some Code Items in  
1091 order to obtain another Code Item.

1092 Please note that the symbols “+” and “-“ do not denote the usual operations of sum and subtraction, but logical  
1093 operations between Code Items which are seen as events of the probability theory. In other words, two or more  
1094 Code Items cannot be summed or subtracted to obtain another Code Item, because they are events and not  
1095 numbers, however they can be manipulated through logical operations like “OR” and “Complement”.

1096 Note also that the “+” also acts as a declaration that all the Code Items denoted by “+” in the formula are mutually  
1097 exclusive one another (i.e. the corresponding events cannot happen at the same time), as well as the “-“ acts as a  
1098 declaration that all the Code Items denoted by “-“ in the formula are mutually exclusive one another and  
1099 furthermore that each one of them is a part of (implies) the result of the composition of all the Code Items having  
1100 the “+” sign.

1101 At intuitive level, the symbol “+” means “with” (Benelux = Belgium *with* Luxembourg *with* Netherland) while the  
1102 symbol “-“ means “without” (EUwithoutUK = EuropeanUnion *without* UnitedKingdom).

1103 When these relationships are applied to additive numeric measures (e.g. the population relevant to geographical  
1104 areas), they allow to obtain the measure values of the compound Code Items (i.e. the population of Benelux and  
1105 EUwithoutUK) by summing or subtracting the measure values relevant to the component Code Items (i.e. the  
1106 population of Belgium, Luxembourg and Netherland in the former case, EuropeanUnion and UnitedKingdom in  
1107 the latter). This is why these logical operations are denoted in VTL through the same symbols as the usual sum  
1108 and subtraction. Please note also that this is valid whichever is the Data Set and whichever is the additive  
1109 measure (provided that the possible other Identifier Components of the Data Set Structure have the same  
1110 values), so that the Rulesets of this kind are potentially reusable.

1111 The Ruleset Signature specifies the space on which the Ruleset is defined. The “antecedentSignature” specifies  
1112 the Variables on which the antecedent conditions of the Rules are defined (the Rules can refer only to these  
1113 Variables and must refer to all of them). The “codeItemRelationSignature” specifies either the Represented  
1114 Variable or the ValueDomain (see the information model) on which the Code Item Relations can be defined  
1115 (when a Represented Variable is specified, the Ruleset is meant to be applicable to DataSets having such Variable  
1116 as a Component, when a Value Domain is specified, the Ruleset is meant to be applicable to Datasets having a  
1117 Component which takes values on it).

1118 The Hierarchical Ruleset may act on one or more Measures of the input Data Set provided that these measures  
1119 are additive (for example it cannot be applied on a measure containing a “mean” because it is not additive).

1120

1121 If a **Hierarchical Ruleset is used for calculation** (see also the “Calc” operator), only the Relations expressing  
1122 coincidence (“=”) are evaluated (provided that the *antecedentCondition* is true). The result Data Set will contain  
1123 the compound Code Items (the left members of those relations) calculated from the component Code Items (the  
1124 right member of those Relations). Moreover, the clauses typical of the validation are ignored (e.g. ErrorCode,  
1125 ErrorLevel/Severity).

1126 If some Code Items are defined equal to themselves, the relevant Data Points are brought in the result  
1127 unchanged. For example, the following Ruleset will maintain in the result the Data Points of the input Data Set  
1128 relevant to Belgium, Luxembourg and Netherland and will add new Data Points containing the calculated value  
1129 for Benelux:

1130 **define hierarchical ruleset AddBenelux (valuedomain=GeoArea) is**  
1131 Belgium = Belgium  
1132 Luxembourg = Luxembourg

<sup>2</sup> “Coincides” means “implies and is implied”

1133 Netherlands = Netherlands  
 1134 Benelux = Belgium + Luxembourg + Netherlands  
 1135 **end hierarchical ruleset**

1136  
 1137 If a **Hierarchical Ruleset** is used for validation (see also the “Check” operators for more detailed information),  
 1138 all the possible Relations (“=”, “>”, “>=”, “<”, “<=”) are evaluated (provided that the *antecedentCondition* is true).  
 1139 The Rules are evaluated independently. The Data Points referred both by the left and the right members of the  
 1140 Relations are taken from the input Dataset. The Antecedent Condition is evaluated and, if “TRUE”, the Code Item  
 1141 Relation is also evaluated (the operations specified in the right member of the Relation are performed and the  
 1142 result is compared to the first member according to the specified Relation). The possible relations in which Code  
 1143 Items are defined as equal to themselves are ignored. The result Data Set will contain, as a Measure, the Boolean  
 1144 result of the validation, and, as Identifiers, the RulesetId, the RuleId and the Identifiers of the input Data Set. The  
 1145 possible clauses typical of the validation are applied (e.g. ErrorCode, ErrorLevel/Severity) and generate  
 1146 additional Measures in the result. Further options are better explained in the Check operator).  
 1147 Please note again that in case of validation the Data Points relevant to both the members of the Relations are  
 1148 expected to belong to the input Data Set. As obvious, if the data to be validated are originally in different  
 1149 DataSets, either they can be merged in advance using other VTL operators or the validation can be done by  
 1150 comparing those Data Sets directly (see also the Check operator), without using this kind of Ruleset.

1151 **The Hierarchical Rulesets allow to declare the time validity of Rules and Relations.** Firstly, the Antecedent  
 1152 Condition may be referred to a time variable, expressing when the Code Item Relation has to be applied (i.e.  
 1153 when it is considered valid as a whole). Secondly, each Code Item of the second member of the Code Item  
 1154 Relation can be qualified with a time validity, so expressing when the Code Item participates in the relation. As a  
 1155 default, when not expressed the validity is considered to be “ever”.

1156 The following two simplified examples show possible ways of defining the European Union in term of Countries.

1157 Example 1

1158 **define hierarchical ruleset** EuroAreaCountries1 (**antecedentvariable**=Time, **variable**=GeoArea) is  
 1159 when Time between 1.1.1958 and 31.12.1972  
 1160 then EU = BE + FR + DE + IT + LU + NL  
 1161 when Time between 1.1.1973 and 31.12.1980  
 1162 then EU = ... *same as above* ... + DK + IE + GB  
 1163 when Time between 1.1.1981 and 02.10.1985  
 1164 then EU = ... *same as above* ... + GR  
 1165 when Time between 1.1.1986 and 31.12.1994  
 1166 then EU = ... *same as above* ... + ES + PT  
 1167 when Time between 1.1.1995 and 30.04.2004  
 1168 then EU = ... *same as above* ... + AT + FI + SE  
 1169 when Time between 1.5.2004 and 31.12.2006  
 1170 then EU = ... *same as above* ... + CY + CZ + EE + HU + LT + LV + MT + PL + SI + SK  
 1171 when Time between 1.1.2007 and 30.06.2013  
 1172 then EU = ... *same as above* ... + BG + RO  
 1173 when Time >= 1.7.2013  
 1174 then EU = ... *same as above* ... + HR  
 1175 **end hierarchical ruleset**

1176 Example 2

1177 **define hierarchical ruleset** EuroAreaCountries2 (**valuedomain**=Geo\_Area) is  
 1178 EU = AT (from 1.1.1995) + BE (from 1.1.1958) + BG (from 1.1.2007)  
 1179 + ... + GB (from 1.1.1973) + ...  
 1180 + SE (from 1.1.1995) + SI (from 1.5.2004) + SK (from 1.5.2004)  
 1181 **end hierarchical ruleset**

1182 In this example, when GB will exit from UE, the GB term would become:  
 1183 + GB (from 1.1.1973 to ... the future date of Brexit ...)

1184 **The Hierarchical Rulesets allow defining hierarchies** either having or not having levels (free hierarchies).  
 1185 For example, leaving aside the time validity for sake of simplicity:

1186 **define hierarchical ruleset** GeoHierarchy (**valuedomain**=Geo\_Area) is  
 1187 World = Africa + America + Asia + Europe + Oceania

```

1188 Africa = Algeria + ... + Zimbabwe
1189 America = Argentina + ... + Venezuela
1190 Asia = Afghanistan + ... + Yemen
1191 Europe = Albania + ... + Vatican City
1192 Oceania = Australia + ... + Vanuatu
1193 Afghanistan = AF_reg_01 + ... + AF_reg_N
1194 ... ..
1195 Zimbabwe = ZW_reg_01 + ... + ZW_reg_M
1196 EuropeanUnion = ... + ... + ... + ...
1197 CentralAmericaCommonMarket = ... + ... + ... + ...
1198 OECD_Area = ... + ... + ... + ...
1199 end hierarchical ruleset

```

1200 Hierarchies may be useful for validation in case more levels of detail are contained in the Data Set to be  
1201 validated. The Hierarchical Rulesets defines the mutual coherency rules of these different levels of detail.  
1202 Because the various Rules can be evaluated independently, their order is not significant.

1203 Hierarchies may also be useful for calculations. For example, they can be used to calculate the upper levels of the  
1204 hierarchy if the data relevant to the leafs (or some other intermediate level) are available in the input Data Set.  
1205 For example, having additive measures broken by region, it would be possible to calculate these measures  
1206 broken by countries, continents and the world. Besides, having additive measures broken by country, it would be  
1207 possible to calculate the same measures broken by continents and the world.

1208 In the Hierarchies there can be dependencies between Rules, because the inputs of some Rules can be the output  
1209 of other Rules, so the former can be evaluated only after the latter. For example, the data relevant to the  
1210 Continents can be calculated only after the calculation of the data relevant to the Countries. As a consequence,  
1211 the order of calculation of the Rules is determined by their mutual dependencies and can be different from the  
1212 order of the Rules in the Ruleset. The dependencies between the Rules form a directed acyclic graph.

1213 **Hierarchical Rulesets allow defining multiple relations for the same Code Item.**

1214 Multiple relations are often useful for validation. For example, the Balance of Payments item "Transport" can be  
1215 broken down both by type of carrier (Air transport, Sea transport, Land transport) and by type of objects  
1216 transported (Passengers and Freights) and both breakdowns must sum up to the whole "Transport" figure. In  
1217 the following example a RuleId is assigned to the different methods of breaking down the Transport.

```

1218 define hierarchical ruleset TransportBreakdown (valuedomain= BoPItem ) is
1219     transport_method1:   Transport = AirTransport + SeaTransport + LandTransport,
1220     transport_method2:   Transport = PassengersTransport + FreightsTransport
1221 end hierarchical ruleset

```

1222 Multiple relations can be deemed as useful even in some case of calculation. For example, imagine that the input  
1223 Data Set contains data about resident units broken down by region and data about non-residents units broken  
1224 down by country. In order to calculate an homogeneous level of aggregation (e.g. by country), a possible Ruleset  
1225 might be the following:

```

1226 define hierarchical ruleset CalcCountryLevel ( valuedomain=Geo_Area ) is
1227     Country1 = Country1
1228     Country1 = Region11 + ... + Region1M
1229     ...
1230     CountryN = CountryN
1231     CountryN = Region N1 + ... + RegionNM
1232 end hierarchical ruleset

```

1233  
1234 A warning is opportune about the possible practice of calculating the same Code Item in more Rules (calculation  
1235 methods) of the same Ruleset. The Rulesets of this kind, in fact, may produce either right or wrong figures  
1236 depending on the content of the input Data Set.

1237 As a matter of fact, in the calculation the outcomes of all the Rules belonging to the Ruleset are aggregated  
1238 together to produce the final result, in order to remove possible duplicates in the Identifiers (duplicate values in  
1239 the Identifiers cannot be allowed, see also the Information Model). As far as each Code Item is defined just once  
1240 as left member of a relation, the values of the Identifiers of the results of the single Rules are all distinct and their  
1241 aggregation cannot generate inconsistencies. This is not ever true if a Code Item is defined more than once (e.g.  
1242 through more than one calculation method).

1243 In the Ruleset of the example above, each Country is calculated using two calculation methods, whose results  
1244 may have the same keys, which will be aggregated together. The output Data Set will be correct provided that, in  
1245 the input Data Set, any information is present either by country or by region (never both of them). The output

1246 Data Set would contain errors if some information is present in the input Data Set both by country and by region:  
1247 the resulting figures would be indicatively (and wrongly) doubled.  
1248 In general, if more left members refer to the same Code Item (in other words, if a Code Item is calculated through  
1249 more calculation methods), the result may be inconsistent for some input DataSets. It is possible to avoid these  
1250 situations by using other approaches for calculating the desired result (e.g. splitting the Ruleset, calculating the  
1251 result in more steps, using antecedentConditions, using other VTL operators). This example has been presented  
1252 to better clarify the behavior of this kind of Ruleset and warn about possible limitations to its reusability.  
1253  
1254

### 1255 *Examples*

1256 1) The Code Item Relation is defined on the Variable "sex": Total is defined as Male + Female.  
1257 No antecedent conditions are defined.

```
1258  
1259 define hierarchical ruleset vr_sex (Variable= sex) is  
1260     TOTAL = MALE + FEMALE;  
1261 end hierarchical ruleset
```

1262  
1263 2) BENELUX is the aggregation of the Code Items BELGIUM, LUXEMBOURG and NETHERLANDS, if not true, the  
1264 errorcode is 2000 and the errorlevel is high (10)

```
1265 define hierarchical ruleset BeneluxCountriesHierarchy ( valuedomain=Geo_Area) is  
1266     BENELUX = BELGIUM + LUXEMBOURG + NETHERLANDS errorcode 2000 errorlevel 10  
1267 end hierarchical ruleset
```

1268  
1269 3) American economic partners. The first rule verifies that the value reported for North America is greater than the  
1270 value reported for US. This type of validation is useful when the data communicated by the data provider do not cover  
1271 the whole composition of the aggregate but only the main elements. No antecedent conditions are defined.

```
1272  
1273 define hierarchical ruleset vr_american_partners (variable= counterpart_area) is  
1274     NORTH_AMERICA > US ;  
1275     SOUTH_AMERICA = BR + UY + AR + CL ;  
1276 end hierarchical ruleset
```

1277  
1278 4) Example of item having multiple definitions. The Balance of Payments item "Transport" can be broken down by type  
1279 of carrier (Air transport, Sea transport, Land transport) and by type of objects transported (Passengers and Freights) and  
1280 both breakdowns must sum up to the total "Transport" figure.

```
1281  
1282 define hierarchical ruleset vr_bop (variable= bop_item ) is  
1283     transport_method1 : Transport = AirTransport + SeaTransport + LandTransport,  
1284     transport_method2 : Transport = PassengersTransport + FreightsTransport  
1285 end hierarchical ruleset
```

## 1287 **define mapping ruleset**

### 1288 *Semantics*

1289 The **define mapping** allows to transcode a set of values of an Identifier Component

### 1290 *Syntax*

```
1291 define mapping map_1 (  
1292     { condition ( IdentifierComponent<?> idCond { IdentifierComponent<?> idCond } * ) }  
1293     map_to ( IdentifierComponent<?> idMapTo )  
1294     map_from ( IdentifierComponent<?> idMapFrom )  
1295 ) is  
1296     { MappingRule ; } +
```

```
1297  
1298  
1299 MappingRule:= { when Component<Boolean> whenCondition then }  
1300                 IdentifierValue valueTo = IdentifierValue valueFrom
```

```
1301 end mapping ruleset
```

1302

1303 *Parameters*

1304 *idCond* : identifier

1305 *idMapTo* : component-ref

1306 *idMapFrom* : component-ref

1307 *whenCondition* : boolean

1308 *valueTo* : string

1309 *valueFrom* : string

1310

- 1311 • *idCond* is the identifier used in the condition part. More than one identifier can be used.
- 1312 • *idMapTo* is the identifier whose values are resulting from the conversion of values of *idMapFrom*
- 1313 • *idMapFrom* is the identifier whose values are converted to values of *idMapTo*
- 1314 • *whenCondition* is a boolean expression. When *whenCondition* is evaluated to true then the corresponding mapping rule is executed. If *whenCondition* is omitted in a rule then it is implicitly assumed to be true.
- 1315
- 1316 • *valueTo* is a valid value for *idMapTo*
- 1317 • *valueFrom* is a valid value for *idMapFrom*

1318

1319 *Constraints*

1320 *idCond*, *idMapFrom* and *idMapTo* are the names of existing Identifier Components). *valueTo* is a valid value for

1321 *idMapTo* and *valueFrom* is a valid value for *idMapFrom*.

1322

1323 *Semantic specification*

1324 It creates a mapping that can be applied to transcode a set of values using the **transcode** statement. A mapping is

1325 a set of rules for transcoding values belonging to the code lists of two identifier components.

1326

1327 *Returns*

1328 None.

1329

1330 *Examples*

1331 See the examples under the **transcode** operator.

1332



## 1333 VTL-ML - General purpose operators and functions

1334

### 1335 Parentheses ( )

#### 1336 *Semantics*

1337 The parenthesis allows to modify the default order of evaluation of the operators.

1338

#### 1339 *Syntax*

1340 ( *expression* )

1341

#### 1342 *Constraints*

1343 None.

1344

#### 1345 *Semantic specification*

1346

1347

### 1348 Assignment :=

#### 1349 *Semantics*

1350 The “:=” symbol allows to assign the value of an expression to a variable parameter.

1351

#### 1352 *Syntax*

1353 *variable\_parameter* := *expression*

1354

#### 1355 *Constraints*

1356 None.

1357

#### 1358 *Semantic specification*

1359 the *expression* may evaluate to any data type.

1360

#### 1361 *Examples*

1362 Assignment of a Constant<Number> value to a parameter:

1363     numpi := 3.14

1364 Assignment of a String value to a parameter:

1365     str := “hello world”

1366 Assignment of an expression to a parameter:

1367     popA := populationDS + 1

1368 Assignment of a Dataset expression to a parameter:

1369     ds\_1 := get(“NAMESPACE/DF\_NAME/2000.USD.M.F.A.BOP.ANN.STO.EABL”)

1370 Assignment of a Constant<Boolean> value to a parameter:

1371     bool\_var := true

1372

### 1373 Membership .

#### 1374 *Semantics*

1375 The membership operator allows to specify a single component of a Dataset

1376

#### 1377 *Syntax*

1378 *ds* . *comp*

#### 1379 *Parameters*

1380 *ds* : Dataset

1381 *comp* : Dataset component-ref

1382

1383 • *ds* – is a Dataset

1384 • *comp* – a valid component of *ds*

1385

1386 *Constraints*

1387 *None.*

1388

1389 *Returns*

1390 A Dataset having all the identifiers and only one Measure or Attribute *c* specified by the operator.

1391

1392 *Semantic specification*

1393 The membership operator is particularly useful to work with operators that have specific constraints in terms of  
1394 the types of the Measure Components or have more than one Measure Component.

1395

1396 *Examples*

1397 1) Suppose *ds\_1* is a multi-measure Dataset, where *M1* is a numeric Measure Component and *M2* is a string  
1398 Measure Component, let *ds\_2* be a mono-measure Dataset with a single Measure Component *M1*. *ds\_1* and *ds\_2*  
1399 have the same Identifier Components. Let us supposed the sum *ds\_1* + *ds\_2* is desired.

1400 The following syntax: *ds\_1.M1* + *ds\_2* represents the resulting Dataset. In this notation *ds\_1* is temporarily  
1401 considered mono-measure

1402

1403 2) Suppose the comparison operator (“=”) needs to be applied on the Component *COUNTRY* of the Dataset *ds\_1*.  
1404 In this expression:

1405 *ds\_2 := ds\_1.COUNTRY="Luxembourg"*

1406 the membership operator specifies that the Identifier Component *COUNTRY* is temporarily considered as the  
1407 only Measure Component to be used in the comparison.

1408

1409 3) Suppose it is needed to round an Component. The round operator acts on Measure Components, which must  
1410 be all Numeric. Suppose we have a Dataset *ds\_1* with a string Measure Component *DESCRIPTION* and a numeric  
1411 Component *AVERAGE\_AGE*, which needs to be rounded to the 3rd decimal. The expression:

1412 *ds\_1 := round(ds\_1.AVERAGE\_AGE,3)*

1413 performs this task.

1414 *ds\_1.AVERAGE\_AGE* temporarily considers *AVERAGE\_AGE* the only numeric Measure Component of *ds\_1*. The  
1415 round is then normally applied.

1416

1417 4) Let us suppose we have two multi-measure Datasets *ds\_1* and *ds\_2*, having the same Identifier Components  
1418 *K1* and *K2*, and the same Measure Components *M1* (which is a Numeric), *M2* which is a String.

1419 The expression:

1420 *ds\_3 := ds\_1.M1 + ds\_2.M1*

1421 sums only the Measure Component *M1*.

1422

1423 5) Let us suppose we have two multi-measure Datasets *ds\_1* and *ds\_2*, having the same Identifier Components  
1424 *K1* and *K2*, and the Measure Components *M1* and *M2*.

1425 The expression:

1426 *ds\_3 := ds\_1.M1 + ds\_2.M2*

1427 sums the Measure Component *M1* with the measure component *M2*.

## 1428 **Alias as**

1429 *Semantics*

1430 The **as** operator allows to rename one component of a Dataset or the component resulting by an expression.

1431

1432 *Syntax*

1433 *ds.comp as alias*

1434

1435 *Parameters*

1436 *ds* : Dataset

1437 *alias* : string

1438

1439 *Constraints*

1440 This operator works only on Measure components.

1441 *Semantics specification*

1442 The operator takes as input a Dataset and the identifier of a Measure Component, and returns a new Dataset  
1443 having only that Measure Component and all the original Identifier Components.

1444 *Examples*

1445 1) Let us suppose we have two multi-measure Datasets *ds\_1* and *ds\_2*, having the same Identifier Components *K1* and  
1446 *K2*, and the Measure Components *M1* and *M2*.

1447 The expression:

1448  $ds_3 := ds_1.M1 + ds_2.M2$  as "M1"

1449 sums the Measure Component *M1* with the measure component *M2*. The outcome Dataset has one Measure  
1450 Components: *M1*, which is obtained as the sum of *M1* in *ds\_1* and *M2* in *ds\_2*.

1451

1452 2) Let us suppose we have a Datasets *ds\_1* and the Measure Components *M1*.

1453 The expression:

1454  $ds_2 := ds_1.M1 * 10$  as "M2"

1455 returns a Dataset having only one measure components *M2* obtained as the product of *M1* and 10.

1456

1457

## 1458 alterDataset

1459 *Semantics*

1460 The **alterDataset** allows to maintain all or a subset of components of the input Dataset having the identifier role.

1461

1462 *Syntax*

1463 **alterDataset( *ds\_1*{, *compList*} { **all** } );**

1464

1465 *Parameters*

1466 *ds\_1* : dataset {identifier <IDENT> as scalar-type}+{measure <IDENT> as scalar-type}\*  
1467 {attribute <IDENT> as scalar-type}\*

1468 *compList* : list<list<component-ref>>

1469

- 1470 • *ds\_1* – is the Dataset that the operator uses to produce the resulting Dataset.
- 1471 • *compList*– is the set of components belonging to the input Dataset.
- 1472 • **all** – its definition implies the presence of all components of *ds\_1* in the resulting Dataset.

1473

1474 *Constraints*

1475 None.

1476

1477 *Returns*

1478 This operator returns a Dataset having only Identifiers Components. The components of the returned Dataset are  
1479 all the components of the input Dataset that are part of the *compList* or, if it is not specified, only the identifier  
1480 components of the input Dataset. If one or more measures or attributes are included in the list, they will be part  
1481 of the returned Dataset but having a role of identifiers. This operator allows removing identifier components  
1482 from the input Dataset removing duplications.

1483

1484 *Semantic specification*

1485 The Dataset resulting will have only Identifiers also if it contains components that were previously measures. If  
1486 the **with measures** flag is specified then the resulting Set will have as added Identifiers, the Measures  
1487 Components of the input one Dataset, too.

1488

1489

1490 *Examples*

1491

1492 1) **alterDataset(ds\_1 all)**

1493

ds_1		
K1	K2	M1
1	A	100
2	B	200

1494

set_1		
K1	K2	M1
1	A	100
2	B	200

1495

1496

1497 2) l\_1 = list<components-ref> (REF\_AREA)

1498

1499

**alterDataset(ds\_1,l1)**

1500

IT_nord_pop		
TIME	REF_AREA	OBS_VALUE
2015	ITCD	27799803
2015	ITC	16138643
2015	ITC1	4424467
2015	ITC2	128298
2015	ITC3	1583263
2015	ITC4	10002615
2015	ITD	11661160
2015	ITD1	518518
2015	ITD2	537416
2015	ITD3	4927596
2015	ITD4	1227122
2015	ITD5	4450508
2014	ITCD	27785211
2014	ITC	16130725
2014	ITC1	4436798
2014	ITC2	128591
2014	ITC3	1591939
2014	ITC4	9973397
2014	ITD	11654486
2014	ITD1	515714
2014	ITD2	536237
2014	ITD3	4926818
2014	ITD4	1229363
2014	ITD5	4446354

1501

1502

set_1
<b>REF_AREA</b>
ITCD
ITC
ITC1
ITC2
ITC3
ITC4
ITD
ITD1
ITD2
ITD3
ITD4
ITD5

1503

## 1504 get

### 1505 *Semantics*

1506 The **get** operator allows to fetch all the instances of a Dataset from the system and returns a Dataset containing  
1507 them.

1508

### 1509 *Syntax*

```
1510 get(
1511     ds_id {, ds_id}*
1512     {keep(keepPart {, keepPart }*)}
1513     {dedup(consResFunctions)}
1514     {filter(filterPart)}
1515     {aggregate( aggregateFunction (aggrPart {, aggrPart}*)*)}
1516 )
```

1517

### 1518 *Parameters*

1519 *ds\_id* : ident

1520 *consResFunctions* : list<component-ref\* (t\*t) -> t > (t is the type of the referred Component)

1521 *keepPart* : component-ref

1522 *filterPart* : boolean

1523 *aggrPart* : component-ref (Component<Numeric>)

1524

- 1525 • *ds\_id* – is the Persistent Dataset to be fetched.
- 1526 • *keepPart* – is a valid reference to a Component of *ds\_id*.
- 1527 • *consResFunctions* – is a List of reference to valid Components of *ds* and conflict resolution Function.
- 1528 • *filterPart* – is a boolean Component expression which is evaluated row-wise and states if a row is to be kept  
1529 (if evaluates to true) or removed (if it does not evaluate to true) from the result.
- 1530 • *aggrPart* – is a valid reference to the numeric Measure Component to aggregate.

1531

### 1532 *Constraints*

- 1533 • All the input Datasets *ds\_id* must be persistent (see put operator) and must have the same Logical Data  
1534 Structure, which is the same Components in number, name and type (static).
- 1535 • If more than a Dataset *ds\_id* is defined, then the definition of *consResFunctions* is mandatory.
- 1536 • The *consResFunction* List, must defines a conflict resolution function for each Measure Component specified  
1537 in the keep clause. For each Component the respective conflict resolution function must return a value of the

1538 same type (as explained in the syntax). If `consResFunction` is not used and duplicated records are present  
1539 the `get` operator return an error.

- 1540 • `keepPart` must be a Component expression containing exactly the name of a Component of any `ds` (complex  
1541 Component expressions, combining more than one Component are not allowed) (static).
- 1542 • `aggrPart` must be a Component expression containing exactly the name of a Measure Component present in  
1543 any `ds` (no complex Component expressions, combining more than one Component is allowed). If there is at  
1544 least one `aggrPart`, there must be one for each Measure Component that is present in a `keepPart`. If `keepPart`  
1545 is omitted, all Measure Components must be in the aggregate. This means that there cannot be Measure  
1546 Components, kept that are not used in aggregations (static).

#### 1547 *Returns*

1548 A Dataset obtained as the union of all the Datasets specified by the identifiers `ds`, keeping only the columns  
1549 specified in the `keepParts` and the rows in the `filterParts`, choosing from duplicate Datapoints through  
1550 `consResFunctions`, aggregating over all the Measure Components in `aggrParts` grouping.

#### 1551 *Semantic specification*

1552 The operator **get**, is the data retrieval command. It takes in input a number (at least one) of Dataset `ds`. Together  
1553 with `put`, it is the only operator in VTL where a persistent Dataset can be mentioned.

1554 The command operates as follows: considers all the instances of the identified Dataset (selected according to the  
1555 semantics of the identifier); builds a union without duplicates (conflicts are resolved using the `consResFunctions`  
1556 specified in the dedup part); keeps in the result only the Components that are present in the `keepPart` (like SQL  
1557 SELECT). If the keep part is omitted, all the Components are preserved in the result; selects the only instances  
1558 returning `true` for the `filterPart` boolean Component expression. For the `filterPart`, any complex boolean  
1559 Component expression over all the Datasets Components (not only the ones mentioned in the `keepPart` can be  
1560 used) and it is evaluated row-wise (like SQL WHERE).

1561 Finally, the command aggregates (like SQL aggregations and GROUP BY) applying an aggregation function (see  
1562 aggregate function operator) over the Measure Component specified in `aggrPart` grouping by the Identifier  
1563 Components that are kept in the `keepPart` (or all if there is no `keepPart`).

1564 NULL values are considered in aggregations only if the `"include NULLS"` part is present. Specifically, they  
1565 propagate as usual resulting in a NULL sum, average or median if at least one NULL is present among the values;  
1566 in a NULL minimum or maximum if the only value to aggregate coincides with NULL; they are considered as  
1567 always distinct in both count and count\_distinct.

1568 Viceversa, if `include NULLS` part is absent, NULL values are not considered in aggregations.

#### 1571 *Examples*

1572 1) The expression:

```
1573 ds_1 := get("DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO", keep(K1, K2, M1))
```

1574 Retrieves Dataset identified by `DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO` from the system, keeping the  
1575 Identifier Components K1 and K2 and the Measure Component M1.

1576

1577

1578

DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO		
K1	K2	M1
1	A	5
2	B	7

1579

1580

ds_1		
K1	K2	M1
1	A	5
2	B	7

1581

1582 2) The expression:

```
1583 ds_1 := get("DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO", "DF_NAME/2011-2012.USD.M.F.A.BOP.ANN.STO",  
1584 keep(K1, K2, K3, M1), dedup(M1*min))
```

1585  
 1586 Retrieves the union of Datasets identified by DF\_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO and  
 1587 DF\_NAME/2011-2012.USD.M.F.A.BOP.ANN.STO, keeping the Identifier Components K1, K2 and K3 and the  
 1588 Measure Component M1 for all of them.  
 1589

DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO			
K1	K2	K3	M1
1	A	X	5
2	B	Y	7

1590  
 1591

DF_NAME/2011-2012.USD.M.F.A.BOP.ANN.STO			
K1	K2	K3	M1
1	A	X	6
2	B	Y	7
3	C	Z	9

1592  
 1593

ds_1			
K1	K2	K3	M1
1	A	X	5
2	B	Y	7
3	C	Z	9

1594  
 1595

1596 The union had produced two duplicates: (1,A,X,5) and (1,A,X,6), (2,B,Y,7) and (2,B,Y,7). The min conflict resolution  
 1597 function take care of the minimum value for M1 between the duplicates.  
 1598

1599

1600 3) The expression:

1601 ds\_1 := **get**("DF\_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO", ("DF\_NAME/2011-2012.USD.M.F.A.BOP.ANN.STO",  
 1602 keep(K1, K2, K3, M1), dedup(M1\*min), filter(K3="X")))

1603

1604 retrieves the union of Datasets identified by DF\_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO and DF\_NAME/2011-  
 1605 2012.USD.M.F.A.BOP.ANN.STO keeping the Identifier Components K1, K2 and K3 and the Measure Component M1  
 1606 for all of them and selecting only the rows where the value of the Component K3 equals to the Constant<String> "X".  
 1607

NAMESPACE/DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO			
K1	K2	K3	M1
1	A	X	5
2	B	Y	7

1608  
 1609

NAMESPACE/DF_NAME/2011-2012.USD.M.F.A.BOP.ANN.STO			
K1	K2	K3	M1
1	A	X	6

2	B	Y	7
3	C	Z	9

1610  
1611

ds_1			
K1	K2	K3	M1
1	A	X	5

1612  
1613  
1614

4)

The expression:

```
ds_1 := get(
  "DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO", "DF_NAME/2011-2012.USD.M.F.A.BOP.ANN.STO",
  keep(K1, K2, M1), dedup(M1*min), aggregate(sum(M1)))
```

retrieves the union of Datasets identified by

DF\_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO

and

DF\_NAME/2011-2012.USD.M.F.A.BOP.ANN.STO

keeping the Identifier Components K1 and K2 and the Measure Component M1 for all of them. It aggregates over the Measure Component M1, grouping by the Identifier Components K1 and K2.

1625

DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO				
K1	K2	K3	M1	M2
1	A	X	5	2
2	B	Y	7	3

1626  
1627

DF_NAME/2011-2012.USD.M.F.A.BOP.ANN.STO				
K1	K2	K3	M1	M2
1	A	Y	6	5
2	B	Y	7	7
3	C	Z	9	11

1628  
1629

ds_1		
K1	K2	M1
1	A	11
2	B	14
3	C	9

1630  
1631

5) The expression:

```
ds_1 := get("DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO", "DF_NAME/2011-2012.USD.M.F.A.BOP.ANN.STO",
  keep(K1, K2, M1,M2), dedup(M1*min, M2*first_value), filter(K3>5 or K3=1), aggregate(sum(M1),max(M2)))
```

1635

retrieves the union of Datasets identified by DF\_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO and DF\_NAME/2011-2012.USD.M.F.A.BOP.ANN.STO keeping the Identifier Components K1 and K2 and the Measure Components M1 and M2 for all of them. It selects only the rows where K3 is greater than 5 or exactly 1. It aggregates over the Measure Component M1 by sum, over the Measure Component M2 by max, grouping by the Identifier Components K1 and K2.

1640



DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO				
K1	K2	K3	M1	M2
1	A	10	5	2
2	B	1	7	3

1641  
1642

DF_NAME/2011-2012.USD.M.F.A.BOP.ANN.STO				
K1	K2	K3	M1	M2
1	A	25	6	5
2	B	1	7	7
3	C	3	9	11

1643

ds_1			
K1	K2	M1	M2
1	A	11	5
2	B	14	7

1644

## 1645 put

### 1646 *Semantics*

1647 It stores the content of a Dataset expression *ds* into a persistent Dataset.

1648

### 1649 *Syntax*

1650 **put**(*ds*, *ds\_id*)

1651

### 1652 *Parameters*

1653 *ds*, *ds\_id* : dataset {identifier <IDENT> as scalar-type}+

1654 {measure <IDENT> as scalar-type}\* {attribute <IDENT> as scalar-type}\*  
1655

- 1656 • *ds* – is the Dataset, or Dataset expression which contents must be stored in the system.
- 1657 • *ds\_id* – is the Dataset that will assumes the contents of *ds*, it will be persistent in the system.

1658

### 1659 *Constraints*

1660 The Logical Data Structure of *ds* must conform to the one of the Dataset in the system that is identified by *ds\_id*  
1661 (static).

1662

### 1663 *Returns*

1664 A Dataset that is a copy of the input one *ds*.

1665

### 1666 *Examples*

1667 1) The expression below is to store the *ds\_1* Dataset.

1668

1669 `ds := put(ds_1, "DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO")`

1670

1671 2) `ds := put(log((ds_1 + ds_2),10), "DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO")`

1672 The result of logarithm is stored, while the sum is not persistent.

1673

1674 3) `ds := put(log(put(ds_1 + ds_2, "DF_NAME/2000-2011.USD.M.F.A.BOP.ANN.STO"),10),"DF_NAME/2000-  
1675 2010.USD.M.F.A.BOP.ANN.STO")`

1676 Both the results of the sum and the logarithm are stored into the system. The fact that `put` outputs the input  
1677 expression allows for this kind of use.

## 1678 eval

### 1679 *Semantics*

1680 The **eval** operator allows to execute an external, non-VTL program, and returns its result as a Dataset.

1681

### 1682 *Syntax*

```
1683 eval (Constant<String> language,  
1684      [{script=}Constant<String> script | Constant<String> programPath],  
1685      [{params=}ConstantList<?> parameterList}  
1686      , {dataset=}PersistentDataset ds_id)
```

1687

### 1688 *Parameters*

```
1689 ds_id : dataset {identifier <IDENT> as scalar-type}+  
1690           {measure <IDENT> as scalar-type}* {attribute <IDENT> as scalar-type}*
```

```
1691 language: string
```

```
1692 script: string
```

```
1693 programPath : string
```

```
1694 parameterList : list<scalar-type>
```

1695

1696 *ds\_id* – the PersistentDataset the program saves into.

1697 *language* – is the programming language of the script.

1698 *script* – is the code of the script.

1699 *programPath* – a path to a script file.

1700 *parameterList* – the List of input parameters for the script.

1701

### 1702 *Constraints*

- 1703 • language must be the name of a programming language, meaningful and executable in the target system  
1704 (such as a SQL stored-procedure language, R, STATA, etc.) (dynamic).
- 1705 • script must be the code of a program, valid with respect to the specified language. The program can  
1706 perform whatever internal logic, but is forced to calculate and autonomously store exactly one  
1707 PersistentDataset, *ds\_id* (dynamic).
- 1708 • programPath must be a valid path in the target system to a program file, compliant with the specified  
1709 language. It does not necessarily correspond to a filesystem file, but can also be the identifier of a DBMS  
1710 stored procedure, and so forth (dynamic).
- 1711 • parameterList must be compatible in order and type with the input parameters of the script (dynamic).

1712

### 1713 *Semantic specification*

1714 The program specified in the eval operator, is user-defined and can perform any internal logic, however it has to  
1715 adhere to some conventions:

- 1716 • it can take as input only String or Numeric parameters, which are directly bound to parameterList;
- 1717 • it must autonomously store its results into a single Dataset *ds\_id*. Indeed, the operator fetches the saved  
1718 Dataset (like a common get operation) and returns it as output, which can be handled within other VTL  
1719 expressions;
- 1720 • it must calculate exactly one Dataset;
- 1721 • it cannot refer to a parameter variable, but can only work with physical objects, such as relational tables  
1722 (for SQL), data frames (for R), which are loaded autonomously by the program with the appropriate  
1723 commands. Therefore, if a Dataset that has been calculated in a previous step needs to be used within a  
1724 user-defined program, it must be stored (with a put) into the system and loaded appropriately by the  
1725 program logic afterwards;
- 1726 • it must return 0 if it has terminated correctly, a negative number otherwise.

## 1727 Join expression

### 1728 *Semantics*

1729 The join expression implements some of the features of the FLWOR expression described in the VTL User  
1730 Manual.

1731

### 1732 *Syntax*

```
1733 { [ join_clause ] } { body }
```

1734  
1735 join\_clause ::= { [**inner** | **outer** | **cross**] } { ds { , ds \* } on dim { , dim } \* }  
1736 body := { clause { , clause } \* }  
1737 clause := calc\_clause | drop\_clause | filter\_clause | keep\_clause | rename\_clause | unfold\_clause | fold\_clause  
1738 calc\_clause := { role } compName = k  
1739 drop\_clause ::= **drop** { cmp { , cmp } \* }  
1740 keep\_clause ::= **keep** { cmp { , cmp } \* }  
1741 filter\_clause ::= **filter** boolean-expression | dpr  
1742 rename\_clause ::= **rename** cmp **to** cmp { , cmp **to** cmp }  
1743 unfold\_clause ::= **unfold** dim , msr **to** elem { , elem }  
1744 fold\_clause ::= **fold** elem { , elem } **to** dim , msr  
1745 role := **identifier** | **measure** | **attribute**

#### 1746 *Parameters*

1747 ds : [ dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as scalar-type }\*  
1748 {attribute <IDENT> as scalar-type} \* ]

1749 cmp : Component

1750 dim : IdentifierComponent

1751 dpr : name of a data point ruleset

1752

1753 ds – is a Dataset name

1754 alias – is an alias for a Dataset, to be used when the same Dataset appears several times in

1755 k – is a scalar expression, or a Dataset expression denoting a single measure or attribute

1756 role – is the role of the calculated component c. If omitted then the role is derived from k (if k is a Dataset  
1757 expression) otherwise the default role of c is **attribute**.

1758 dim – is an Identifier Component common to all Datasets specified in the join clause

1759

#### 1760 *Constraints*

1761 For inner and outer joins (see below), one of the Datasets specified in the *join\_clause* must contain all Identifier  
1762 Components from all other Datasets from the join\_clause (with the same name and the basic scalar type, number,  
1763 boolean, string, or date).

1764 The name of the component cannot be **filter**, **keep** or **rename**, or those names must be quoted within "".

1765 A Dataset ds should appear only once in the list of Datasets.

1766

#### 1767 *Returns*

1768 The Dataset returned by the last statement of the body.

1769

#### 1770 *Semantic specification*

1771 This operator implements some of the features of the FLWOR expression described in the VTL manual part 1.

1772 Only the features that are useful for validation and transformation purposes are retained in the VTL operator.

1773 First VTL executes the join clause and then the body.

1774 The statements are executed in the specified order and operate on an input working Dataset.

1775 The Dataset resulting from the join clause is the input for the first statement of the body.

1776 The Dataset resulting from a statement is the input for the following statement.

1777 The Dataset returned by the last statement of the body is returned as the final result of the join expression.

1778

#### 1779 *join\_clause*

1780 The meaning of the **inner** and the **outer** join is the same as the meaning of INNER JOIN and FULL OUTER JOIN  
1781 constructs, respectively, in the SQL-92 standard. These are the differences:

1782 **inner** ds1, ds2 the resulting Dataset contains the data points that exist both in ds1 and ds2 (i.e. the common  
1783 Identifier Components of ds1 and ds2 have the same values in ds1 and ds2).

1784 **outer** ds1, ds2 the resulting Dataset contains the data points that exist either in ds1 or ds2. Measures and  
1785 attributes of data points that exist only in ds1 or ds2 (but not in both) have the **null** value.

1786 **cross** ds1, ds2 the resulting Dataset contains all data points of ds1 combined with all data points of ds2 (i.e.  
1787 the Cartesian product of ds1 and ds2). The statements contained in the body are expected to  
1788 reduce the number of data points by filtering them as needed. Measures and attributes of  
1789 data points that exist only in ds1 or ds2 (but not in both) have the **null** value.

1790 The join clause builds the input Dataset of the first statement, according to the following rules.

1791

- 1792 • If the join clause contains a single *Dataset* then that *Dataset* is the initial working *Dataset*. It is possible  
1793 to refer to the components of the *Dataset* simply by using their name. Suppose that ds1 has a measure  
1794 m, then  
1795 `[ ds1 ] { a = m + 1 }` correct
- 1796 • For **inner**, **outer** and **cross** joins, the initial working *Dataset* is the result of the inner or outer or cross  
1797 join applied to the *Datasets* specified in the join clause. If the *Datasets* have common measures or  
1798 attributes (i.e. with identical names) then it is mandatory to refer to those components by specifying  
1799 both the *Dataset* name and the measure name. Suppose that ds1 and ds2 have a common measure m,  
1800 then:  
1801 `[ds1,ds2] { a = ds1.m + 1 }` correct  
1802 `[ds1,ds2] { a = m + 1 }` not correct (ambiguous: m can refer to ds1 or ds2)  
1803 The measures can be renamed with the rename clause:  
1804 `[ds1,ds2] { rename 'ds1.m' to m1 , a = m1 + 1 }` correct  
1805 The use of the quotation is necessary because ds1.m syntactically is not a valid name (this exception to  
1806 the syntax rules is allowed only in the join body).  
1807 In the final result of the join expression the common measures and attributes that have not been  
1808 renamed are automatically dropped. The same applies when the working Dataset is the input for a filter  
1809 that uses a datapoint (horizontal) ruleset.
- 1810 • If the **on** clause is specified then the join is possibly defined on a subset of the common **Identifier**  
1811 **Components** of the *Datasets*. If the *Datasets* have common **Identifier Components** (i.e. with identical  
1812 names, data type and values domain) that are not specified in the **on** clause then it is mandatory to refer  
1813 to those **Identifier Components** by specifying both the *Dataset* name and the measure name. For  
1814 example, if ds1 and ds2 have some common Identifier Components d1, d2 and d3, the following  
1815 expression:  
1816 `[ ds1,ds2 on d1, d2 ]`  
1817 returns a Dataset with the following Identifier Components:  
1818 d1, d2, 'ds1.d3', 'ds2.d3'  
1819 the Identifier Components can be renamed using the rename clause:  
1820 `[ ds1,ds2 on d1, d2 ] { rename 'ds1.d3' to new1, 'ds2.d3' to new2 }`  
1821 In the final result of join expression the common Identifier Components that are not listed in the **on**  
1822 clause and have not been renamed are automatically renamed by replacing the "." with an underscore  
1823 "\_". The same applies when the working Dataset is the input for a filter that uses a datapoint (horizontal)  
1824 ruleset.
- 1825 • The join clause can be omitted. In this case VTL implicitly adds a join clause containing all *Datasets* that  
1826 are used inside the body. For example, the following join expression:  
1827 `{ a = ds1.m1 + ds2.m1 }`  
1828 is automatically treated by VTL as equivalent to:  
1829 `[inner ds1, ds2] { a = ds1.m1 + ds2.m1 }`  
1830 and  
1831 `[ outer ] { a = ds1.m1 + ds2.m1 }`  
1832 is equivalent to:  
1833 `[outer ds1, ds2] { a = ds1.m1 + ds2.m1 }`  
1834

### 1835 Examples

1836 1) inner join returns data points that exists in both Datasets

```
1837 ds3 := [ ds1, ds2 ] {
1838     obs_value = ds1.obs_value + ds2.obs_value ,
1839     obs_status = ds1.obs_status
1840 }
```

ds1				
TIME	REF_AREA	PARTNER	OBS_VALUE	OBS_STATUS
2010	EU25	CA	20	E
2010	BG	CA	2	P
2010	RO	CA	2	P

1842

ds2				
TIME	REF_AREA	PARTNER	OBS_VALUE	OBS_STATUS
2010	EU25	CA	10	P

1843  
1844

ds3				
TIME	REF_AREA	PARTNER	OBS_VALUE	OBS_STATUS
2010	EU25	CA	30	E

1845

the example above can be expressed equivalently as:

```
1847 ds_bop3 := {
1848     obs_value = ds1.obs_value + ds2.obs_value ,
1849     obs_status = ds1.obs_status
1850 }
```

1851

2) outer join returns data points that exist in at least one Dataset when a data point does not exist in the other Dataset, the value of its measures and components is null compare with the following example:

1852

1853

1854

```
1855 ds_bop3 := [outer ds1, ds2 ] {
1856     obs_value = ds1.obs_value + ds2.obs_value ,
1857     obs_status = ds1.obs_status
1858 }
```

1859

ds3				
TIME	REF_AREA	PARTNER	OBS_VALUE	OBS_STATUS
2010	EU25	CA	30	E
2010	BG	CA		P
2010	RO	CA		P

1860

3) nvl is used to replace the null value with 0 (compare with the previous example)

1861

1862

1863

1864

1865

ds3				
TIME	REF_AREA	PARTNER	OBS_VALUE	OBS_STATUS
2010	EU25	CA	30	E
2010	BG	CA	2	P
2010	RO	CA	2	P

1866

1867

1868

4) example of join defined on a subset of the Identifier Components (family\_id)

ds_census			
PERSON_ID	FAMILY_ID	REL	NATIONALITY
1	1	HEAD	IT
2	1	SPOUSE	IT
3	1	CHILD	IT

4	2	HEAD	US
5	2	SPOUSE	US
6	2	CHILD	IT
7	2	CHILD	IT

```

1869
1870 head := ds_census (rel=HEAD);
1871 spouse := ds_census (rel=SPOUSE);
1872 child := ds_census(rel= CHILD );
1873 [ head, spouse, child on family_id ] {
1874     rename head.person_id to head_id, spouse.person_id to spouse_id, child.person_id to child_id ;
1875     rename head. nationality to head_nationality, spouse. nationality to spouse_nationality, child. nationality to
1876     child_nationality ;
1877 }
1878

```

ds_result						
FAMILY_ID	HEAD_ID	SPOUSE_ID	CHILD_ID	HEAD_NATIO NALITY	SPOUSE_NAT IONALITY	CHILD_NATIO NALITY
1	1	2	3	IT	IT	IT
2	4	5	6	US	US	IT
2	4	5	7	US	US	IT

1879

### calc\_clause

1880

1881

calc\_clause := { role } compName = k

1882

1883

The calc\_clause adds a new component (Identifier , Measure or Attribute Component) or replaces an existing component (Measure or Attribute: the Identifier Components cannot be replaced) of the working Dataset. If calc\_comp coincides with the name of an existing Component in the working Dataset (even with different type), the calculated one replaces the former, in name, value and type.

1884

1885

1886

1887

1888

### Examples

1889

Suppose merge\_flags is a user defined function (not shown here) that returns EP when applied to E, P

1890

1891

```

ds3 := [ ds1, ds2 ] {
    obs_value = ds1.obs_value + ds2.obs_value ,
    obs_status = merge_flags ( ds1.obs_status, ds2.obs_status )
}

```

1892

1893

1894

1895

ds3				
TIME	REF_AREA	PARTNER	OBS_VALUE	OBS_STATUS
2010	EU25	CA	30	EP

1896

### drop\_clause

1897

1898

drop\_clause ::= **drop** { cmp { , cmp } \* }

1899

The **drop** clause drops from the working Dataset the measures and attributes specified.

1900

### Examples

1901

1902

```

ds2 := [ ds1 ] { drop obs_status }

```

ds1			
TIME	REF_AREA	PARTNER	OBS_VALUE
2010	EU25	CA	20
2010	BG	CA	2

2010	RO	CA	2
------	----	----	---

1903

**keep\_clause**

1904  
1905 `keep_clause ::= keep { cmp { , cmp } * }`

1906 The **keep** clause keeps in the working Dataset only the measures and attributes specified.

*Examples*

1907  
1908 `ds2 := [ ds1 ] { keep time, ref_area, partner, obs_value }`

ds1			
TIME	REF_AREA	PARTNER	OBS_VALUE
2010	EU25	CA	20
2010	BG	CA	2
2010	RO	CA	2

1909

**filter\_clause**

1910  
1911 `filter_clause ::= filter boolean-expression | dpr`

1912  
1913 When a boolean expression is specified, **filter** filters out the data points of the working Dataset for which the  
1914 boolean expression evaluates to false or null (i.e., only the data points for which the Boolean expression  
1915 evaluates to true are maintained).

1916 when a data point ruleset is specified, **filter** filters out the data points of the working Dataset for which at least  
1917 one antecedent condition evaluates to true and its corresponding consequent condition evaluates to false or null  
1918 (i.e., only the data points that satisfy the whole ruleset are maintained).

1919 Note that **null** as a result of a boolean expression is always interpreted as "not satisfied".

1920

*Examples*

1921  
1922 1) Compute new measure `obs_value_neg` derived from `obs_value`, rename `ds1.obs_status` to keep it in the result.  
1923 `ds1.obs_value` is not kept

1924 `ds_bop3 := [ outer ds1, ds2 ] {`  
1925 `filter ds2.obs_value <> 0 ,`  
1926 `obs_value = ds1.obs_value / ds2.obs_status ,`  
1927 `rename 'ds1.obs_status' to obs_status`  
1928 `}`

ds3				
TIME	REF_AREA	PARTNER	OBS_VALUE_NEG	OBS_STATUS
2010	EU25	CA	2	E

1929

2) Simple filter

1930  
1931  
1932 `ds2 := [ ds1 ] { filter obs_value < 10 and time = "2010" }`

ds2				
TIME	REF_AREA	PARTNER	OBS_VALUE	OBS_STATUS
2010	BG	CA	2	P
2010	RO	CA	2	P

1933

**rename\_clause**

1934  
1935 `rename_clause ::= rename cmp to cmp { , cmp to cmp }`

1936

1937 **rename** allows renaming one or more components (Identifier , Measure or Attribute Component). VTL verifies  
 1938 that the resulting Dataset, after renaming all the specified components, has unique names of its components  
 1939 (otherwise an error is raised). Renaming an Identifier Component implies that the actual values of it are valid for  
 1940 the dimension type (usually the code list associated to the Identifier).

1941

1942 *Examples*

1943 compute the measure obs\_value\_neg derived from obs\_value

1944 rename ds1.obs\_status to keep it in the result

1945 ds1.obs\_value is not kept

1946

1947 ds\_bop3 := [ **outer** ds1, ds2 ] {

1948     obs\_value\_neg = -ds1.obs\_value,

1949     **rename** 'ds1.obs\_status' to obs\_status                     equivalent to obs\_status = ds1.obs\_status

1950     }

ds3				
TIME	REF_AREA	PARTNER	OBS_VALUE_NEG	OBS_STATUS
2010	EU25	CA	-20	E
2010	BG	CA		P
2010	RO	CA		P

1951

1952 *unfold\_clause*

1953

1954     unfold\_clause ::= **unfold** dim , msr **to** elem { , elem }

1955

1956 **unfold** creates the resulting Dataset in the following way: drops the Identifier Component *dim* and the measure  
 1957 *msr* from the resulting Dataset, partitions the input Dataset by grouping the values of the remaining Identifiers of  
 1958 the Dataset, transposes the data points of each group into a single data point of the resulting Dataset and adds  
 1959 new measures elements (all *elem* in the list). Then in the newly created data point **unfold** assigns to the value of  
 1960 each measure *elem* the value of *msr* existing in the input Dataset where *dim* = *elem* (if such a data point exists) or  
 1961 **null** otherwise.

1962 The data points where "dim **not in** (*elem* , ...)" are removed from the resulting Dataset.

1963 Note that the attributes created may have names that are not syntactically correct (they may start with a digit,  
 1964 contain special characters, etc.): those names must be quoted (included in single quote " ' " ) in any expression,  
 1965 and it is not allowed to create a Dataset based on those data. It is also not allowed to return a Dataset as the final  
 1966 result of the join expression with names not complying with the VTL rules. Note that the names can be renamed  
 1967 using the **rename** operator.

1968

1969 *Examples*

1970 Unfold and fold Identifier ref\_area and measure obs\_value

1971 ds\_unfold := [ ds1 ] { **unfold** ref\_area, obs\_value **to** (EU25, BG, RO) }

1972

ds1				
TIME	REF_AREA	PARTNER	OBS_VALUE	OBS_STATUS
2010	EU25	CA	20	E
2010	BG	CA	2	P
2010	RO	CA	2	P

1973

1974

ds_unfold				
TIME	PARTNER	EU25	BG	RO
2010	CA	20	2	2



1975

1976 **fold\_clause**

1977

1978 `fold_clause ::= fold elem { , elem } to dim , msr`

1979

1980 **fold** transposes a single data point of the input Dataset into several data points. It adds Identifier dim and  
1981 measure msr to the resulting Dataset, inserts into the resulting Dataset a data point for each value A in the  
1982 element list and assigns to the inserted data point dim = A and msr = value of measure A in the input Dataset.  
1983 When measure A is null then **fold** does not create a data point for that measure.

1984 Note that in general unfolding and folding are not exactly symmetric operations, i.e. in some cases the fold  
1985 operation applied to the unfolded Dataset does not recreate exactly the original Dataset (before unfolding).

1986

1987 *Examples*

1988 `ds_fold := [ ds_unfold ] { fold (EU25, BG, RO ) to ref_area, obs_value }`

1989

ds_fold			
TIME	REF_AREA	PARTNER	OBS_VALUE
2010	EU25	CA	20
2010	BG	CA	2
2010	RO	CA	2

1990

1991

1992 **Function Creation**

1993 *Semantics*

1994 Creates a named function with given arguments, defined by a given expression.

1995

1996 *Syntax*

1997 `create function function-name ( arg-list )`

1998 `[ returns return-type ]`

1999 `as defining-expression`

2000

2001 *Parameters*

2002 `function-name` : <IDENT>

2003 `arg-list` : [ arg { , arg } ]

2004 `arg` : arg-name [ **as** arg-type ] [ := default-value ]

2005 `arg-name` : <IDENT>

2006 `arg-type` : type

2007 `default-value` : literal

2008 `defining-expression` : expression

2009 `return-type` : type

2010

2011 `function-name` – the name under which the function is created

2012 `arg-list` – the comma-separated list of formal arguments (can be empty)

2013 `arg-name` – the name of an individual argument

2014 `arg-type` – the optionally specified argument type

2015 `default-value` – the optionally specified argument default value; it can be a scalar literal (number, string, Boolean,  
2016 or date) or a function literal (an anonymous function)

2017 `return-type` – the optionally specified function return type

2018 `defining-expression` – the expression that defines the function

2019

2020 *Constraints*

2021

- Each *arg-name* must be unique within the *arg-list*. For each *arg-name*, element *arg-type* can be omitted if the argument type can be inferred from the definition. If both *arg-type* and *default-value* are given, then *default-value* must be compatible with *arg-type*. Arguments that have *default-value* must come at the end of *arg-list*.
- If *return-type* is omitted, the statically inferred type of *defining-expression* is used as an implicit *return-type*. If *return-type* is given, the inferred type of *defining-expression* must be compatible with *return-type*.

2030 *Returns*  
2031 Nothing

### 2033 *Semantic specification*

2034 The **create function** construct creates a named function with zero or more given arguments, defining  
2035 expression, and the return type. The function can be called by name followed by the sequence of comma-  
2036 separated call arguments in parentheses. Each call argument is an expression of type compatible with the  
2037 corresponding *arg-type*, whose result is passed by value. The named function call syntax is:

2038 *function-call* ::= function-name ( call-arg-list )

2039 *call-arg-list* ::= { call-arg { , call-arg } }

2040 *call-arg* ::= positional-arg | named-arg

2041 *positional-arg* ::= arg-value

2042 *named-arg* ::= *arg-name* := arg-value

2043 *arg-value* ::= expression

2044 In *call-arg-list*, positional arguments and named arguments cannot be arbitrarily mixed: named arguments must  
2045 come after all positional arguments (if any).

2046 For *function-call* to be valid, the following properties are statically checked:

2047 First, the *function-name* must refer to a function created with **create function**.

2048 Second, all arguments to *function-name* that do not have *default-value* must be supplied. Positional arguments  
2049 are supplied in the order in which they appear in the corresponding *arg-list*. The named arguments can be given  
2050 in any order after the positional arguments, but cannot refer to arguments whose values are already given by an  
2051 earlier *positional-arg* or *named-arg*.

2052 For each *arg-name*, the type of the provided *arg-value* must be compatible with the corresponding *arg-type*.

2053 Values for arguments with *default-value* must be specified using *named-arg*, nor *positional-arg*.

2054 The result of a *function-call* to a *function-name* defined using **create function** is the value of *defining-expression*  
2055 for the values of *arg-list* as supplied by the *call-arg-list*.

### 2057 *Examples*

2058 1)

2059 **create function** compare\_integer\_descending(x as integer, y as integer)

2060 **returns boolean**

2061 **as** x > y

2062 creates function *compare\_integer\_descending* which takes two integer arguments, *x* and *y*, and returns **true** if *x*>*y*,

2063 otherwise **false**. Call *compare\_integer\_descending*(1, 4) returns **false**, and call *compare\_integer\_descending*(8,0)

2064 returns **true**.

2065

2066 2)

2067 **define function** has\_solution(a, b, c)

2068 **as** b\*b-4\*a\*c>0

2069 creates function *has\_solution* takes three number arguments, *a*, *b* and *c*, and returns **true** only if the quadratic

2070 equation  $ax^2+bx+c=0$  has at least one solution. The types for *a*, *b* and *c* are inferred as *number* because in the

2071 defining expression the left-hand side of > must be a number to be comparable with the right-hand side 0. Also,

2072 *return-type* is inferred as *Boolean*, because that is the result type for the comparison operator > on scalars.

2073 Call *has\_solution*(1,0,0) returns **true**, and *has\_solution*(1,0,1) returns **false**. The latter is equivalent to

2074 *has\_solution*(a:=1, b:=0, c:=1), which is also equivalent to *has\_solution*(c:=1, a:=1, b:=0).

2075

2076

2077

## 2078 VTL-ML - String operators and functions

### 2079 length

#### 2080 *Semantics*

2081 The **length** operator returns the length of a character string.

2082

#### 2083 *Syntax*

2084 **length** ( *ds* )

2085

#### 2086 *Parameters*

2087 *ds* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as string-literal}+  
2088 {attribute <IDENT> as scalar-type}\* | string]

2089

2090 *ds* – is a Dataset expression or a string

2091

#### 2092 *Constraints*

2093 If *ds* is a scalar then it must be a **string** type.

2094 If *ds* is a Dataset then it must have at least a measure of type **string**.

2095

#### 2096 *Returns*

2097 If *ds* is a scalar then **length** returns a scalar **integer** representing the length of *ds*.

2098 If *ds* is a Dataset and has N string measure components, then **length** returns a Dataset having the Identifier

2099 Components of *ds* and N numeric Measures with the same name as the string Measures of *ds* and containing the  
2100 length of the corresponding measures.

2101

#### 2102 *Examples*

2103 On scalar

2104 A := **length** ( "Hello, World!" )                      A = 13

2105

2106 On Dataset

2107 ds\_r := **length**(ds\_1)

2108

ds_1		
K1	K2	M1
1	A	hello
2	B	null

2109

2110

ds_1		
K1	K2	M1
1	A	5
2	B	null

2111

2112 Note: the last value of M1 is null because the corresponding value of ds\_1 is null.

2113

### 2114 String concatenation ||

#### 2115 *Semantic*

2116 The operator || concatenates two strings.

2117

2118 *Syntax*

2119 *ds\_1 || ds\_2*

2120 *Parameters*

2121 *ds\_1, ds\_2* : [ dataset {identifier <IDENT> as scalar-type }+ {measure <IDENT> as string }+  
2122 {attribute <IDENT> as scalar }\* | string]

2123

2124 *ds\_1, ds\_2* – is a Dataset expression or a string

2125

2126 *Constraints*

- 2127 • If *ds\_1* (*ds\_2*) is a scalar then it must be a **string** data type.
- 2128 • If *ds\_1* (*ds\_2*) is a Dataset then it has at least a measure of **string** type.
- 2129 • If both *ds\_1* and *ds\_2* are Datasets then they must have at least one Identifier in common (with the same name and datatype).
- 2130 • If both *ds\_1* and *ds\_2* are Datasets then either they have one or more measures in common, or at least one of them has only a measure.

2133

2134 *Returns*

2135 The operator returns:

2136 If both *ds\_1* and *ds\_2* are scalar values then the || operator returns a scalar string value, the concatenation of *ds\_1*  
2137 and *ds\_2*.

2138 If either *ds\_1* or *ds\_2* is a Dataset then the || operator returns a Dataset having the following components:

- 2139 • The superset of the Identifier Components of *ds\_1* and *ds\_2*
- 2140 • If *ds\_1* and *ds\_2* have one or more string measures in common (i.e., with the same name) then the resulting  
2141 Dataset has these common string measures, with the same name, containing the concatenation of the  
2142 respective measures of *ds\_1* and *ds\_2*. Otherwise, if *ds\_1* and *ds\_2* do not have any measures in common and  
2143 have only one measure then the resulting Dataset contains a measure named **CONDITION** that contains the  
2144 concatenation of the single measures of *ds\_1* and *ds\_2*.

2145 The resulting Dataset contains a data point for each pair of data points of *ds\_1* and *ds\_2* that have the same key  
2146 (the same values of the Identifier Components).

2147

2148 *Examples*

2149 On scalar

2150 A := "Hello" || ", world! "

C = "Hello, world! "

2151

2152 On Dataset

2153 ds\_r := ds\_1 || ds\_2

ds_1		
K1	K2	M1
1	A	"hello"
2	B	"hi"

2154

2155

2156

ds_2		
K1	K2	M1
1	A	"world"
2	B	"there"

2157

2158

ds_r		
K1	K2	M1
1	A	"helloworld"
2	B	"hithere"

2159

## 2160 trim /rtrim/ltrim

### 2161 *Semantics*

2162 The **trim** /**rtrim**/**ltrim** operators eliminate trailing or/and leading whitespace from a string.

2163

### 2164 *Syntax*

2165 **[trim | rtrim | ltrim] ( ds )**

2166

### 2167 *Parameters*

2168 *ds* : [dataset {identifier <IDENT> as scalar-type }+ {measure <IDENT> as string }+  
2169 {attribute <IDENT> as scalar-type }\* | string]

2170

2171 *ds* – is a Dataset expression or a string

2172

### 2173 *Constraints*

2174 If *ds* is a scalar then it must be a **string** data type.

2175 If *ds* is a Dataset then it must have at least a measure of **string** data type.

2176

### 2177 *Returns*

2178 If *ds* is a scalar then operators returns a scalar string representing the input string without trailing or/and  
2179 leading whitespace.

2180 If *ds* is a Dataset and has N string measures then operators returns a Dataset having the Identifier Components of  
2181 *ds* and N string measures with the same name as the string measures of *ds* where the values take the value of the  
2182 input ones without whitespaces from left and right (trim), or alternatively without the left (ltrim) or right (rtrim)  
2183 whitespaces.

2184

### 2185 *Semantic specification*

2186 The operators trim whitespaces from left and right of it (trim), or alternatively only the left (ltrim) or right  
2187 (rtrim) whitespaces.

2188

### 2189 *Examples*

2190 example on scalar

2191 If A = " Hello, world! ":

2192 B := **trim**(A)                                    B = "Hello, world!"

2193

2194 example on Dataset

2195 ds\_1 := **trim**(ds)

2196

2197

ds		
K1	K2	M1
1	A	" hello world "
2	B	"hi "
3	C	" help! "

2198

ds_1		
K1	K2	M1
1	A	"hello world"
2	B	"hi"
3	C	"help!"

2199

2200

## 2201 upper/lower

### 2202 *Semantics*

2203 The **upper/lower** operators convert all characters of a string to upper / lower case.

### 2204 *Syntax*

2205 [**upper** | **lower**] ( *ds* )

2206

### 2207 *Parameters*

2208 *ds* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as string }+  
2209 {attribute <IDENT> as scalar-type}\*] string]

2210

2211 *ds* – is a Dataset expression or a string

2212

### 2213 *Constraints*

2214 If *ds* is a scalar string then it must be a **string** data type.

2215 If *ds* is a Dataset then it must have at least a measure of **string** data type.

2216

### 2217 *Returns*

2218 If *ds* is a scalar then operators returns a scalar string that is the upper case or the lower case of the input one.

2219 If *ds* is a Dataset and has N string measures, then the operators return a Dataset having the Identifier Components  
2220 of *ds* and N string measures with the same name as the string measures of *ds* where the Measure Components  
2221 assumes the upper case or lower case values of the respective input value of the Measure Components’.

2222

### 2223 *Examples*

2224 On scalar

2225 1) If A = "Hello, World!":

2226 B := **upper**(A)

2227 B := **lower**(A)

2228

B = "HELLO, WORLD!"

B = "hello, world!"

2229 On Dataset

2230 2) *ds\_r* := **upper**(*ds\_1*)

2231

ds_1		
K1	K2	M1
1	A	"hello world"
2	B	"hi"
3	C	"help"

2232

ds_r		
K1	K2	M1
1	A	"HELLO WORLD"
2	B	"HI"
3	C	"HELP"

2233

## 2234 substr

### 2235 *Semantics*

2236 The operator **substr** extracts a substring from a string

2237

### 2238 *Syntax*

2239 **substr** ( *ds*, {, *startPosition*} {, *length*} )

2240

### 2241 *Parameters*

2242 *ds*: [dataset {identifier <IDENT> as scalar-type }+ {measure <IDENT> as string }+  
 2243 {attribute <IDENT> as scalar-type }\*|string]  
 2244 *startPosition* : integer  
 2245 *length* : integer

- 2246 • *ds* – is the input Dataset or the input string.
- 2247 • *startPosition* – is the index of the character in the string from which the substring is performed.
- 2248 • *length* – is the number of the characters in the string to be taken starting from *startPosition*.

2250 **Constraints**

- 2251 • *startPosition* must be major or equal than 0 and minor than the whole length of the input string.
- 2252 • *startPosition* plus *length* must be minor than the whole length of the input string, otherwise the *length* parameter is ignored.
- 2253 • If *ds* is a scalar then it must be a **string** data type.
- 2254 • If *ds* is a Dataset then it must have at least a measure of **string** data type.

2255 **Returns**

2256 If *ds* is a scalar string then operators returns a substring of the input one starting from *startPosition* and extracting *length* characters.  
 2257 If *ds* is a Dataset and has N string measures then operators returns a Dataset having the Identifier Components of *ds* and N string measures with the same name of the string measures of *ds* where the Measure Components assumes substring values of the respective input Measure Components's values, obtained starting from *startParameters* and taking *length* characters.

2265 **Semantics**

2266 The substring of the input string is obtained stating from *startPosition* and extracting *length* characters, if *length* plus *startPosition* is greater than the whole length of the input string, then *length* parameter is ignored.

2269 **Examples**

2270 On scalar

2271 1) Assuming that A = "Hello, world!":  
 2272           B := **substr**(A, 2)                                B = "lo, world!"  
 2273           B := **substr**(A, 2, 5)                            B = "lo, w"  
 2274           B := **substr**(A, 0, 4)                           B = "Hell"

2275 On Dataset

2276 2) *ds\_r* := **substr**(*ds\_1*,7)

2280

ds_1		
K1	K2	M1
1	A	"hello world"

ds_r		
K1	K2	M1
1	A	"rld"

2281 3) *ds\_r* := **substr**(*ds\_1*,0,5)

2282

ds_1		
K1	K2	M1
1	A	"hello world"

2283

ds_r		
K1	K2	M1

1	A	"hello"	2284 2285
---	---	---------	--------------

2286

## 2287 instr

### 2288 *Semantics*

2289 The **instr** operator returns the position of a string in another one

2290

### 2291 *Syntax*

2292 `instr (ds, strToSearch {, startPosition} {, occurrence} )`

2293

### 2294 *Parameters*

2295 `ds` : [dataset {identifier <IDENT> as scalar-type }+ {measure <IDENT> as string }+  
2296 {attribute <IDENT> as scalar-type }\*|string]

2297 `strToSearch` : string

2298 `startPosition` : integer

2299 `occurrence` : integer

2300

- 2301 • `ds` – is the input string or the input Dataset.
- 2302 • `strToSearch` – is the string to search.
- 2303 • `startPosition` – is the index of the character in the string from which start to search.
- 2304 • `occurrence` – is the number of occurrences of the `strToSearch` from which start to search

2305

### 2306 *Constraints*

- 2307 • If `ds` is a scalar then it must be a **string** data type.
- 2308 • If `ds` is a Dataset then it must have at least a measure of **string** data type.

2309

### 2310 *Returns*

2311 If `ds` is scalar then the operator returns the position of the first character of `strToSearch` in the string. The  
2312 `startPosition` and `occurrence` are integers indicating the character of string and the number of occurrences from  
2313 which start to search, respectively.

2314 If `ds` is a Dataset and has N string measures then operators returns a Dataset having the IdentifierComponents of  
2315 `ds` and N string measures with the same name of the string measures of `ds` where the Measure Components are  
2316 integer representing the first character of `strToSearch` in the string. The `startPosition` and `occurrence` are integer  
2317 indicating respectively the character of string and the number of occurrences from which start to search.  
2318 A negative value of `startPosition` counts backward from the end of string.

2319

### 2320 *Semantic specification*

2321 If the string to search is not present in `str`, then the value returned is -1. If `startPosition` is omitted the start  
2322 position is 1, if `occurrence` is omitted the value is 1.

2323

### 2324 *Examples*

2325 On scalar

2326 1) Assuming that A = "abcde":

2327        B := **instr** (A, "c" )                                B = 2

2328

2329 On Dataset

2330 2) `ds_2 := instr(ds_1,"hello")`

2331

ds_1		
K1	K2	M1
1	A	"hello world"
2	A	"say hello"
3	A	"he"
4	A	"hi, hello!"



2332

ds_2		
K1	K2	M1
1	A	0
2	A	4
3	A	-1
4	A	4

2333

## 2334 date\_from\_string

### 2335 *Semantics*

2336 The operator **date\_from\_string** converts a string into a date.

2337

### 2338 *Syntax*

2339 **date\_from\_string**( *ds*, *format* )

2340

### 2341 *Parameters*

2342 *ds* : [dataset {identifier <IDENT> as scalar-type }+ {measure <IDENT> as string }+  
2343 {attribute <IDENT> as scalar-type }\* |string]

2344 *Format* : string

2345

- 2346 • *ds* – is the input string or the input Dataset
- 2347 • *format* – is the format of the resulting date.

2348

### 2349 *Constraints*

- 2350 • If *ds* is a scalar then it must be a **string** data type.
- 2351 • If *ds* is a dataset then it must have at least a measure of **string** data type.
- 2352 • *format* must respect one of these patterns:

2353

Format	Frequency	Example	Frequency
YYYY		2000	Annual
YYYYSN	S	2000S1	Semestrial
YYYYQN	Q	2000Q1	Quarterly
YYYYMNN	M	2000M01	Monthly
YYYYDNNNN	D	2000D0101	Daily
YYYYA	A	2000A	Annual
YYYYSN	S	2000S1	Semestrial
YYYY-QN	Q	2000-Q1	Quarterly
YYYY-NN	M	2000-01	Monthly
YYYY-NN-NN	D, M, Q or A	2000-01-01	Daily, Monthly, Quarterly or Annual

2354

2355

2356

### 2357 *Returns*

2358 If *ds* is a scalar, the operator returns its date representation, based on the chosen format.

2359 If *ds* is a Dataset having N string Measure Components, the operator returns a Dataset having the same Identifier Components as *ds* and N Measure Components varying in type (from string-literal to date) assuming values of the date representations (on the base of the *format*) of the dates in the input Measure Components.

2361

2362

2363 *Examples*  
 2364 On scalar  
 2365 1) If A = "2016-02"  
 2366        B := **date\_from\_string** (A, YYYY-MM)                    B = 2016-02-01  
 2367 2) If A = "2016-02"  
 2368        B := **date\_from\_string** (A, YYYY-MM-DD)                B = 2016-02-01  
 2369 A date component has always years, months and days.

2370  
 2371 On Dataset  
 2372 3) ds\_2:= **date\_from\_string** (ds\_1, "YYYY-MM")

ds_1		
K1	K2	M1
1	A	"2015-12"
2	B	"2015-06"
3	C	"2015-12"
4	E	"2015-06"

2375  
 2376

ds_2		
K1	K2	M1
1	A	2015-12-01
2	B	2015-06-01
3	C	2015-12-01
4	E	2015-06-01

2377

## 2378 **replace**

2379 *Semantics*  
 2380 The **replace** operator replaces a substring with a given string.

2381 *Syntax*  
 2382 **replace**( ds, str\_old {, str\_new })

2383 *Parameters*  
 2384 ds : [dataset {identifier <IDENT> as scalar-type }+ {measure <IDENT> }+  
 2385        {attribute <IDENT> as scalar-type }\*|string]  
 2386 str\_old, str\_new : string

- 2387 • ds – is the input string or the input Dataset,
- 2388 • str\_old – is the string to be replaced,
- 2389 • str\_new – is the string to replace. If omitted then all occurrences of str\_old are removed.

2390 *Constraints*  
 2391 • If ds is a scalar then it must be a **string** data type.  
 2392 • If ds is a Dataset then it must have at least a measure of **string** data type.

2393 *Returns*  
 2394 • If ds is a scalar, the operator returns a string having the ds value obtained replacing str\_old with str\_new.  
 2395 • If ds is a Dataset having N string Measure Components, returns a Dataset having the Identifier Component of ds and N string Measure Components obtained replacing str\_old with str\_new.

2402

2403 *Examples*

2404 On scalar

2405 1) If A = "Hello"

2406 B := **replace** (A,"ello","i")

B = "Hi"

2407

2408 On Dataset

2409 2) ds\_2:= **replace** (ds\_1,"ello","i")

2410

ds_1		
K1	K2	M1
1	A	"hello world"
2	A	"say hello"
3	A	"he"
4	A	"hello!"

2411

2412

ds_2		
K1	K2	M1
1	A	"hi world"
2	A	"say hi"
3	A	"he"
4	A	"hi!"

2413

2414

## VTL-ML - Numeric operators and functions

### 2415 unary plus +

#### 2416 *Semantics*

2417 The + operator leaves the sign unaltered.

2418

#### 2419 *Syntax*

2420 + *ds*

2421

#### 2422 *Parameters*

2423 *ds* : [ dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as number}+

2424 {measure <IDENT> as scalar-type}\* {attribute <IDENT> as scalar-type}\*|number]

2425

2426 *ds* – is the input scalar number or the input Dataset.

2427

#### 2428 *Constraints*

2429 If *ds* is a scalar then it must be a **numeric** data type.

2430 If *ds* is a Dataset then it must have at least a measure of **numeric** data type.

2431

#### 2432 *Returns*

2433 If *ds* is a scalar then operator return the input number without altering its sign.

2434 If *ds* is a Dataset and has N numeric measures then operator return a Dataset having the Identifier Components of

2435 *ds* and N numeric measures without alterations.

2436

#### 2437 *Examples*

2438 On scalar

2439 1) A := +B

2440 if B = 5, then A = 5

2441

### 2442 unary minus -

#### 2443 *Semantics*

2444 The - operator inverts the sign

2445

#### 2446 *Syntax*

2447 - *ds*

2448

#### 2449 *Parameters*

2450 *ds* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as number}+

2451 {measure <IDENT> as scalar-type}\* {attribute <IDENT> as scalar-type}\*|number]

2452

2453 *ds* – is the input scalar number or the input Dataset.

2454

#### 2455 *Constraints*

2456 If *ds* is a scalar then it must be a **numeric** data type.

2457 If *ds* is a Dataset then it must have at least a measure of **numeric** data type.

2458

#### 2459 *Returns*

2460 If *ds* is a scalar then operator return the input number negated.

2461 If *ds* is a Dataset and has N numeric measures then operator return a Dataset having the Identifier Components of

2462 *ds* and N numeric measures with the sign of the values in the numeric Measure Components inverted.

2463

#### 2464 *Examples*

2465 On scalar

2466 1)  $A := -B$   
 2467       if  $B = 5$ , then  $A = -5$   
 2468       if  $B = -7$ , then  $A = 7$

2469 On Dataset  
 2471 2)  $ds_2 := - ds_1$   
 2472

ds_1		
K1	K2	M1
1	A	11
2	B	-14
3	C	9

2473  
 2474

ds_1		
K1	K2	M1
1	A	-11
2	B	14
3	C	-9

2475  
 2476 3)  $ds_2 := - ds_1$   
 2477

ds_1				
K1	K2	M1	M2	M3
1	A	11	12	"A"
2	B	-14	13	"B"
3	C	9	14	"C"

2478  
 2479  
 2480

ds_1				
K1	K2	M1	M2	M3
1	A	-11	-12	"A"
2	B	14	-13	"B"
3	C	-9	-14	"C"

2481  
 2482

## 2483 addition and subtraction + -

2484 *Semantics*  
 2485 The operator + or - compute the sum or subtraction

2486  
 2487 *Syntax*  
 2488  $ds_1 [ + | - ] ds_2$

2489  
 2490 *Parameters*

2491  $ds_1, ds_2$  : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as number}+  
 2492 {measure <IDENT> as scalar-type}\* {attribute <IDENT> as scalar-type}\*|number]

- 2493 •  $ds_1$ – is the first input scalar number or the first input Dataset.
- 2494 •  $ds_2$  – is the second input scalar number or the second input Dataset.

2495  
 2496 *Constraints*

- 2497 • If  $ds_1$  ( $ds_2$ ) is a scalar then it must be a **numeric** data type.
- 2498 • If  $ds_1$  ( $ds_2$ ) is a Dataset then it has at least a measure of **numeric** type.
- 2499 • If both  $ds_1$  and  $ds_2$  are Datasets then they must have at least one Identifier Component in common (with  
 2500 the same name and data type).
- 2501 • If both  $ds_1$  and  $ds_2$  are Datasets then either they have one or more measures in common, or at least one of  
 2502 them has only a measure.

2503  
 2504 *Returns*

2505 If both  $ds_1$  and  $ds_2$  are scalar values then the operators return the algebraic sum or subtraction of  $ds_1$  and  
 2506  $ds_2$ .

2507 If either  $ds_1$  or  $ds_2$  is a Dataset then the operators return a Dataset having the following components:

- 2508 • The superset of the Identifier Components of  $ds_1$  and  $ds_2$
- 2509 • If  $ds_1$  and  $ds_2$  have one or more numeric measures in common (i.e., with the same name) then the resulting  
 2510 Dataset has these common string measures, with the same name, containing the algebraic sum or subtraction  
 2511 of the respective measures of  $ds_1$  and  $ds_2$ . Otherwise, if  $ds_1$  and  $ds_2$  do not have any measures in  
 2512 common and have only one measure then the resulting Dataset contains only a measure named **CONDITION**  
 2513 that contains the algebraic sum or subtraction of the single measures of  $ds_1$  and  $ds_2$ .

2514 The resulting Dataset contains a data point for each pair of data points of  $ds_1$  and  $ds_2$  that have the same values  
 2515 on the common Identifier Components).

2516  
 2517 *Semantic specification*

2518 See also the operator **listsum** than returns a data point for those data points that would be ignored.

2519  
 2520 *Examples*

2521 example on Dataset  
 2522 1)

2523 In this example, we calculate the total population of a set of countries given two Datasets: one of the male  
 2524 population, and another, of the female population. They contain one measure each. Thus, the result will contain a  
 2525 single measure with the results of the addition.

2526  
 2527  $ds_3 := ds_1 + ds_2$   
 2528

ds_1		
TIME	GEO	POPULATION
2013	Belgium	5
2013	Denmark	2
2013	France	3
2013	Spain	4

2529  
 2530

ds_2			
TIME	GEO	AGE	POPULATION
2013	Belgium	Total	10
2013	Greece	Total	11
2013	Belgium	Y15-24	NULL
2013	Greece	Y15-24	2
2013	Spain	Y15-24	6

2531  
2532

ds_3			
TIME	GEO	AGE	POPULATION
2013	Belgium	Total	15
2013	Belgium	Y15-24	NULL
2013	Spain	Y15-24	10

2533  
2534  
2535  
2536  
2537  
2538

Note that the Data Points of ds\_1 and ds\_2 that has a missing in the other Dataset are not shown in the resulting one.

2) ds\_bop1 := ds\_bop1 + 1

ds_bop1				
TIME	REF_AREA	PARTNER	OBS_VALUE	OBS_STATUS
2010	EU25	CA	20	D
2010	BG	CA	2	P
2010	RO	CA	2	P

2539  
2540

ds_bop1				
TIME	REF_AREA	PARTNER	OBS_VALUE	OBS_STATUS
2010	EU25	CA	21	D
2010	BG	CA	3	P
2010	RO	CA	3	P

2541  
2542  
2543

3) ds\_plus := ds\_bop1 + ds\_bop2

ds_bop1				
TIME	REF_AREA	PARTNER	OBS_VALUE	OBS_STATUS
2010	EU25	CA	20	D
2010	BG	CA	2	P
2010	RO	CA	2	P

2544

ds_bop2				
TIME	REF_AREA	PARTNER	OBS_VALUE	OBS_STATUS
2010	EU25	CA	10	D

2545  
2546

ds_plus			
TIME	REF_AREA	PARTNER	OBS_VALUE
2010	EU25	CA	30

2547

## 2548 multiplication and division \* /

### 2549 *Semantics*

2550 The operator \* or / multiply or divide two numbers.

2551

### 2552 *Syntax*

2553  $ds_1$  [ \* | / ]  $ds_2$

2554

### 2555 *Parameters*

2556  $ds_1, ds_2$  : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as number}+  
2557 {measure <IDENT> as scalar-type}\* {attribute <IDENT> as scalar-type}\*|number]

- 2558 •  $ds_1$  - is the first input scalar number or the first input Dataset.
- 2559 •  $ds_2$  - is the second input scalar number or the second input Dataset.

2560

### 2561 *Constraints*

- 2562 • If  $ds_1$  ( $ds_2$ ) is a scalar then it must be a **numeric** data type.
- 2563 • If  $ds_1$  ( $ds_2$ ) is a Dataset then it has at least a measure of **numeric** type.
- 2564 • If both  $ds_1$  and  $ds_2$  are Datasets then they must have at least a dimension in common (with the same name and data type).
- 2566 • If both  $ds_1$  and  $ds_2$  are Datasets then either they have one or more measures in common, or at least one of them has only a measure.

2567

### 2568 *Returns*

2569 The operators return:

2570 If both  $ds_1$  and  $ds_2$  are scalar values then the operators return the algebraic product or ratio of  $ds_1$  and  $ds_2$ .

2571 If either  $ds_1$  or  $ds_2$  is a Dataset then the operators return a Dataset having the following components:

- 2572 • The superset of the Identifier Components of  $ds_1$  and  $ds_2$
- 2573 • If  $ds_1$  and  $ds_2$  have one or more numeric measures in common (i.e., with the same name) then the resulting Dataset has these common numeric measures, with the same name, containing the algebraic sum or subtraction of the respective measures of  $ds_1$  and  $ds_2$ . Otherwise, if  $ds_1$  and  $ds_2$  do not have any measures in common and have only one measure then the resulting Dataset contains a measure named **CONDITION** that contains the algebraic product or ratio of the single measures of  $ds_1$  and  $ds_2$ .

2578 The resulting Dataset contains a data point for each pair of data points of  $ds_1$  and  $ds_2$  that have the same key (the same values of the Identifier Components).

2580 The algebraic product of the input numbers or the ratio between them.

2581

### 2582 *Semantic specification*

2583 Division by zero results in a runtime exception.

2584

### 2585 *Examples*

2586

2587 On Dataset

2588 DSr:=total\_population[**rename** POPULATION as PERCENTAGE].PERCENTAGE \*

2589 Overcrowding\_rate\_urbanization.PERCENTAGE

2590

total_population				
TIME	GEO	AGE	POPULATION	UNEMPLOYMENT_RATE
2012	Belgium	Total	100	7.6
2012	Greece	Total	10	24.3
2012	Spain	Total	20	25
2012	Belgium	Y15-24	30	3.6
2012	Greece	Y15-24	5	18.3
2012	Switzerland	Y15-24	2	20



2591

Overcrowding_rate_urbanization		
TIME	GEO	PERCENTAGE
2012	Belgium	0.01
2012	Greece	0.1
2012	Spain	0.2
2012	Malta	0.3
2012	Finland	0.4
2012	France	0.5

2592

DSr			
TIME	GEO	AGE	PERCENTAGE
2012	Belgium	Total	1
2012	Greece	Total	1
2012	Spain	Total	4
2012	Greece	Y15-24	0.5

2593

2594

2595

## round/ceil/floor

2596

### Semantics

2597

The operators **round/ceil/floor** round a number.

2598

2599

### Syntax

2600

**[round(ds, decimals) | ceil(ds) | floor(ds)]**

2601

2602

### Parameters

2603

*ds*: [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as number}+  
{attribute <IDENT> as scalar-type}\*|number]

2604

2605

*decimals*: integer

2606

- *ds* – is the input scalar number or the input Dataset.

2607

- *decimals* – the decimal position to round to.

2608

2609

### Constraints

2610

If *ds* is a scalar then it must be a **numeric** data type.

2611

If *ds* is a Dataset then it must have at least a measure of **numeric** data type.

2612

*decimals* – must be an **integer** greater or equal than zero.

2613

2614

### Returns

2615

If *ds* is a scalar then the operators return the input number rounded using round, ceil or floor operator.

2616

If *ds* is a Dataset and has N numeric measures then the operators return a Dataset having the Identifier

2617

Components of *ds* and N numeric measures rounded using round, ceil or floor operator.

2618

2619

### Semantic specification

2620

The operator **round** takes as input a number and a number of decimal digits and rounds the former number to the number of decimal digits specified by the latter.

2621

The operator **floor** rounds to the largest previous integer, while ceil rounds to the smallest greater integer.

2622

2623

2624

### Examples

2625

On scalar

2626 1) If  $P = 3.14159$   
 2627        A := **round**(P, 2)                    A = 3.14  
 2628        B := **round**(P, 4)                    B = 3.1416  
 2629        C := **floor**(P)                        C = 3  
 2630        D := **floor**(P)                        D = 4

2631 On Dataset  
 2632  
 2633

unemployment					
AGE	TIME	GEO	SEX	YOUTH_UNEMPLOYMENT	UNEMPLOYMENT
From 20 to 29 years	2011	Germany	Total	7.5	5.9
From 20 to 29 years	2012	Germany	Total	7.1	5.5
From 20 to 29 years	2011	Greece	Total	33.7	17.7
From 20 to 29 years	2012	Greece	Total	42.5	24.3

2634  
 2635 2) ds\_1 := **round**(unemployment,0)  
 2636  
 2637

ds_1					
AGE	TIME	GEO	SEX	YOUTH_UNEMPLOYMENT	UNEMPLOYMENT
From 20 to 29 years	2011	Germany	Total	8	6
From 20 to 29 years	2012	Germany	Total	7	6
From 20 to 29 years	2011	Greece	Total	34	18
From 20 to 29 years	2012	Greece	Total	43	24

2638 3) ds\_1 := **round**(unemployment.YOUTH\_UNEMPLOYMENT, 0)  
 2639  
 2640

ds_1					
AGE	TIME	GEO	SEX	YOUTH_UNEMPLOYMENT	UNEMPLOYMENT
From 20 to 29 years	2011	Germany	Total	8	5.9
From 20 to 29 years	2012	Germany	Total	7	5.5
From 20 to 29 years	2011	Greece	Total	34	17.7
From 20 to 29 years	2012	Greece	Total	43	24.3

2641 4) ds\_1 := **ceil**(unemployment.YOUTH\_UNEMPLOYMENT)  
 2642  
 2643

ds_1					
AGE	TIME	GEO	SEX	YOUTH_UNEMPLOYMENT	UNEMPLOYMENT
From 20 to 29 years	2011	Germany	Total	8	6
From 20 to 29 years	2012	Germany	Total	7	6
From 20 to 29 years	2011	Greece	Total	34	18
From 20 to 29 years	2012	Greece	Total	43	25

2644  
  
 2645 **abs**

2646 *Semantics*  
 2647 The operator **abs** calculates the absolute value of a number

2648  
2649  
2650  
2651

*Syntax*

**abs(ds)**

2652

*Parameters*

2653 *ds* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as number}+  
2654 {attribute <IDENT> as scalar-type}\* |number]

2655 *ds* – is the input scalar number or the input Dataset.

2656

*Constraints*

2658 If *ds* is a scalar then it must be a **numeric** data type.

2659 If *ds* is a Dataset then it must have at least a measure of **numeric** data type.

2660

*Returns*

2662 If *ds* is a scalar then the operator returns the absolute value of the input number.

2663 If *ds* is a Dataset and has N numeric measures then the operator returns a Dataset having the Identifier

2664 Components of *ds* and the N numeric measures containing the absolute values of the corresponding ones in *ds*.

2665

*Examples*

2667 On scalar

2668 1) Let us assume A = -5:

2669 B := **abs(A)** B = 5

2670 C := **abs(B)** C = 5

2671

2672 On Dataset

2673 2) DatasetB := **abs(DatasetA)**

2674

Dataset A			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	0.484183
FR	Females	2011	-0.515817
FR	Total	2011	-1.000000

2675

2676

Dataset B			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	0.484183
FR	Females	2011	0.515817
FR	Total	2011	1.000000

2677

2678

2679

3) ds\_bop1 := **abs** ( ds\_bop1 )

ds_bop1				
TIME	REF_AREA	PARTNER	OBS_VALUE	OBS_STATUS
2010	EU25	CA	20	D
2010	BG	CA	2	P
2010	RO	CA	-2	P

2680

2681

ds_bop1				
TIME	REF_AREA	PARTNER	OBS_VALUE	OBS_STATUS
2010	EU25	CA	20	D
2010	BG	CA	2	P
2010	RO	CA	2	P

2010	EU25	CA	20	D
2010	BG	CA	2	P
2010	RO	CA	2	P

2682

2683 **trunc**

2684 *Semantics*

2685 The operator **trunc** truncates the decimal digits of a number.

2686

2687 *Syntax*

2688 **trunc**(*ds*, *decimals*)

2689 *Parameters*

2690 *ds* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as number}+  
 2691 {attribute <IDENT> as scalar-type}\* |number]

2692 *decimals* : integer

2693

- 2694 • *ds* – is the input scalar number or the input Dataset.
- 2695 • *decimals* – the decimal position beyond which the decimal digits are discarded.

2696

2697 *Constraints*

2698 If *ds* is a scalar then it must be a **numeric** data type.

2699 If *ds* is a Dataset then it must have at least a measure of **numeric** data type.

2700 *decimals* must be greater or equal than zero.

2701

2702 *Returns*

2703 If *ds* is a scalar then the operator returns the input number with the decimal digits discarded beyond the number  
 2704 of digits specified by *decimals*.

2705 If *ds* is a Dataset and has N numeric measures then the operator returns a Dataset having the Identifier  
 2706 Components of *ds* and the N numeric measures obtained by discarding the decimal digits after the *decimals*  
 2707 position.

2708

2709 *Examples*

2710 On scalar

2711 1) If P = 3.14159

2712       A := **trunc**(P, 2)                               A = 3.14  
 2713       B := **trunc**(P, 4)                               B = 3.1415

2714

2715 Differ from round:

2716       B := **round**(P, 4)                               B = 3.1416

2717

2718 On Dataset

2719 2) ds\_1 := **trunc**(DatasetA, 2)

2720

DatasetA			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	0.484183
FR	Females	2011	0.515817
FR	Total	2011	1.000000

2721

ds_1			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	0.48

FR	Females	2011	0.51
FR	Total	2011	1.00

2722

2723 **exp**

2724 *Semantics*

2725 The **exp** operator calculates the exponential of a number

2726

2727 *Syntax*

2728 **exp(ds)**

2729

2730 *Parameters*

2731 *ds* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as number}  
 2732 {measure <IDENT> as scalar-type}\* {attribute <IDENT> as scalar-type}\* |number]

2733

2734 *ds* – is the input scalar number or the input Dataset.

2735

2736 *Constraints*

2737 If *ds* is a scalar then it must be a **numeric** data type.

2738 If *ds* is a Dataset then it must have at least a measure of **numeric** data type.

2739

2740 *Returns*

2741 If *ds* is a scalar then the operator returns *e* (Napier’s – or Euler’s – constant) raised to *ds*.

2742 If *ds* is a Dataset and has N numeric measures then the operator returns a Dataset having the Identifier  
 2743 components of *ds* and the N numeric measures obtained by elevating *e* (Nepero’s number) to the value in the  
 2744 original Measure Component.

2745

2746 *Examples*

2747 On scalar

2748 1) If B = 5:

2749 A := **exp(B)** A = 148.413

2750 2) If B = -1:

2751 A := **exp(B)** A = 0.368

2752 3) If B = 0:

2753 A := **exp(B)** A = 1.0

2754

2755 On Dataset

2756

2757 4) DatasetB := **exp(DatasetA)**

Dataset A			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	5
FR	Females	2011	8
FR	Total	2011	2

2758

2759

Dataset B			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	148.41
FR	Females	2011	2980.95
FR	Total	2011	7.389

2760

2761

## 2762 ln

### 2763 Semantics

2764 The operator **ln** calculates the natural logarithm of a number

2765

### 2766 Syntax

2767 **ln(ds)**

2768

### 2769 Parameters

2770 *ds* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as number}+  
2771 {attribute <IDENT> as scalar-type}\* | number]

2772

2773 *ds* – is the input number.

2774

### 2775 Constraints

2776 If *ds* is a scalar then it must be a **numeric** data type greater than zero.

2777 If *ds* is a Dataset then it must have at least a measure of **numeric** data type.

2778

### 2779 Returns

2780 If *ds* is a scalar then the operator returns the natural logarithm (base *e*) of *ds*.

2781 If *ds* is a Dataset and has N numeric measures then the operator returns a Dataset having the Identifier

2782 Components of *ds* and the N numeric measures obtained by calculating the natural logarithm (in base *e*, Nepero's  
2783 number) of the value in the original Measure Component.

2784

2785 The logarithm of a zero or negative number results in a runtime exception.

2786

### 2787 Examples

2788 On scalar

2789 1) If B = 1:

2790       A := **ln(B)**                               A = 0

2791 2) If B = 148:

2792       A := **ln(B)**                               A = 4.997

2793

2794 On Dataset

2795

2796 3) DatasetB := **ln(DatasetA)**

2797

Dataset A			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	148.41
FR	Females	2011	2980.95
FR	Total	2011	7.389

2798

Dataset B			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	5
FR	Females	2011	8
FR	Total	2011	2

2799

2800

## 2801 log

### 2802 *Semantics*

2803 The **log** operator calculates the logarithm of a number to a base b

2804

### 2805 *Syntax*

2806 **log(ds, base)**

2807

### 2808 *Parameters*

2809 *ds* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as number}+  
2810 {attribute <IDENT> as scalar-type}\* |number]

2811 *base* : integer

2812

- 2813 • *ds* – is the input scalar number (greater than zero) or the input Dataset.
- 2814 • *base* – the base of the logarithm.

2815

### 2816 *Constraints*

- 2817 • If *ds* is a scalar then it must be a **numeric** data type greater than zero.
- 2818 • If *ds* is a Dataset then it must have at least a measure of **numeric** data type having values greater than zero.
- 2819 • *base* must be greater than zero.

2820

### 2821 *Returns*

2822 If *ds* is a scalar then the operator returns the *base* logarithm of *ds*.

2823 If *ds* is a Dataset and has N numeric measures then the operator returns a Dataset having the Identifier

2824 Components of *ds* and the N numeric Measures are obtained by calculating the logarithm in *base* of the value in  
2825 the original Measure Component.

2826

2827 The logarithm of a zero or negative number results in a runtime exception.

2828

### 2829 *Examples*

2830 On scalar

2831 1) If B = 1024:

2832           A := **log**(B, 2)                                   A = 10

2833           A := **log**(B, 10)                                  A = 3.01

2834

2835 On Dataset

2836 2) DatasetB := **log**(2, DatasetA)

Dataset A			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	1024
FR	Females	2011	64
FR	Total	2011	32

2837

Dataset B			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	10
FR	Females	2011	6
FR	Total	2011	5

2838

## 2839 power

### 2840 *Semantics*

2841 The operator **power** calculates the power of a number raised to an exponent

2842  
 2843  
 2844  
 2845  
 2846  
 2847  
 2848  
 2849  
 2850  
 2851  
 2852  
 2853  
 2854  
 2855  
 2856  
 2857  
 2858  
 2859  
 2860  
 2861  
 2862  
 2863  
 2864  
 2865  
 2866  
 2867  
 2868  
 2869  
 2870  
 2871  
 2872

*Syntax*

**power(ds, exponent)**

*Parameters*

ds : [ dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as number}+  
 {attribute <IDENT> as scalar-type}\* | number]

exponent : integer

- ds – is the input scalar number or the input Dataset.
- exponent – is the exponent of the power.

*Constraints*

- If ds is a scalar then it must be a **numeric** data type.
- If ds is a Dataset then it must have at least a measure of **numeric** data type.

*Returns*

If ds is a scalar then the operator returns ds raised to the *exponent* power.  
 If ds is a Dataset and has N numeric measures then the operator returns a Dataset having the Identifier.  
 Components of ds and the N numeric measures are obtained by elevating the original Measure Component to the exponent-th power.

*Examples*

On scalar

1) If A = 2, B = 5:

$$C := \text{power}(B, A) \quad C = 25$$

On Dataset

2) DatasetB := **power**(DatasetA,2)

2873

Dataset A			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	3
FR	Females	2011	4
FR	Total	2011	5

Dataset B			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	9
FR	Females	2011	16
FR	Total	2011	25

2874

**sqrt**

*Semantics*

The operator **sqrt** calculates the square root of a number

*Syntax*

**sqrt(ds)**

*Parameters*

ds : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as number}+]

2883



2884 {attribute <IDENT> as scalar-type}\* [number]

2885

2886 *ds* – is the input scalar number or the input Dataset.

2887

2888 *Constraints*

- 2889 • If *ds* is a scalar then it must be a **numeric** data type greater than zero.
- 2890 • If *ds* is a Dataset then it must have at least a measure of **numeric** data type having values greater than zero.

2891

2892 *Returns*

2893 If *ds* is a scalar then the operator returns the square root of *ds*.

2894 If *ds* is a Dataset and has N numeric measures then the operator returns a Dataset having the Identifier.

2895 Components of *ds* and the N numeric measures are obtained by calculating the square root of the original Measure Component.

2897

2898 The square root of a negative number results in a runtime exception.

2899 *Examples*

2900 On scalar

2901 1) If A = 25:

2902 B := **sqrt**(A) B = 5

2903

2904 On Dataset

2905 2) DatasetB := **sqrt**(DatasetA)

2906

Dataset A			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	16
FR	Females	2011	81
FR	Total	2011	64

2907

Dataset B			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	4
FR	Females	2011	9
FR	Total	2011	8

2908

2909 **nroot**

2910 *Semantics*

2911 The **nroot** operator calculates the n-th root of a number

2912

2913 *Syntax*

2914 **nroot**(*ds*, *index*)

2915

2916 *Parameters*

2917 *ds* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as number}+

2918 {attribute <IDENT> as scalar-type}\* [number]

2919

2919 *index* : integer

2920

- 2921 • *ds* – is the input scalar number or the input Dataset.
- 2922 • *index* – the index of the root.

2923

2924 *Constraints*

- 2925 • If *ds* is a scalar then it must be a **numeric** data type greater than zero when index even (dynamic).

- 2926 • If *ds* is a Dataset then it must have at least a measure of **numeric** data type having values greater than or  
 2927 equa to zero when index even (dynamic).

2928  
 2929 *Returns*

- 2930 • If *ds* is a scalar then the operator returns the index-th root of *ds*.  
 2931 • If *ds* is a Dataset and has *N* numeric measures then the operator returns a Dataset having the Identifier.  
 2932 Components of *ds* and the *N* numeric measures are obtained by calculating the index-th root of the original  
 2933 Measure Component.

2934  
 2935 *Semantic specification*

2936 In case of even index and negative argument, it results in a runtime exception.

2937  
 2938 *Examples*

2939 On scalar

2940 1) If  $A = 2, B = 25$ :

2941  $C := \mathbf{nroot}(B, A)$   $C = 5$

2942 On Dataset

2943 2)  $\text{DatasetB} := \mathbf{nroot}(\text{DatasetA}, 3)$

2945

Dataset A			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	8
FR	Females	2011	27
FR	Total	2011	64

2946

2947

Dataset B			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	2
FR	Females	2011	3
FR	Total	2011	4

2948

## 2949 mod

2950 *Semantics*

2951 The operator **mod** calculates the remainder of the division of a number by a denominator

2952

2953 *Syntax*

2954 **mod**(*ds*, *den*)

2955

2956 *Parameters*

2957 *ds* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as number}+  
 2958 {attribute <IDENT> as scalar-type}\* |number]

2959 *den*: integer

2960

- 2961 • *ds* – is the input scalar number or the input Dataset.

- 2962 • *den* – is the input denominator.

2963

2964 *Constraints*

- 2965 • If *ds* is a scalar then it must be a **numeric** data type.

- 2966 • If *ds* is a Dataset then it must have at least a measure of **numeric** data type.

- 2967 • *den* must be greater than zero.

2968  
2969  
2970  
2971  
2972  
2973  
2974  
2975  
2976  
2977  
2978  
2979  
2980  
2981  
2982  
2983  
2984  
2985  
2986

### Returns

- If *ds* is a scalar then the operator returns the remainder of the division of *ds* by *den*.
- If *ds* is a Dataset and has N numeric measures then the operator returns a Dataset having the Identifier Components of *ds* and the N numeric measures are obtained by calculating the remainder of the division of the original Measure Component by *den*.

### Semantics

The operator takes as input a numerator and a denominator and returns the remainder of the division of the numerator by the denominator.

### Examples

On scalar

1) If A = 5, B = 2:

$$C := \text{mod}(A, B) \qquad C = 1$$

On Dataset

2) DatasetB := mod(DatasetA,3)

Dataset A			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	7
FR	Females	2011	10
FR	Total	2011	12

2987

Dataset B			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	1
FR	Females	2011	1
FR	Total	2011	0

2988

## 2989 listsum

### 2990 Semantics

2991 **listsum** returns the sum of the specified values and replaces the missing data points with a zero value

2992

### 2993 Syntax

2994 **listsum** (*ds* { , *ds* } \* )

2995

### 2996 Parameters

2997 *ds* : dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as number}+  
2998 {attribute <IDENT> as scalar-type}\*

2999

3000 *ds* – is the input Dataset (s).

3001

### 3002 Constraints

3003 The Dataset (s) must have at least a measure of **numeric** data type.

3004

### 3005 Returns

3006 A Dataset denoting the sum of the values. If any expression evaluates to an empty data point then the 0 value is substituted for that expression. If all operands evaluate to empty data points then no data points are returned (i.e., the result is a Dataset containing no data points).

3009

3010 *Semantic specification*

3011 The difference with the + operator is that **listsum** substitutes an empty data point with 0 (therefore returning a  
3012 result) while the + operator returns an empty data point when one of the operands is an empty data point.

3013

3014 *Examples*

3015

3016 1) ds\_sum := **listsum** ( ds\_bop1 , ds\_bop2 )

3017

ds_bop1				
TIME	REF_AREA	PARTNER	OBS_VALUE	OBS_STATUS
2010	EU25	CA	20	D
2010	BG	CA	2	P
2010	RO	CA	2	P

3018

ds_bop2				
TIME	REF_AREA	PARTNER	OBS_VALUE	OBS_STATUS
2010	EU25	CA	10	D

3019

ds_sum				
TIME	REF_AREA	PARTNER	OBS_VALUE	OBS_STATUS
2010	EU25	CA	30	D
2010	BG	CA	2	P
2010	RO	CA	2	P

3020

3021 Compare with the "+" operator:

3022

3023 ds\_plus := ds\_bop1 + ds\_bop2

3024

ds_plus				
TIME	REF_AREA	PARTNER	OBS_VALUE	OBS_STATUS
2010	EU25	CA	30	D

3025

3026 2) ds\_sum := **listsum** ( ds\_bop1 , - ds\_bop2 )

3027

ds_sum				
TIME	REF_AREA	PARTNER	OBS_VALUE	OBS_STATUS
2010	EU25	CA	10	D
2010	BG	CA	2	P
2010	RO	CA	2	P

3028

3029

# VTL-ML - Boolean operators and functions

3030

equal to =

3031

### Semantic

3032

The operator = compares two values to evaluate if they are equal.

3033

3034

### Syntax

3035

$ds_1 = ds_2$

3036

3037

### Parameters

3038

$ds_1, ds_2$  : [dataset {identifier <IDENT> as scalar-type}<sup>+</sup> {measure <IDENT> as <T>}<sup>+</sup> | boolean]

3039

3040

$ds_1, ds_2$  - is a Dataset expression or a boolean

3041

3042

### Constraints

3043

- If both  $ds_1$  and  $ds_2$  are Datasets then they must have at least a Identifier Component in common (with the same name and data type).

3044

3045

- If both  $ds_1$  and  $ds_2$  are Datasets then either they have one or more measures in common, or at least one of them has only a measure.

3046

3047

3048

### Returns

3049

If both  $ds_1$  and  $ds_2$  are scalar values then the = operator returns a scalar boolean value representing the result of the *equal to* validation.

3050

3051

If either  $ds_1$  or  $ds_2$  is a Dataset then the = operator returns a Dataset having the following components:

3052

- The superset of the identifier components of  $ds_1$  and  $ds_2$

3053

- If  $ds_1$  and  $ds_2$  have one or more measures in common (i.e., with the same name) then the resulting Dataset has these common measures, with the same name concatenated with the suffix “\_CONDITION”, containing the results of the *equal to* validation of the respective measures of  $ds_1$  and  $ds_2$ . Otherwise, if  $ds_1$  and  $ds_2$  do not have any measures in common and have only one measure then the resulting Dataset contains a measure named **CONDITION** that contains the result of the *equal to* validation of the single measures of  $ds_1$  and  $ds_2$ .

3054

3055

3056

3057

3058

The resulting Dataset contains a data point for each pair of data points of  $ds_1$  and  $ds_2$  that have the same key (the same values of the Identifier Components).

3059

3060

3061

3062

### Semantic specification

3063

If the two values are equal, the the result of the validation will be *true* *false* if they differ.

3064

3065

### Examples

3066

On scalar

3067

1) If A = 5, B = 9, C = 5:

3068

D := A = B D = false

3069

D := A = C D = true

3070

2) If A = “hello”, B = “hi”, C = “Hi”:

3071

D := A = B D = false

3072

3073

On Dataset

3074

3) DSr:=Overcrowding\_rate\_urbanization = 0.08

3075

overcrowding_rate_urbanization				
TIME	GEO	AGE	SEX	VALUE
2012	Belgium	Total	Total	NULL
2012	Greece	Total	Total	0.286
2012	Spain	Total	Total	0.064

2012	Malta	Total	Total	0.043
2012	Finland	Total	Total	0.08
2012	Switzerland	Total	Total	0.08

3076

DSr				
TIME	GEO	AGE	SEX	CONDITION
2012	Belgium	Total	Total	NULL
2012	Greece	Total	Total	false
2012	Spain	Total	Total	false
2012	Malta	Total	Total	false
2012	Finland	Total	Total	true
2012	Switzerland	Total	Total	true

3077

## 3078 not equal to <>

### 3079 Semantic

3080 The operator <> compares two values to evaluate if they are not equal.

3081

### 3082 Syntax

3083  $ds_1 \langle \rangle ds_2$

3084

### 3085 Parameters

3086  $ds_1, ds_2$  : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as <T>}+ | boolean]

3087

3088  $ds_1, ds_2$  - is a Dataset expression or a boolean

3089

### 3090 Constraints

- 3091 • If both  $ds_1$  and  $ds_2$  are Datasets then they must have at least a Identifier Component in common (with the same name and data type).
- 3092 • If both  $ds_1$  and  $ds_2$  are Datasets then either they have one or more measures in common, or at least one of them has only a measure.

3095

### 3096 Returns

3097 If both  $ds_1$  and  $ds_2$  are scalar values then the operator returns a scalar Boolean value representing the result of the *not equal to* validation.

3098 If either  $ds_1$  or  $ds_2$  is a Dataset then the operator returns a Dataset having the following components:

- 3100 • The superset of the Identifier Components of  $ds_1$  and  $ds_2$
- 3101 • If  $ds_1$  and  $ds_2$  have one or more measures in common (i.e., with the same name) then the resulting Dataset has these common measures, with the same name concatenated with the suffix “\_CONDITION”, containing the results of the *not equal to* validation of the respective measures of  $ds_1$  and  $ds_2$ . Otherwise, if  $ds_1$  and  $ds_2$  do not have any measures in common and have only one measure then the resulting Dataset contains a measure named **CONDITION** that contains the result of the *not equal to* validation of the single measures of  $ds_1$  and  $ds_2$ .

3107 The resulting Dataset contains a data point for each pair of data points of  $ds_1$  and  $ds_2$  that have the same key (the same values of the Identifier Components).

3109

### 3110 Semantic specification

3111 If the values are not equals, the result of the validation will be *true*, *false* if they not differ.

### 3112 Examples

3113 On scalar

3114 1) If A = 5, B = 9, C = 5:

3115 D := A <> C D = false  
 3116 2) If A = "hello", B = "hi", C = "Hi":  
 3117 D := A <> B D = true

3118 On Dataset  
 3119 3) compare\_ds := y\_unemployment\_2012 <> y\_unemployment\_2011  
 3120  
 3121

Y_unemployment_2012				
GEO	SEX	UNIT	C_BIRTH	VALUE
Germany	Total	Percentage	Total	7.1
Greece	Total	Percentage	Total	NULL

3122

y_unemployment_2011				
GEO	SEX	UNIT	C_BIRTH	VALUE
Germany	Total	Percentage	Total	7.5
Greece	Total	Percentage	Total	3

3123

compare_ds				
GEO	SEX	UNIT	C_BIRTH	VALUE_CONDITION
Germany	Total	Percentage	Total	true
Greece	Total	Percentage	Total	NULL

3124

3125

3126

If VALUE for Greece in the second operand had also been NULL, then the result would still be NULL for Greece.

3127 **greater than** > >=

3128

#### Semantic

3129

The operator > >= compares two values to evaluate if one is greater (or equal) to the other.

3130

3131

#### Syntax

3132

*ds\_1* [ > | >= ] *ds\_2*

3133

3134

#### Parameters

3135

*ds\_1*, *ds\_2* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as <T>}+ | boolean]

3136

3137

*ds\_1*, *ds\_2* – is a Dataset expression or a Boolean

3138

3139

#### Constraints

3140

- If both *ds\_1* and *ds\_2* are Datasets then they must have at least one Identifier Component in common (with the same name and data type).

3141

3142

- If both *ds\_1* and *ds\_2* are Datasets then either they have one or more measures in common, or at least one of them has only a measure.

3143

3144

3145

#### Returns

3146

If both *ds\_1* and *ds\_2* are scalar values then the operator returns a scalar boolean value representing the result of

3147

the comparison *greater* (or *equal*) validation.

3148

If either *ds\_1* or *ds\_2* is a Dataset then the operator returns a Dataset having the following components:

3149

- The superset of the Identifier Components of *ds\_1* and *ds\_2*

3150

- If *ds\_1* and *ds\_2* have one or more measures in common (i.e., with the same name) then the resulting Dataset has these common measures, with the same name concatenated with the suffix “\_CONDITION”, containing the results of the comparison greater (or equal) validation of the respective measures of *ds\_1* and *ds\_2*.

3151

Otherwise, if *ds\_1* and *ds\_2* do not have any measures in common and have only one measure then the

3152

3153

3154 resulting Dataset contains a measure named **CONDITION** that contains the result of the comparison greater  
 3155 (or equal) validation of the single measures of *ds\_1* and *ds\_2*.  
 3156 The resulting Dataset contains a data point for each pair of data points of *ds\_1* and *ds\_2* that have the same key  
 3157 (the same values of the Identifier Components).

3158 *Semantic specification*

3159 If the value on the left side is greater (or equal) than the value on the right side, the result of the validation will  
 3161 be *true* if, *false* if not or either of them is NULL.

3162 *Examples*

3163 On scalar

- 3164 1) If A = 5, B = 9, C = 5:  
 3166 D := A > B D = false  
 3167 D := A >= C D = true  
 3168 2) If A = "hello", B = "hi", C = "Hi":  
 3169 D := A > B D = false

3170 On Dataset

3171 3) compare\_ds := foreign\_languages\_known > 20  
 3172  
 3173

foreign_languages_known					
N_LANG	GEO	TIME	AGE	UNIT	VALUE
2	Germany	2011	Total	Percentage	NULL
2	Greece	2011	Total	Percentage	12.2
2	Finland	2011	Total	Percentage	29.5

3174

compare_ds					
N_LANG	GEO	TIME	AGE	UNIT	CONDITION
2	Germany	2011	Total	Percentage	NULL
2	Greece	2011	Total	Percentage	false
2	Finland	2011	Total	Percentage	true

3175

3176 4) compare\_ds := y\_unemployment\_2012 > y\_unemployment\_2011  
 3177

y_unemployment_2012				
GEO	SEX	UNIT	C_BIRTH	VALUE
Germany	Total	Percentage	Total	7.1
Greece	Total	Percentage	Total	42.5

3178

y_unemployment_2011				
GEO	SEX	UNIT	C_BIRTH	VALUE
Germany	Total	Percentage	Total	7.5
Greece	Total	Percentage	Total	33.7

3179

compare_ds				
GEO	SEX	UNIT	C_BIRTH	VALUE_CONDITION
Germany	Total	Percentage	Total	false
Greece	Total	Percentage	Total	true

3180



3181 If the VALUE column for Germany in the y\_unemployment\_2012 Dataset had a NULL value the result would be:

compare_ds				
GEO	SEX	UNIT	C_BIRTH	VALUE_CONDITION
Germany	Total	Percentage	Total	NULL
Greece	Total	Percentage	Total	true

3182

3183 less than < <=

3184 *Semantic*

3185 The operator < <= compares two values to evaluate if one is less (or equal) to the other.

3186

3187 *Syntax*

3188  $ds_1$  [ < | <= ]  $ds_2$

3189

3190 *Parameters*

3191  $ds_1, ds_2$  : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as <T>}+ | boolean]

3192

3193  $ds_1, ds_2$  – is a Dataset expression or a boolean

3194

3195 *Constraints*

- 3196 • If both  $ds_1$  and  $ds_2$  are Datasets then they must have at least one Identifier Component in common (with the same name and data type).
- 3197 • If both  $ds_1$  and  $ds_2$  are Datasets then either they have one or more measures in common, or at least one of them has only a measure.

3200

3201 *Returns*

3202 If both  $ds_1$  and  $ds_2$  are scalar values then the operator returns a scalar Boolean value representing the results of the comparison *less* (or *equal*) than validation.

3203 If either  $ds_1$  or  $ds_2$  is a Dataset then the operator returns a Dataset having the following components:

- 3204 • The superset of the Identifier Components of  $ds_1$  and  $ds_2$
- 3205 • If  $ds_1$  and  $ds_2$  have one or more measures in common (i.e., with the same name) then the resulting Dataset has these common measures, with the same name concatenated with the suffix “\_CONDITION”, containing the results of the comparison *less* (or *equal*) than validation of the respective measures of  $ds_1$  and  $ds_2$ . Otherwise, if  $ds_1$  and  $ds_2$  do not have any measures in common and have only one measure then the resulting Dataset contains a measure named **CONDITION** that contains the results of the comparison *less* (or *equal*) than validation of the single measures of  $ds_1$  and  $ds_2$ .

3209 The resulting Dataset contains a data point for each pair of data points of  $ds_1$  and  $ds_2$  that have the same key (the same values of the Identifier Components).

3214

3215 *Semantic specification*

3216 *f* the value on the left side is less (or equal) than the value on the right side the result of the validation will be *true*, *false* if not or if either of them is NULL.

3217

3218 *Examples*

3219 On scalar

3220 1) If A = 5, B = 9, C = 5:

3221 D := A < B D = true

3222 D := A <= C D = true

3223 2) If A = “hello”, B = “hi”, C = “Hi”:

3224 D := C < B D = false

3225

3226 On Dataset

3227 3) compare\_ds := total\_population < 15000000

3228

total_population				
TIME	GEO	AGE	SEX	VALUE

2012	Belgium	Total	Total	11094850
2012	Greece	Total	Total	11123034
2012	Spain	Total	Total	46818219
2012	Malta	Total	Total	NULL
2012	Finland	Total	Total	5401267
2012	Switzerland	Total	Total	7954662

3229

compare_ds				
TIME	GEO	AGE	SEX	CONDITION
2012	Belgium	Total	Total	true
2012	Greece	Total	Total	true
2012	Spain	Total	Total	false
2012	Malta	Total	Total	NULL
2012	Finland	Total	Total	true
2012	Switzerland	Total	Total	true

3230

## 3231 in, not in

### 3232 *Semantic*

3233 The operator **in**, **not in** verifies if a value belongs to a set of values of a set or a list

3234

### 3235 *Syntax*

3236 *ds* {**not**} **in** [*list* | *inlineList* ]

3237

### 3238 *Parameters*

3239 *ds* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as <T>}+|constant<T>]

3240 *list* : list-ref

3241 *inlineList* : list( {constant<T>}+ )

3242

3243 • *ds* – is a Dataset expression or a scalar

3244 • *list* – is a reference to a valid List.

3245 • *inlineList* – is an in-line specification of a List. The elements of the List are constants.

### 3246 *Constraints*

3247 • if *ds* is a scalar the elements of the List must be of the same type and If set is specified, then it must be a reference to a mono-dimensional Set.

3249 • if *ds* is a Dataset, all the Measure Components of *ds* must have the same type T (which is also the type of the Set or List),

3250

### 3252 *Returns*

3253 If *ds* is a scalar then **in**, **not in** returns a Boolean value representing the presence of the constant in the List.

3254 If *ds* is a Dataset and has N measures then **in**, **not in** returns a Dataset having the identifier components of *ds* and N Boolean Measure Components having the same name concatenated with the suffix “\_CONDITION” that states if the values of *ds* are (not) in the *list*.

3257

### 3258 *Examples*

3259 On Dataset

3260 *ds\_1* := total\_population **in** (11094850, 46818219, 222, 111)

total_population				
TIME	GEO	AGE	SEX	POPULATION
2012	Belgium	Total	Total	11094850

2012	Greece	Total	Total	11123034
2012	Spain	Total	Total	46818219
2012	Malta	Total	Total	417546
2012	Finland	Total	Total	5401267
2012	NULL	Total	Total	7954662

3261

ds_1				
TIME	GEO	AGE	SEX	POPULATION_CONDITION
2012	Belgium	Total	Total	true
2012	Greece	Total	Total	false
2012	Spain	Total	Total	true
2012	Malta	Total	Total	false
2012	Finland	Total	Total	false
2012	NULL	Total	Total	false

3262

3263

## between

3264

### Semantic

3265

The operator **between** verifies if a value belongs to an interval of values

3266

3267

### Syntax

3268

*ds\_1* **between** *ds\_2* and *ds\_3*

3269

3270

### Parameters

3271

*ds\_1*, *ds\_2*, *ds\_3* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as <T>}+ |constant<T>]

3272

3273

- *ds\_1* – is the Dataset or the scalar to validate.

3274

- *ds\_2* – is the lowerbound of a value's range.

3275

- *ds\_3* – is the upperbound of a value's range.

3276

3277

### Constraints

3278

- If *ds\_1* is a scalar then the defined constants must be all of the same type.

3279

- If *ds\_1* is a Dataset then:

3280

- At least one Dataset must be defined.

3281

- If two (or three) Datasets are defined, for every pair of Datasets, it must hold that either they have the same Identifier Components or the ones of the former is a subset of the ones of the latter (static).

3282

- If two (or three) Datasets are defined, they must have the same Measure Components, in name and number (as explained in the syntax) (static).

3283

3284

- If at least one Constant is defined, the Datasets must have a single Measure Component of type <T> (static).

3285

3286

3287

3288

If *ds\_1* is a scalar then **between** returns a Boolean value representing if *c\_1* is greater or equal than *c\_2* and less or equal than *c\_3*.

3291

If *ds* is a Dataset and has N measures then operator returns a Dataset having the Identifier Components of *ds* and N Boolean measures having the same name concatenated with the suffix “\_CONDITION” containing the result of the range comparison.

3294

3295

### Examples

3296

On Dataset

3297

1) comparison\_ds := unemployment\_rate **between** 7.5 and 8.0

3298

unemployment_rate		
TIME	GEO	UNEMPLOYMENT_RATE
2013	Finland	8.2
2012	Finland	7.7
2011	Finland	7.8
2010	Finland	8.4
2009	Finland	NULL

3299

comparison_ds		
TIME	GEO	UNEMPLOYMENT_RATE
2013	Finland	false
2012	Finland	true
2011	Finland	true
2010	Finland	false
2009	Finland	NULL

3300

3301

3302

3303

2) comparison\_ds := overcrowding\_rate\_urbanization\_2011 **between** Overcrowding\_rate\_urbanization\_2010 **and** Overcrowding\_rate\_urbanization\_2012

overcrowding_rate_urbanization_2011	
GEO	VALUE
Belgium	NULL
Greece	0.276
Finland	0.093
Switzerland	0.08
United Kingdom	0.089
France	0.125

3304

overcrowding_rate_urbanization_2010			
GEO	AGE	SEX	VALUE
Belgium	Total	Total	0.06
Greece	Total	Total	0.281
Spain	Total	Total	0.06
Malta	Total	Total	0.041
Switzerland	Total	Total	NULL

3305

overcrowding_rate_urbanization_2012				
TIME	GEO	AGE	SEX	VALUE
2012	Belgium	Total	Total	0.023
2012	Greece	Total	Total	0.286
2012	Spain	Total	Total	0.064
2012	Malta	Total	Total	0.043
2012	Finland	Total	Total	0.08

2012	Switzerland	Total	Total	0.08
------	-------------	-------	-------	------

3306  
3307

comparison_ds				
TIME	GEO	AGE	SEX	VALUE_CONDITION
2012	Belgium	Total	Total	NULL
2012	Greece	Total	Total	false
2012	Switzerland	Total	Total	NULL

3308

## 3309 isnull

### 3310 *Semantics*

3311 The **isnull** operator, compares the values with the NULL.

3312

### 3313 *Syntax*

3314 **isnull(ds)**

3315

### 3316 *Parameters*

3317 *ds* : [ dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as scalar-type }\*  
3318 {attribute <IDENT> as scalar-type}\*|constant]

3319

3320 *ds* – is a Dataset or a scalar value.

3321

### 3322 *Constraints*

3323 None

3324

### 3325 *Returns*

3326 If *ds* is a scalar then **isnull** returns a boolean value representing if the value is (not) NULL.

3327 If *ds* is a Dataset and has N measures then **isnull** returns a Dataset having the Identifier Components of *ds* and N  
3328 numeric measures with the same name concatenated with the suffix “\_CONDITION” but assuming a boolean  
3329 value if the value is (not) NULL.

3330

### 3331 *Examples*

3332 On scalar

3333 1) If C is null:

3334 A := **isnull(C)** A = true

3335 2) If C is not null:

3336 A := **isnull(C)** A = false

3337

3338 On Dataset

3339 3) ds\_1 := **isnull(population)**

3340

population				
TIME	GEO	AGE	SEX	POPULATION
2012	Belgium	Total	Total	11094850
2012	Greece	Total	Total	11123034
2012	Spain	Total	Total	NULL
2012	Malta	Total	Total	417546
2012	Finland	Total	Total	5401267
2012	NULL	Total	Total	NULL

3341

ds_1				
TIME	GEO	AGE	SEX	CONDITION
2012	Belgium	Total	Total	false
2012	Greece	Total	Total	false
2012	Spain	Total	Total	true
2012	Malta	Total	Total	false
2012	Finland	Total	Total	false
2012	NULL	Total	Total	true

3342

## 3343 exists\_in, not\_exists\_in/in\_all

### 3344 *Semantics*

3345 The **exists\_in, not\_exists\_in/in\_all** operators match the existence or not of data points of a Dataset in another  
3346 Dataset.

3347

### 3348 *Syntax*

3349 `ds_1 [exists_in|exists_in_all|not_exists_in|not_exists_in_all ] ds_2`

3350 Parameters

3351 `ds_1, ds_2` : dataset {identifier <IDENT> as <T>}+ {measure <IDENT> as <T> }\*  
3352 {attribute <IDENT> as scalar-type}\*

3353

3354 `ds_1, ds_2` – are the input Datasets.

3355

### 3356 *Constraints*

3357 `ds_1` and `ds_2` must have at least one Identifier Component in common (with the same name and data type).

3358

### 3359 *Returns*

3360 A Dataset with all Identifier Components of the two Datasets and one boolean Measure Component named  
3361 **CONDITION**. The Measure Component value in each Data Points in the output indicates whether a Data Point  
3362 with matching key (not) exists in the second argument for the corresponding Data Point of the first argument.

3363

### 3364 *Semantic specification*

3365 If **all** versions are used, both the *true* and the *false* Data Points are kept in the result. Otherwise, only the *true*  
3366 Data Points are kept.

3367

### 3368 *Examples*

3369 1) `ds_check := population exists_in_all urbanization_rate`

3370

population				
TIME	GEO	AGE	SEX	POPULATION
2012	Belgium	Total	Total	11094850
2012	Greece	Total	Total	11123034
2012	Spain	Total	Total	46818219
2012	Malta	Total	Total	417546
2012	Finland	Total	Total	5401267
2012	Switzerland	Total	Total	7954662

3371

urbanization_rate				
TIME	GEO	AGE	SEX	RATE

2012	Belgium	Total	Total	0.023
2012	Greece	Total	Total	0.286
2012	Spain	Total	Total	0.064
2012	Malta	Total	Total	0.043
2012	Finland	Total	Total	NULL
2012	Switzerland	Total	Total	0.08

3372

ds_check				
TIME	GEO	AGE	SEX	CONDITION
2012	Belgium	Total	Total	true
2012	Greece	Total	Total	false
2012	Spain	Total	Total	true
2012	Malta	Total	Total	false
2012	Finland	Total	Total	false
2012	Switzerland	Total	Total	true

3373

3374

3375

2) R := C1 exists\_in C2

C1		
K1	K2	M1
1	A	100
2	B	200
3	C	700
4	A	550
5	D	120

3376

C2		
K1	K2	M1
1	A	100
2	B	200
5	D	700

3377

R		
K1	K2	CONDITION
1	A	True
2	B	True
5	D	True

3378

3379

3380

3) R := C1 exists\_in\_all C2

C1		
K1	K2	M1
1	A	100

2	B	200
3	C	700
4	A	550
5	D	120

3381  
3382

C2		
K1	K2	M1
1	A	100
2	B	200
5	D	700

3383  
3384

R		
K1	K2	CONDITION
1	A	True
2	B	True
3	C	False
4	A	False
5	D	True

3385  
3386  
3387

4) R := C1 does **not\_exist\_in** C2

C1		
K1	K2	M1
1	A	100
2	B	200
3	C	700
4	A	550
5	D	120

3388

C2		
K1	K2	M1
1	A	100
2	B	200
5	D	700

3389

R		
K1	K2	CONDITION
3	C	True
4	A	True

3390  
3391  
3392  
3393



3394 5) R := C1 not\_exists\_in\_all C2  
 3395

C1		
K1	K2	M1
1	A	100
2	B	200
3	C	700
4	A	550
5	D	120

3396

C2		
K1	K2	M1
1	A	100
2	B	200
5	D	700

3397

R		
K1	K2	CONDITION
1	A	False
2	B	False
3	C	True
4	A	True
5	D	False

3398

3399 6) R := C1 not\_exists\_in\_all C2  
 3400

C2			
K1	K2	K3	M1
1	A	X	100
2	B	Y	200
5	D	Z	700
5	D	K	1500

3401

C1		
K1	K2	M1
1	A	100
2	B	200
3	C	700
4	A	550
5	D	120

3402

3403

R		
K1	K2	CONDITION

1	A	False
2	B	False
3	C	True
4	A	True
5	D	False

3404

## 3405 match\_characters

### 3406 *Semantics*

3407 The **match\_character** operator checks whether a value respects a given pattern

3408

### 3409 *Syntax*

3410 **Match\_characters** ( *ds*, *pattern* {, **all**})

3411

### 3412 *Parameters*

3413 *ds* : dataset {identifier <IDENT> as scalar-type}<sup>+</sup> {measure <IDENT> as string }<sup>\*</sup>  
 3414 {attribute <IDENT> as scalar-type}<sup>\*</sup>

3415 *pattern* : regexp

3416 *regexp* : string

3417

- 3418 • *ds* – is the input Dataset.
- 3419 • *pattern* – is a regular expression that defines a string pattern.
- 3420 • *regexp* – is a regular expression.

3421

### 3422 *Constraints*

- 3423 • *ds* must have only one **string** Measure Components (static).
- 3424 • *pattern* must be a regular expression according to POSIX extended standard  
 3425 ([http://pubs.opengroup.org/onlinepubs/009696899/basedefs/xbd\\_chap09.html](http://pubs.opengroup.org/onlinepubs/009696899/basedefs/xbd_chap09.html)) (static).

3426

### 3427 *Returns*

3428 A Dataset having the same Identifier and Attribute Components and a Boolean Measure Component for each  
 3429 string Measure Component in *ds* with the same name concatenated with the suffix “\_CONDITION”, containing  
 3430 the value resulting from the matching between the values in *ds* and the specified *pattern*.

3431

### 3432 *Semantic specification*

3433 The Data Points of *ds* are copied into the output Dataset; the Boolean Measure Component will have *true* if the  
 3434 respective in *ds* matches with the *pattern*, *false* otherwise.

3435 The **all** flag allows to specify that both *true* and *false* Data Points have to be kept in the output. If it is not present,  
 3436 only *true* Data Points are kept.

3437

### 3438 *Examples*

3439 *ds\_r* := **match\_characters**(population.TIME, “[123456789,]”, **all**)

3440

population				
TIME	GEO	AGE	SEX	POPULATION
2012	Belgium	Total	Total	11094850
2012A	Greece	Total	Total	11123034
2012	Spain	Total	Total	46818219
2012	Malta	Total	Total	417546
2012	Finland	Total	Total	5401267
2012C	Switzerland	Total	Total	7954662

3441

population				
TIME	GEO	AGE	SEX	POPULATION_CONDITION
2012	Belgium	Total	Total	true
2012A	Greece	Total	Total	false
2012	Spain	Total	Total	true
2012	Malta	Total	Total	true
2012	Finland	Total	Total	true
2012C	Switzerland	Total	Total	false

3442

## 3443 all

### 3444 *Semantics*

3445 The **all** operator verifies that all values in the Dataset are true

3446

### 3447 *Syntax*

3448 **all** (*ds*)

3449

### 3450 *Parameters*

3451 *ds* : dataset {identifier <IDENT> as scalar-type }+ {measure <IDENT> as boolean }+  
 3452 {attribute <IDENT> as scalar-type }\*

3453

3454 *ds* – is a Dataset

3455

### 3456 *Constraints*

3457 *ds* must have at least a measure of type **boolean**.

3458

### 3459 *Returns*

3460 A Dataset with only one Boolean measure, called **CONDITION**, equal to *true* if, for all data points of the input  
 3461 Dataset, the Boolean measures are equal to *true*, *false* otherwise.

3462

3463

### 3464 *Examples*

3465 1) *ds\_2* := **all**( *ds\_1*.VALUE>100 )

3466

ds_1		
GENDER	TIME	VALUE
M	2000	200
F	2000	50
M	2001	150
F	2001	120

3467

ds_2
CONDITION
FALSE

3468

3469

3470

3471 2) ds\_2 := **all**( ds\_1[filter time= "2001"].VALUE>100 )  
 3472

ds_1		
GENDER	TIME	VALUE
M	2000	200
F	2000	50
M	2001	150
F	2001	120

3473

ds_2
CONDITION
TRUE

3474 3) ds\_2 := **all**(ds\_1.VALUE\_2000>100 AND ds\_1.VALUE\_2001>100 )  
 3475  
 3476

ds_1		
GENDER	VALUE_2000	VALUE_2001
M	200	150
F	50	120

3477

ds_2
CONDITION
FALSE

3478

## 3479 any

### 3480 *Semantics*

3481 The **any** operator verifies that at least one value in the Dataset is true

3482

### 3483 *Syntax*

3484 **any** (*ds*)

3485

### 3486 *Parameters*

3487 *ds* : dataset {identifier <IDENT> as scalar-type }+ {measure <IDENT> as boolean}+

3488 {attribute <IDENT> as scalar-type }\*

3489

3490 *ds* – is a Dataset

3491

### 3492 *Constraints*

3493 *ds* must have at least one measure of type **Boolean**.

3494

### 3495 *Returns*

3496 A Dataset with only one Boolean measure, called **CONDITION**, equal to *true* if, for at least one data point of the  
 3497 input Dataset, the Boolean measures are equal to *true*, *false* otherwise.

3498

3499

3500

3501 *Examples*

3502 1) ds\_2 := **any**( ds\_1.VALUE>100 )

ds_1		
GENDER	TIME	VALUE
M	2000	200
F	2000	50
M	2001	90
F	2001	120

3503

ds_2
CONDITION
TRUE

3504

3505 2) ds\_2 := **any** (ds\_1.VALUE\_2000>100 AND ds\_1. VALUE\_2001>100 )

3506

ds_1		
GENDER	VALUE_2000	VALUE_2001
M	200	90
F	50	120

3507

ds_2
CONDITION
FALSE

3508

3509 3) ds\_2 := **any** (ds\_1.VALUE\_2000>100 AND ds\_1.VALUE\_2001>100)

3510

ds_1		
GENDER	VALUE_2000	VALUE_2001
M	200	90
F	50	120

3511

ds_2
CONDITION
TRUE

3512

3513

3514

3515

3516 **unique**

3517 *Semantics*

3518 The **unique** operator verifies the presence of one single Data Point having true as the value for the Measure  
3519 component.

3520

3521 *Syntax*

3522 **unique (ds)**

3523

3524 *Parameters*

3525 ds : dataset {identifier <IDENT> as scalar-type }+ {measure <IDENT> as boolean }+  
3526 {attribute <IDENT> as scalar-type }\*

3527

3528 ds – is a Dataset

3529

3530 *Constraints*

3531 ds must have at least a measure of type **boolean**.

3532

3533 *Returns*

3534 A Dataset with only one boolean measure, called **CONDITION**, equal to *true* if, for only one data point of the input  
3535 Dataset, the boolean measures are is equal to *true*, *false* otherwise.

3536

3537 *Examples*

3538 1) ds\_2 := **unique** ( ds\_1.VALUE>100 )

3539

ds_1		
GENDER	TIME	VALUE
M	2000	200
F	2000	150
M	2001	90
F	2001	120

3540

ds_2
CONDITION
FALSE

3541

3542

3543

3544 2) ds\_2 := **unique** (ds\_1.VALUE\_2000>100 AND ds\_1.VALUE\_2001>100 )

3545

ds_1		
GENDER	VALUE_2000	VALUE_2001
M	200	90
F	150	120

3546

ds_2
CONDITION
TRUE

3547  
 3548 3) ds\_2 := **unique** (ds\_1.VALUE\_2000>100 AND ds\_1.VALUE\_2001>100 )  
 3549

ds_1		
GENDER	VALUE_2000	VALUE_2001
M	200	90
F	150	120

3550

ds_2
CONDITION
FALSE

3551

## 3552 func\_dep

### 3553 Semantics

3554 The **func\_dep** operator checks the functional dependency between components of a Dataset.

3555

### 3556 Syntax

3557 **func\_dep** ( ds , listCompFrom , listCompTo )

3558

3559 *listCompFrom* : (comp { , comp } \*)

3560 *listCompTo* : (comp { , comp } \*)

3561

### 3562 Parameters

3563 *ds* : dataset {identifier <IDENT> as scalar-type }+ {measure <IDENT> as scalar-type }+  
 3564 {attribute <IDENT> as scalar-type }\*

3565

3566 *listCompFrom* –the components that form the left side of the functional dependency

3567 *listCompTo* – the components that form the right side of the functional dependency

3568

### 3569 Constraints

3570 None

3571

### 3572 Returns

3573 A Dataset having the only measure **CONDITION**, assuming value *true* if the functional dependency between the  
 3574 left and the right side is respected.

3575

### 3576 Semantic specification

3577 The *func\_dep* operator verifies the existence of a functional dependency from the components in *listCompFrom*  
 3578 to the the components in *listCompTo* ( *listCompFrom* → *listCompTo* ), that is, each combination of values of the  
 3579 components *listCompFrom* corresponds to one combination of values of the components *listCompTo*.

3580

3581

### 3582 Examples

3583 1) ds\_2 := **func\_dep** ( ds\_1, (FISCAL\_CODE), (NAME))

3584

ds_1			
FISCAL_CODE	NAME	DATE_OF_BIRTH	PLACE_OF_BIRTH
FC1	John Smith	10/09/1968	London
FC2	Helen Brown	18/10/1976	London

FC3	Steve McGill	21/08/1966	Dublin
FC4	Helen Brown	26/02/2001	Dublin

3585

ds_2
<b>CONDITION</b>
TRUE

3586

3587 2) ds\_3:= **func\_dep** ( ds\_1, (FISCAL\_CODE ), (NAME, DATE\_OF\_BIRTH,PLACE\_OF\_BIRTH))

3588

ds_3
<b>CONDITION</b>
TRUE

3589

3590 3) ds\_2 := **func\_dep** ( NAME), (FISCAL\_CODE) )

3591

ds_2
<b>CONDITION</b>
FALSE

3592

## 3593 and

### 3594 *Semantics*

3595 The **and** operator calculates the logical AND

3596

### 3597 *Syntax*

3598 *ds\_1 and ds\_2*

3599

### 3600 *Parameters*

3601 *ds\_1, ds\_2* : [dataset {identifier <IDENT> as scalar-type }+ {measure <IDENT> as boolean}+  
3602 {attribute <IDENT> as scalar-type}\*|boolean]

3603

3604 *ds\_1, ds\_2* – are the input Dataset or boolean scalars.

3605

### 3606 *Constraints*

- 3607 • If *ds\_1* (*ds\_2*) is a scalar then it must be a **boolean** data type.
- 3608 • If *ds\_1* (*ds\_2*) is a Dataset then it has at least a measure of **boolean** type.
- 3609 • If both *ds\_1* and *ds\_2* are Datasets then they must have at least one Identifier Component in common (with the same name and data type).
- 3610 • If both *ds\_1* and *ds\_2* are Datasets then either they have one or more boolean measures in common, or at least one of them has only a boolean measure.

3613

### 3614 *Returns*

3615 If both *ds\_1* and *ds\_2* are scalar values then the **and** operator returns a boolean value that is the result of the **and** operation.

3617 If either *ds\_1* or *ds\_2* is a Dataset then the **and** operator returns a Dataset having the following components:

- 3618 • The superset of the Identifier Components of *ds\_1* and *ds\_2*
- 3619 • If *ds\_1* and *ds\_2* have one or more Boolean measures in common (i.e., with the same name) then the resulting Dataset has these common Boolean measures, with the same name, varied on the base of the logical **and**

3620



3621 between the Measure Components of ds\_1 and ds\_2. Otherwise, if ds\_1 and ds\_2 do not have any measures in  
 3622 common and have only one measure then the resulting Dataset contains a measure named **CONDITION** that  
 3623 contains a Boolean value that is the result of the **and** operation.  
 3624 The resulting Dataset contains a data point for each pair of data points of ds\_1 and ds\_2 that have the same key  
 3625 (the same values of the Identifier Components).  
 3626

3627 *Examples*

3628 On scalar

3629 1) If A = True, B = False

3630 C := A **and** B C = False

3631 On Dataset

3632 2) ds\_r:=population.sex="M" **and** population.age="Y15-64"  
 3634

population				
SEX	AGE	GEO	TIME	VALUE
M	Y_LT15	BE	2013	970428
M	Y15-64	BE	2013	3678355
M	Y_GE65	BE	2013	838653
F	Y_LT15	BE	2013	927644
F	Y15-64	BE	2013	3625561
F	Y_GE65	BE	2013	1121001
M	Y_LT15	UK	2013	5757444
M	Y15-64	UK	2013	20748657
M	Y_GE65	UK	2013	4917238
F	Y_LT15	UK	2013	5488356
F	Y15-64	UK	2013	20915924
F	Y_GE65	UK	2013	6068452

3635  
 3636

ds_r				
SEX	AGE	GEO	TIME	CONDITION
M	Y_LT15	BE	2013	false
M	Y15-64	BE	2013	true
M	Y_GE65	BE	2013	false
F	Y_LT15	BE	2013	false
F	Y15-64	BE	2013	false
F	Y_GE65	BE	2013	false
M	Y_LT15	UK	2013	false
M	Y15-64	UK	2013	true
M	Y_GE65	UK	2013	false
F	Y_LT15	UK	2013	false
F	Y15-64	UK	2013	false
F	Y_GE65	UK	2013	false

3637



F	Y15-64	UK	2013	20915924
F	Y_GE65	UK	2013	6068452

3679  
3680

DS_or				
SEX	AGE	GEO	TIME	CONDITION
M	Y_LT15	BE	2013	true
M	Y15-64	BE	2013	true
M	Y_GE65	BE	2013	true
F	Y_LT15	BE	2013	false
F	Y15-64	BE	2013	true
F	Y_GE65	BE	2013	false
M	Y_LT15	UK	2013	true
M	Y15-64	UK	2013	true
M	Y_GE65	UK	2013	true
F	Y_LT15	UK	2013	false
F	Y15-64	UK	2013	true
F	Y_GE65	UK	2013	true

3681  
3682

## 3683 XOR

### 3684 *Semantics*

3685 The **xor** operator calculates the logical XOR

3686

### 3687 *Syntax*

3688  $ds_1 \text{ xor } ds_2$

3689

### 3690 *Parameters*

3691  $ds_1, ds_2$  : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as boolean}+  
3692 {attribute <IDENT> as scalar-type}\*[boolean]

3693

3694  $ds_1, ds_2$  – are the input Dataset or boolean scalars.

3695

### 3696 *Constraints*

- 3697 • If  $ds_1$  ( $ds_2$ ) is a scalar then it must be a **boolean** data type.
- 3698 • If  $ds_1$  ( $ds_2$ ) is a Dataset then it has at least a Measure of **boolean** type.
- 3699 • If both  $ds_1$  and  $ds_2$  are Datasets then they must have at least one Identifier Component in common (with the same name and data type).
- 3700 • If both  $ds_1$  and  $ds_2$  are Datasets then either they have one or more boolean Measures in common, or at least one of them has only a Boolean Measure.

3703

### 3704 *Returns*

3705

3706 If both  $ds_1$  and  $ds_2$  are scalar values then the **xor** operator returns a boolean value that is the result of the **xor** operation.

3707 If either  $ds_1$  or  $ds_2$  is a Dataset then the **xor** operator returns a Dataset having the following components:

- 3708 • The superset of the identifier components of  $ds_1$  and  $ds_2$
- 3709 • If  $ds_1$  and  $ds_2$  have one or more boolean Measures in common (i.e., with the same name) then the resulting Dataset has these common boolean Measures, with the same name, varied on the base of the logical **xor**

3711

3712 between the Measure Components of ds\_1 and ds\_2. Otherwise, if ds\_1 and ds\_2 do not have any Measures in  
 3713 common and have only one Measure then the resulting Dataset contains a Measure named **CONDITION** that  
 3714 contains a boolean value that is the result of the **xor** operation.  
 3715 The resulting Dataset contains a data point for each pair of data points of ds\_1 and ds\_2 that have the same key  
 3716 (the same values of the Identifier Components).

3717 *Examples*

3718 On scalar

3719 1) If A = True, B = False

3720 C := A **or** B

C = True

3722 On Dataset

3723 2) DS\_xor:=population.sex="M" **xor** population.age\_group="Y15-64"

3724

3725

DS_xor				
SEX	AGE	GEO	TIME	CONDITION
M	Y_LT15	BE	2013	970428
M	Y15-64	BE	2013	3678355
M	Y_GE65	BE	2013	838653
F	Y_LT15	BE	2013	927644
F	Y15-64	BE	2013	3625561
F	Y_GE65	BE	2013	1121001
M	Y_LT15	UK	2013	5757444
M	Y15-64	UK	2013	20748657
M	Y_GE65	UK	2013	4917238
F	Y_LT15	UK	2013	5488356
F	Y15-64	UK	2013	20915924
F	Y_GE65	UK	2013	6068452

3726  
3727

DS_xor				
SEX	AGE	GEO	TIME	CONDITION
M	Y_LT15	BE	2013	true
M	Y15-64	BE	2013	false
M	Y_GE65	BE	2013	true
F	Y_LT15	BE	2013	false
F	Y15-64	BE	2013	true
F	Y_GE65	BE	2013	false
M	Y_LT15	UK	2013	true
M	Y15-64	UK	2013	false
M	Y_GE65	UK	2013	true
F	Y_LT15	UK	2013	false
F	Y15-64	UK	2013	true
F	Y_GE65	UK	2013	false

3728

3729 **not**

3730 *Semantics*

3731 The **not** operator calculates the logical negation of a boolean condition

3732

3733 *Syntax*

3734 **not** *ds\_1*

3735

3736 *Parameters*

3737 *ds* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as string-literal}+  
3738 {attribute <IDENT> as scalar-type}\* | boolean]

3739

3740 *ds* – is a Dataset expression or a string

3741

3742 *Constraints*

3743 If *ds* is a scalar then it must be a **boolean** type.

3744 If *ds* is a Dataset then it must have at least a Measure of type **boolean**.

3745

3746 *Returns*

3747 If *ds* is a scalar then **not** returns the logical negation of *ds*.

3748 If *ds* is a Dataset and has N boolean Measures then **not** returns a Dataset having the Identifier Components of *ds*  
3749 and N numeric Measures with the same name of the boolean Measures of *ds* and containing the logical negation  
3750 of the corresponding Measures.

3751

3752 *Examples*

3753 On scalar

3754 1) If A = True

3755 B := **not** A

3756

B = False

3757 On Dataset

3758 2) *ds\_r* := **not** population.sex="M"

3759

population				
SEX	AGE	GEO	TIME	CONDITION
M	Y_LT15	BE	2013	970428
M	Y15-64	BE	2013	3678355
M	Y_GE65	BE	2013	838653
F	Y_LT15	BE	2013	927644
F	Y15-64	BE	2013	3625561
F	Y_GE65	BE	2013	1121001
M	Y_LT15	UK	2013	5757444
M	Y15-64	UK	2013	20748657
M	Y_GE65	UK	2013	4917238
F	Y_LT15	UK	2013	5488356
F	Y15-64	UK	2013	20915924
F	Y_GE65	UK	2013	6068452

3760

3761

ds_r				
SEX	AGE	GEO	TIME	CONDITION
M	Y_LT15	BE	2013	false

M	Y15-64	BE	2013	false
M	Y_GE65	BE	2013	false
F	Y_LT15	BE	2013	true
F	Y15-64	BE	2013	true
F	Y_GE65	BE	2013	true
M	Y_LT15	UK	2013	false
M	Y15-64	UK	2013	false
M	Y_GE65	UK	2013	false
F	Y_LT15	UK	2013	true
F	Y15-64	UK	2013	true
F	Y_GE65	UK	2013	true

3762

3764 **extract**3765 *Semantics*

3766 The operator **extract** returns an integer that is part of a given date, based on the value assumed by the *part*  
3767 parameter.

3769 *Syntax*

3770 **extract**( *ds*, *part* )

3772 *Parameters*

3773 *ds* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as date}+  
3774 {attribute <IDENT> as scalar-type}\* | date]

3775 *part* : string

3776 *ds* – is the input Dataset or date.

3777 *part* – is the part of date (year, month or day) to extract.

3780 *Constraints*

- 3781 • *part* can assume a restricted number of values ("A", "S", "Q", "M", "W", "D").

3782 If *ds* is a Dataset, it must have only date Measure Component.

3783 *Returns*

3784 If *ds* is a date value then the **extract** operator returns an integer value that is the part of *ds* specified in the *part*  
3785 parameter.

3786 If *ds* is a Dataset then the **extract** operator returns a Dataset having all the Identifier, Measure and Attribute  
3787 Components of *ds*, where the Measure Components change the data in type (from date to integer) and assume  
3788 the values of part of the dates (on the base of the *part*) in the input Measure Components

3790 *Examples*

3791 On date

3792 1) If A = 28/02/2016

3793 B := **extract** (A, "Y") B = 2016

3794 B := **extract** (A, "M") B = 2

3795 On Dataset

3796 2) ds\_2:= **extract** (ds\_1, "Y")

3798

ds_1		
K1	K2	M1
1	A	2015/12/10
2	B	2016/06/11
3	C	2015/12/10
4	E	2013/06/11

3799

ds_2		
K1	K2	M1
1	A	2015
2	B	2016
3	C	2015
4	E	2013

3800

## 3801 string from date

### 3802 *Semantics*

3803 The operator **string\_from\_date** converts a date value into a string.

3804

### 3805 *Syntax*

3806 **string\_from\_date**( *ds*, *format* )

3807

### 3808 *Parameters*

3809 *ds* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as date}+  
3810 {attribute <IDENT> as scalar-type}\*|date]

3811 *format* : string-literal

3812

3813 date – is the input Dataset or date.

3814 format – is the format of the resulting string.

3815

### 3816 *Constraints*

- 3817 • If *ds* is a Dataset then it must have only one date Measure Component,
- 3818 • *format* must respect one of the following patterns:

3819

Format	Frequency	Example	Frequency
YYYY		2000	Annual
YYYYSN	S	2000S1	Semestrial
YYYYQN	Q	2000Q1	Quarterly
YYYYMNN	M	2000M01	Monthly
YYYYDNNNN	D	2000D0101	Daily
YYYYA	A	2000A	Annual
YYYYSN	S	2000S1	Semestrial
YYYY-QN	Q	2000-Q1	Quarterly
YYYY-NN	M	2000-01	Monthly
YYYY-NN-NN	D, M, Q or A	2000-01-01	Daily, Monthly, Quarterly or Annual

3820

### 3821 *Returns*

3822 If *ds* is a date then the operator returns a string representation of the input date, based on the chosen format.

3823 If *ds* is a Dataset then the operator returns a Dataset having all the Identifier, Measure and Attribute Components  
3824 of the *ds*, where the Measure Components change in the data type (from date to string-literal) and assume the  
3825 values of the string representations (on the base of the *format*) of the dates in the input Measure Components.

3826

### 3827 *Semantic specification*

3828 If the format does not conform to any of the formats expressed in the constraints section, then a runtime  
3829 exception is raised.

3830

### 3831 *Examples*

3832 On date

3833 1) If A = 28/02/2016

3834 B := **string\_from\_date** (A, "YYYY-MM")

3835 B = "2016-02"

3836 On Dataset

3837 2) ds\_2 := **string\_from\_date** (ds\_1, "YYYY-MM")

3838

ds\_1



K1	K2	M1
1	A	2015/12/10
2	B	2015/06/11
3	C	2015/12/10
4	E	2015/06/11

3839  
3840

ds_2		
K1	K2	M1
1	A	"2015-12"
2	B	"2015-06"
3	C	"2015-12"
4	E	"2015-06"

3841  
3842

## 3843 current\_date

### 3844 *Semantic*

3845 The operator **current\_date** returns the current date.

3846

### 3847 *Syntax*

3848 **current\_date()**

3849

### 3850 *Parameters*

3851 *None*

3852

### 3853 *Constraints*

3854 *None*

3855

### 3856 *Returns*

3857 A Dataset having only one date Measure Component, with only one single Data Point representing the current  
3858 date.

3859

3860

## VTL-ML - Set functions

3861

### union

3862

#### Semantics

3863

The operator **union** takes as input a list of Datasets and returns a single Dataset containing all the Data Points, without duplicates, that appear in any of them.

3864

3865

3866

#### Syntax

3867

**union** ( *ds* {, *ds*}\* {**dedup**(*consResFunction*) }?)

3868

3869

#### Parameters

3870

*ds* : dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as scalar-type}\*  
 {attribute <IDENT> as scalar-type}+

3871

3872

*consResFunctions* : list<component-ref \* (t\*t) -> t > (t is the type of the referred Component)

3873

3874

- *ds* – are the input Datasets.

3875

- *consResFunction* is a list of functions used to solve conflicts caused by the presence of Data Points with the same values for the Identifier Components.

3876

3877

3878

#### Constraints

3879

All the *ds* Datasets must have the same Identifier and Measure Components, in name and type (static).

3880

3881

#### Returns

3882

The operator allows to eliminate duplicates through *consResFunction*. If the resulting set of data contains duplicates then **union** generates a run-time error.

3883

3884

3885

#### Semantic specification

3886

The operator takes as input a list of Datasets and returns a Dataset with the same structure as the input one and containing all the Data Points from every *ds* without duplicates. The *consResFunction* allows the user to specify a strategy to eliminate duplicates. In particular, for any single n-uple of duplicate Data Points, the function is applied recursively so as to reduce the duplicates to one single Data Point. If only a Dataset is specified, then it is returned unchanged.

3890

3891

3892

1) *ds\_r* := **union**(*total\_population1*, *total\_population2*)

3893

3894

total_population1				
TIME	GEO	AGE	SEX	POPULATION
2012	Belgium	Total	Total	5
2012	Greece	Total	Total	2
2012	France	Total	Total	3
2012	Malta	Total	Total	7
2012	Finland	Total	Total	9
2012	Switzerland	Total	Total	12

3895

3896

3897

Total_population2				
TIME	GEO	AGE	SEX	POPULATION
2012	Netherlands	Total	Total	23

3898  
3899

2012	Greece	Total	Total	2
2012	Spain	Total	Total	5
2012	Iceland	Total	Total	1

3900  
3901  
3902

ds_r				
TIME	GEO	AGE	SEX	POPULATION
2012	Belgium	Total	Total	5
2012	Greece	Total	Total	2
2012	France	Total	Total	3
2012	Malta	Total	Total	7
2012	Finland	Total	Total	9
2012	Switzerland	Total	Total	12
2012	Netherlands	Total	Total	23
2012	Spain	Total	Total	5
2012	Iceland	Total	Total	1

2) ds\_r := **union**(total\_population1, total\_population2)

3903  
3904

total_population1				
TIME	GEO	AGE	SEX	POPULATION
2012	Belgium	Total	Total	1
2012	Greece	Total	Total	2
2012	France	Total	Total	3
2012	Malta	Total	Total	4
2012	Finland	Total	Total	5
2012	Switzerland	Total	Total	6

3905  
3906

total_population2				
TIME	GEO	AGE	SEX	POPULATION
2011	Belgium	Total	Total	10
2012	Greece	Total	Total	20
2012	France	Total	Total	30
2012	Malta	Total	Total	40
2012	Finland	Total	Total	50
2012	Switzerland	Total	Total	60

total_population1				
TIME	GEO	AGE	SEX	POPULATION
2012	Belgium	Total	Total	1
2012	Greece	Total	Total	2

2012	France	Total	Total	3
2012	Malta	Total	Total	4
2012	Finland	Total	Total	5
2012	Switzerland	Total	Total	6
2011	Belgium	Total	Total	10

3907  
3908  
3909  
3910

3) total\_population := **union** (total\_population1, total\_population2)

total_population1		
TIME	GEO	POPULATION
2012	Belgium	5
2012	Greece	2
2012	France	3
2012	Malta	7
2012	Finland	9
2012	Switzerland	12

3911

total_population2		
TIME	GEO	POPULATION
2012	Netherlands	23
2012	Greece	2
2012	Spain	5
2012	Iceland	1

3912

total_population		
TIME	GEO	POPULATION
2012	Belgium	5
2012	Greece	2
2012	France	3
2012	Malta	7
2012	Finland	9
2012	Switzerland	12
2012	Netherlands	23
2012	Spain	5
2012	Iceland	1

3913  
3914  
3915

4) time\_geo := **union** (time\_geo1, time\_geo2)

time_geo1	
TIME	GEO
2012	Belgium
2012	Greece

2012	France
2012	Malta
2012	Finland
2012	Switzerland

3916

time_geo2	
TIME	GEO
2012	Netherlands
2012	Greece
2012	Spain
2012	Iceland

3917

time_geo	
TIME	GEO
2012	Belgium
2012	Greece
2012	France
2012	Malta
2012	Finland
2012	Switzerland
2012	Netherlands
2012	Spain
2012	Iceland

3918

## 3919 intersect

### 3920 *Semantics*

3921 The operator **intersect** takes as input Datasets and returns another Dataset with the intersection of the input  
3922 Datasets.

3923

### 3924 *Syntax*

3925 **intersect** ( *ds* {, *ds*}\* {**dedup**(consResFunction) }?)

3926

### 3927 *Parameters*

3928 *ds* : dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as scalar-type}\*  
3929 {attribute <IDENT> as scalar-type}+

3930 *consResFunctions* : list<component-ref\* (t\*t) -> t > (t is the type of the referred Component)

3931

- 3932 • *ds* – are the input Datasets.
- 3933 • *consResFunction* is a list of functions used to solve conflicts caused by the presence of Data Points with the  
3934 same values for the Identifier Components.

3935

### 3936 *Constraints*

3937 All the Datasets *ds* must have the same Identifier and Measure Components, in name and type (static).

3938

### 3939 *Returns*

3940 A Dataset having the same Identifier, Measure and Attribute Components of the input ones, containing all the  
3941 Data Points that are present in every *ds*.

3942  
 3943  
 3944  
 3945  
 3946  
 3947  
 3948  
 3949  
 3950  
 3951  
 3952

*Semantic specification*

The operator takes as input Datasets and returns another one Dataset with the same structure of the input ones containing all the Data Points that are present in every *ds*, which is their intersection. If two Data Points appear in all the input Datasets, but with different values for the Measure Components, then the values for the Measures are determined by combining the input ones with a *consResFunction* that solves the conflicts.

*Examples*

`d_r := intersect(total_population1, total_population2)`

total_population1				
TIME	GEO	AGE	SEX	POPULATION
2012	Belgium	Total	Total	1
2012	Greece	Total	Total	2
2012	France	Total	Total	3
2012	Malta	Total	Total	4
2012	Finland	Total	Total	5
2012	Switzerland	Total	Total	6

3953

total_population2				
TIME	GEO	AGE	SEX	POPULATION
2011	Belgium	Total	Total	10
2012	Greece	Total	Total	2
2011	France	Total	Total	30
2011	Malta	Total	Total	40
2011	Finland	Total	Total	50
2011	Switzerland	Total	Total	60

3954

d_r				
TIME	GEO	AGE	SEX	POPULATION
2012	Greece	Total	Total	2

3955

3956 **symdiff**

*Semantics*

The operator **symdiff** takes as input two Datasets and returns another Dataset with the symmetric difference of the input Datasets.

*Syntax*

**symdiff** ( *ds\_1*, *ds\_2* )

*Parameters*

*ds\_1*, *ds\_2* : dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as scalar-type}\*  
 {attribute <IDENT> as scalar-type}+

3968  
 3969  
 3970

- *ds\_1* – is the first input Dataset.
- *ds\_2* – is the second input Dataset.

3971

3972 *Constraints*

3973 *ds\_1* and *ds\_2* must have the same Identifier and Measure Components in name and type (static).

3974

3975 *Returns*

3976 A Dataset having the same Identifier, Measure and Attribute Components of the input ones, containing all the  
3977 Data Points that are present either in *ds\_1* or in *ds\_2* but not in both.

3978

3979 *Semantic specification*

3980 The operator takes as input two Datasets and returns another one Dataset with the same structure of the input  
3981 ones containing all the Data Points that are present either in *ds\_1* or in *ds\_2* but not in both.

3982

3983 *Examples*

3984 `d_r := symdiff(total_population1, total_population2)`

total_population1				
TIME	GEO	AGE	SEX	POPULATION
2012	Belgium	Total	Total	1
2012	Greece	Total	Total	2
2012	France	Total	Total	3
2012	Malta	Total	Total	4
2012	Finland	Total	Total	5
2012	Switzerland	Total	Total	6

3985

total_population2				
TIME	GEO	AGE	SEX	POPULATION
2011	Belgium	Total	Total	1
2012	Greece	Total	Total	2
2012	France	Total	Total	3
2012	Malta	Total	Total	4
2012	Finland	Total	Total	5
2012	Switzerland	Total	Total	6

3986

d_r				
TIME	GEO	AGE	SEX	POPULATION
2012	Belgium	Total	Total	1
2011	Belgium	Total	Total	1

3987

3988 **setdiff**

3989 *Semantics*

3990 The operator **setdiff** takes as input two Datasets and returns another Dataset with the difference of the input  
3991 Datasets.

3992

3993 *Syntax*

3994 **setdiff** ( *ds\_1* , *ds\_2* )

3995

3996 *Parameters*

3997 *ds\_1* , *ds\_2* : dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as scalar-type}\*

3998 {attribute <IDENT> as scalar-type}+

3999

- 4000 • *ds\_1* – is the first input Dataset.
- 4001 • *ds\_2* – is the second input Dataset.

4002

#### 4003 *Constraints*

4004 *ds\_1* and *ds\_2* must have the same Identifier and Measure Components in name and type (static).

4005

#### 4006 *Returns*

4007 A Dataset having the same Identifier, Measure and Attribute Components of the input ones, containing all the  
4008 Data Points that are present in *ds\_1* but not in *ds\_2*.

4009

#### 4010 *Semantic specification*

4011 The operator takes as input two Datasets and returns another Dataset with the same structure as the input ones  
4012 containing all the Data Points that are present in either *ds\_1* but not in *ds\_2*, which is their difference.

4013

#### 4014 *Examples*

4015 1) *d\_r* := **setdiff** ( total\_population1,total\_population2)

4016

total_population1				
TIME	GEO	AGE	SEX	POPULATION
2012	Belgium	Total	Total	10
2012	Greece	Total	Total	20
2012	France	Total	Total	30
2012	Malta	Total	Total	40
2012	Finland	Total	Total	50
2012	Switzerland	Total	Total	60

4017

total_population2				
TIME	GEO	AGE	SEX	POPULATION
2011	Belgium	Total	Total	10
2012	Greece	Total	Total	20
2012	France	Total	Total	30
2012	Malta	Total	Total	40
2012	Finland	Total	Total	50
2012	Switzerland	Total	Total	60

4018

<i>d_r</i>				
TIME	GEO	AGE	SEX	POPULATION
2012	Belgium	Total	Total	10

4019

4020 2) DatasetC := **setdiff** (DatasetA ,DatasetB)

Dataset A			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	7
FR	Females	2011	10
FR	Total	2011	12

4021



Dataset B			
COUNTRY	SEX	YEAR	VALUE
FR	Males	2011	7
FR	Females	2011	10

4022

Dataset C			
COUNTRY	SEX	YEAR	VALUE
FR	Total	2011	12

4023

## 4024 subscript

### 4025 *Semantics*

4026 The operator **subscript** takes as input a Dataset and a sequence of Identifier Components with their respective  
 4027 values, and returns another Dataset having only the data points that contains the values specified in the  
 4028 subscript for the respective Identifier Component.

4029

### 4030 *Syntax*

4031 `ds [ comp = comp_value1 { , comp = comp_value2 } * ]`

4032

### 4033 *Parameters*

- 4034 • *ds* – is the input Dataset
- 4035 • *comp* – Dataset component-ref
- 4036 • *comp\_value1, comp\_value2* – is a valid value for component

4037

### 4038 *Constraints*

- 4039 • *comp* must be a valid Identifier of *ds* component.
- 4040 • *comp\_value1, comp\_value2* must be a valid value for the related component.

4041

### 4042 *Returns*

4043 A Dataset having the same Measure and Attribute Components as the input one, and all the Identifier  
 4044 Components that are not specified as parameters (*comp*). The Data points of the returned Dataset are all those of  
 4045 *ds* whose values having for the subscripted identifier component(s) coincide with the values specified in the  
 4046 subscript.

4047

### 4048 *Semantic specification*

4049 This operator removes identifiers components of the Dataset performing before a filter over the components  
 4050 values specified in the subscript. This avoids inconsistency on the returned Dataset.

4051

### 4052 *Examples*

4053 1) `ds_2 := ds_1 [ time = 2010, ref_area = EU25 ]`

4054

ds_1				
TIME	REF_AREA	PARTNER	OBS_VALUE	OBS_STATUS
2010	EU25	CA	20	D
2010	BG	CA	1	P
2010	RO	CA	1	P
2010	EU27	CA	23	P

4055

4056

ds_2		
PARTNER	OBS_VALUE	OBS_STATUS
CA	20	D

4057  
4058  
4059

2) ds\_2 := ds\_1 [time = 2010, ref\_area = EU25, partner = CA ]

ds_2	
OBS_VALUE	OBS_STATUS
20	D

4060  
4061  
4062

3) ds\_2 := ds\_1 [ref\_area = EU25 ] + ds\_1[ ref\_area = BG ] + ds\_1 [ ref\_area = RO ]

ds_2		
TIME	PARTNER	OBS_VALUE
2010	CA	22

4063  
4064  
4065

4) ds\_2 := ds\_1 [ time = 2010, ref\_area = EU25 ]

ds_1				
TIME	REF_AREA	PARTNER	OBS_VALUE	OBS_STATUS
2010	EU25	CA	20	D
2010	EU25	NF	1	P
2010	RO	CA	1	P
2010	EU27	CA	23	P

4066

ds_2		
PARTNER	OBS_VALUE	OBS_STATUS
CA	20	D
NF	1	P

4067

## 4068 transcode

### 4069 *Semantics*

4070 The **transcode** operator recodes the identifiers values using a map Dataset or a mapping object.

4071

### 4072 *Syntax*

4073 **transcode**(ds.comp, [ds\_map| mapping])

4074

### 4075 *Parameters*

4076 ds.comp : Component-ref

4077 ds : dataset {identifier <IDENT> as scalar-type}+

4078 {measure <IDENT> as scalar-type}\* {attribute <IDENT> as scalar-type}\*

4079 ds\_map : dataset {identifier **MAPS\_FROM** as scalar-type; } {measure **MAPS\_TO** as scalar-type; }

4080

- 4081 • ds.comp – is a valid Identifier Component of the Dataset.

- 4082 • *ds\_map* – is the Dataset that defines the mapping. It has an Identifier Component, **MAPS\_FROM**, that  
4083 specifies the values to be transformed and a Measure Component, **MAPS\_TO**, specifying the target value for  
4084 each of them.
- 4085 • *mapping* – a mapping, persistent object created with `define mapping`

4086  
4087 **Constraints**

4088 The following conditions guarantee that the resulting Dataset does not have duplicates:

- 4089 • All the values of the Measure Component **MAPS\_TO** must be distinct.
- 4090 • For each distinct value of the Identifier Component to be recoded, there is a value (and only one) in the  
4091 Identifier Component **MAPS\_FROM** in *ds\_map* or in the *mapping* object

4092  
4093 **Returns**

4094 A Dataset that has the same Identifier, Measure and Attribute Components as the input one. The values of the  
4095 Identifier Component are recoded into the corresponding values in the **MAPS\_TO** Measure Component of the  
4096 Dataset *ds\_map*.

4097  
4098 **Semantic specification**

4099 This operator allows to transform an input Dataset by mapping the values of one Identifier Component into  
4100 corresponding values, as specified by a mapping Dataset. Since the mapping Dataset is guaranteed to have one  
4101 distinct target value for each input one, and the input values appear only once, the resulting Dataset will contain  
4102 no duplicates.

4103 All the Data Points of *ds* are also present in the result and the values of the Identifier Component *ds.comp* are  
4104 modified as follows. For each data points of the Dataset, the value *v* of *ds.comp* is replaced by the value included in  
4105 the *ds\_map* or in the *mapping* corresponding to *v*.

4106  
4107 **Examples**

4108 `ds_2 := transcode( ds, ds_map, REF_AREA )`  
4109

ds_map	
MAPS_FROM	MAPS_TO
LU	LUX
BE	BEL
IT	ITA

4110

ds	
REF_AREA	VALUE
LU	10
BE	11
IT	13

4111

ds_2	
REF_AREA	VALUE
LUX	10
BEL	11
ITA	13

4112 **aggregate**

4113 **Semantics**

4114 The operator **aggregate** takes as input a Dataset and returns a new Dataset with the data aggregated based on  
4115 the rules and Boolean conditions specified in the hierarchical ruleset.

4116  
4117 *Syntax*  
4118 **aggregate** (*ds*, *hr*, { [**total** | **partial**] }, { [**return aggregates** | **return all data points**] } );

#### 4120 *Parameters*

4121 *ds*: dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as numeric}+  
4122 {measure <IDENT> as scalar-type}\* {attribute <IDENT> as scalar-type}\*  
4123

- 4124 • *ds* – is the input Dataset to aggregate.
- 4125 • *hr* – is the hierarchical ruleset (see define hierarchical ruleset) where the rules and the conditions to  
4126 perform the aggregate operation are defined.
- 4127 • *total* – a keyword to specify that the aggregation is performed only when all the elements in the right-hand  
4128 elements of the aggregation conditions in *vr* are not NULL (default behaviour).
- 4129 • *partial* – a keyword to specify that the aggregation is performed when at least one element of the right-hand  
4130 side of the aggregation conditions in *vr* is not NULL and, in this case, all the NULLs are treated as zero (that  
4131 is, ignored in the summation).
- 4132 • *return aggregates* – a keyword to specify that the output Dataset contains only the data points resulting from  
4133 aggregations (default behaviour).
- 4134 • *return all data points* – a keyword to specify that the output Dataset contains data points resulting from  
4135 aggregations as well as the data points of the input Dataset *ds*.

#### 4137 *Constraints*

- 4138 • *ds* must have at least one numeric Measure.
- 4139 • *hr* must be defined for calculation purposes (hence following the respective constraints).

#### 4141 *Returns*

4142 A Dataset with all the Identifier and Measure Components of *ds*, with the data aggregated on the basis of the  
4143 rules and Boolean conditions specified in the hierarchical (vertical) Ruleset *hr*.

#### 4145 *Semantic specification*

4146 The aggregate operator takes as input a Dataset, with at least a numeric Measure Component, and a hierarchical  
4147 ruleset and returns a new Dataset, with the data aggregated based on the rules and Boolean conditions specified  
4148 in the ruleset.

4149 The operator computes the numeric Measure Components associated to the aggregates defined in the left side of  
4150 the rules in *hr*. The aggregation is performed computing all aggregates in a single operation according to a  
4151 bottom-up calculation.

4152 The rules are executed in an appropriate order. In practice, if a rule in the ruleset depends on another one, the  
4153 latter is evaluated before, and its output exploited by the former. The functional constraints ensure that each  
4154 aggregate is calculated once.

4155 By default, the aggregation is performed only when all element of the right side of an aggregation rule in the  
4156 hierarchical (vertical) Ruleset of input *hr* are not NULL in the input Dataset *ds* (**total** clause). By specifying the  
4157 **partial** clause the aggregation is performed either if there are NULL values.

4158 The Dataset's data points that are not implied in the aggregation are not shown in the resulting Dataset,  
4159 essentially the data points containing values that are not involved in the aggregation will be lost (**return  
4160 aggregates** clause). Specifying the **return all data points** clause, the returned Dataset will contain also the  
4161 disaggregated data points of the input Dataset *ds*.

#### 4163 *Examples*

4164 In this example an aggregation is performed using the following hierarchical ruleset.

```
4165 define hierarchical ruleset hr_ref_area ( condition ( time ) rule ( ref_area ) ) is  
4166 EU15 = AT + BE + LU + DE + ES + FI + FR + EL + IE + IT + NL + PT + DK + UK + SE ;  
4167 EU25 = EU15 + CY + CZ + ES + HU + LT + LV + MT + PL + SK + SI ;  
4168 EU27 = EU25 + BG + RO ;  
4169 EU28 = EU27 + HR ;  
4170 when time between 1995 and 2003 then EU = EU15 ;  
4171 when time between 2004 and 2005 then EU = EU25 ;  
4172 when time between 2006 and 2012 then EU = EU27 ;  
4173 when time >= 2013 then EU = EU28  
4174 EEA15 = EU15 + IS + NO + LI ;  
4175
```

```

4176 EEA25 = EU25 + IS + NO + LI ;
4177 EEA27 = EU27 + IS + NO + LI ;
4178 EEA30 = EU27 + IS + NO + LI ;
4179 when time between 1995 and 2003 then EEA = EEA15 ;
4180 when time between 2004 and 2005 then EEA = EEA25 ;
4181 when time between 2006 and 2012 then EEA = EEA27 ;
4182 when time >= 2013 then EEA = EEA30 ;

```

```

4183 end hierarchical ruleset

```

```

4184
4185 The Dataset to aggregate:
4186

```

ds_bop				
TIME	REF_AREA	PARTNER	OBS_VALUE	OBS_STATUS
2010	EU25	CA	20	D
2010	BG	CA	1	P
2010	RO	CA	1	P
2010	EU27	CA	25	P
2010	HR	CA	2	P

```

4187
4188 1) ds_2 := aggregate( ds_bop, hr_ref_area );
4189

```

ds_2			
TIME	REF_AREA	PARTNER	OBS_VALUE
2010	EU27	CA	22
2010	EU28	CA	27

```

4190
4191 2) ds_2 := aggregate( ds_bop, hr_ref_area, return all data points );
4192

```

ds_2			
TIME	REF_AREA	PARTNER	OBS_VALUE
2010	EU25	CA	20
2010	BG	CA	1
2010	RO	CA	1
2010	EU27	CA	22
2010	HR	CA	2

```

4193
4194 3) In this example an aggregation is performed using the following hierarchical ruleset.
4195

```

```

4196 define hierarchical ruleset hr_ref_area ( condition ( time ) rule ( ref_area ) ) is
4197     when time = 2010 then EU= IT+BE+LU;
4198     when time = 2010 the AS=IN+CH;
4199 end hierarchical ruleset

```

```

4200
4201 ds_2 := aggregate(ds_bop, hr_ref_area);
4202

```

ds_bop				
TIME	REF_AREA	PARTNER	OBS_VALUE	OBS_STATUS
2010	IT	CA	2	P
2010	BE	CA	1	P
2010	LU	CA	1	P
2010	IN	CA	3	D
2010	CH		5	D

4203

ds_2				
TIME	REF_AREA	PARTNER	OBS_VALUE	OBS_STATUS
2010	EU	CA	4	P
2010	AS	CA	8	D

4204

4205

4206

4207

4208

4209

4210

4211

4212

4213

4214

4) In this example the Italian northern population has been obtained summing the population of nord\_east and nord-west (rule 1) and summing the population of all the regions being part of the nord (rule 2).

Hierarchical (vertical) Ruleset:

**define hierarchical ruleset** hr\_IT\_north\_pop (rule ( ref\_area ) ) is

ITCD = ITC + ITD;

ITCD =ITC1+ITC2+ITC3+ITC4+ITD1+ITD2

**end hierarchical ruleset**

ds\_2 := **aggregate**(IT\_nord\_bop, hr\_IT\_nord\_pop);

IT_nord_pop		
TIME	REF_AREA	OBS_VALUE
2015	ITCD	27799803
2015	ITC	16138643
2015	ITC1	4424467
2015	ITC2	128298
2015	ITC3	1583263
2015	ITC4	10002615
2015	ITD	11661160
2015	ITD1	518518
2015	ITD2	537416
2015	ITD3	4927596
2015	ITD4	1227122
2015	ITD5	4450508

4215

ds_2		
TIME	REF_AREA	OBS_VALUE
2015	ITCD	27799803

4216

4218 

## Aggregate functions

4219 

### Semantics

4220 VTL includes a set of statistical functions, that can be used to aggregate data.

4221

4222 

### Syntax

4223 `aggregateFunction ( ds {, other_parameters } ) { [ group by | along ] ( idComp {, idComp }* ) }`

4224

4225 

### Parameters

4226 `ds` : dataset {identifier <IDENT> as scalar-type; }+

4227 [ {measure &lt;IDENT&gt; as numeric}+ | {attribute &lt;IDENT&gt; as numeric}+ ]

4228 {measure &lt;IDENT&gt; as scalar-type}\* {attribute &lt;IDENT&gt; as scalar-type}\*

4229 `idComp` : component-ref

4230

- 4231 • `aggregateFunction` – is one of the aggregate functions described in the table below.
- 4232 • `other_parameters` – specific parameters additional to `ds`, related to the function used (see table **List of aggregate functions**)
- 4233 • `ds` – is the input Dataset to which the aggregate function is applied.
- 4234 • **group by** – represents the VTL groups data composed by the Identifier Components specified as `idComp`.
- 4235 • **along** – represents the VTL groups data composed by the Identifier Components of `ds` that are not specified as `idComp`. With the **along** clause the same VTL program can be reused for all Datasets that contain the Identifier Components specified in the **along** clause.
- 4236 • `idComp` – a component identifier of `ds`

4240

4241 

### Constraints

- 4242 • If `ds` has more than one Measure Component, then a Measure or attribute must be defined using the membership operator.
- 4243 • `idComp` must be a valid reference to an existing Identifier Component owned by `ds`.

4244

4245 

### Returns

4246 A Dataset having the Identifier Components of `ds` specified in the **group by** clause (or not specified in the **along** clause) and the Measure Components (or the implicit Measure Component deduced in a mono-Measure Dataset), with the data aggregated on the basis of the specific aggregate function and the partitions defined by **group by** or **along**.

4250

4251 

### Semantic specification

4252 An aggregate function groups together, evaluating a value that is specific for each aggregate function, the values of multiple data points having the same values of the specified Identifier Components `idComp`.

4253 The operator takes as input a Dataset, a Measure Component (specified with the membership operator on `ds`, or implicitly selected if `ds` has only one Measure component) on which the aggregate function will compute the result, and a sequence of Identifier Components `idComp` that will be used for the partitioning (the aggregate function is then applied separately for each partition). It returns another Dataset having the Identifier Components of `ds`, specified in the **group by** or **along**, and the Measure Component used for the aggregation. The other Identifier Components are removed from the resulting Dataset.

4257 If neither a **group by** or **along** clause is specified, then the aggregate function returns a single Data Point that has zero Identifier Components and only one Measure Component that is the one specified for the aggregation with the membership operator (or deduced from `ds`).

4258 Most of the aggregate functions can be also used as analytic functions (with a different syntax): See analytic functions.

4265

4266

4267

4268

List of aggregate functions	
Aggregate function	Description
<b>avg</b> ( <i>ds_1</i> )	average value of the not null values of <i>ds_1</i>
<b>corr</b> ( <i>ds_1</i> , <i>ds_2</i> )	Coefficient of correlation of ( <i>ds_1</i> , <i>ds_2</i> )
<b>covar_pop</b> ( <i>ds_1</i> , <i>ds_2</i> )	population covariance of ( <i>ds_1</i> , <i>ds_2</i> )
<b>covar_samp</b> ( <i>ds_1</i> , <i>ds_2</i> )	sample covariance of ( <i>ds_1</i> , <i>ds_2</i> )
<b>count</b> ( <i>ds_1</i> )	number of non-empty data points of <i>ds_1</i>
<b>median</b> ( <i>ds_1</i> )	median value of the not null values of <i>ds_1</i>
<b>min</b> ( <i>ds_1</i> )	minimum value of <i>ds_1</i>
<b>max</b> ( <i>ds_1</i> )	maximum value of <i>ds_1</i>
<b>percentile_cont</b> ( <i>ds_1</i> , constant) order by expression [ <b>asc</b>   <b>desc</b> ]	inverse distribution function that assumes a continuous distribution model
<b>percentile_disc</b> ( <i>ds_1</i> , constant) order by expression [ <b>asc</b>   <b>desc</b> ]	inverse distribution function that assumes a discrete distribution model
<b>rank</b> ( <i>ds_1</i> )	rank of a value in a group of values
<b>regr_slope</b> ( <i>ds_1</i> , <i>ds_2</i> )	linear regression (slope of the line)
<b>regr_intercept</b> ( <i>ds_1</i> , <i>ds_2</i> )	linear regression (y-intercept)
<b>regr_count</b> ( <i>ds_1</i> , <i>ds_2</i> )	linear regression (count non-null number pairs)
<b>regr_r2</b> ( <i>ds_1</i> , <i>ds_2</i> )	linear regression (coefficient of determination)
<b>regr_avgx</b> ( <i>ds_1</i> , <i>ds_2</i> )	linear regression (average of independent variable <i>ds_2</i> )
<b>regr_avgy</b> ( <i>ds_1</i> , <i>ds_2</i> )	linear regression (average of dependent variable <i>ds_1</i> )
<b>regr_sxx</b> ( <i>ds_1</i> , <i>ds_2</i> )	linear regression (auxiliary function)
<b>regr_syy</b> ( <i>ds_1</i> , <i>ds_2</i> )	linear regression (auxiliary function)
<b>regr_sxy</b> ( <i>ds_1</i> , <i>ds_2</i> )	linear regression (auxiliary function)
<b>stddev_pop</b> ( <i>ds_1</i> )	population standard deviation of <i>ds_1</i>
<b>stddev</b> ( <i>ds_1</i> )	standard deviation of <i>ds_1</i>
<b>sum</b> ( <i>ds_1</i> )	sum of values of <i>ds_1</i>
<b>var_pop</b> ( <i>ds_1</i> )	population variance of <i>ds_1</i>
<b>var_samp</b> ( <i>ds_1</i> )	sample variance of <i>ds_1</i>
<b>variance</b> ( <i>ds_1</i> )	variance of <i>ds_1</i>



4269  
4270  
4271  
4272

*Examples*

1) ds\_agg := **avg ( ds\_bop.obs\_value ) group by time**

ds_bop				
TIME	REF_AREA	PARTNER	OBS_VALUE	OBS_STATUS
2010	EU25	CA	20	
2010	BG	CA	1	
2010	RO	CA	1	
2010	EU27	CA	23	
2011	EU25	CA	20	P
2011	BG	CA	1	P
2011	RO	CA	-1	P
2011	EU27	CA	20	P
2012	LU	CA	40	P
2012	EU25	CA	30	P

4273

ds_agg	
TIME	OBS_VALUE
2010	11.25
2011	11.25
2012	30

4274  
4275  
4276  
4277  
4278  
4279  
4280

Note: the example above can be rewritten equivalently in the following forms:

ds\_agg := **avg ( ds\_bop ) along ref\_area, partner**

ds\_agg := **avg ( bop ) group by time**

2) ds\_agg := **avg ( ds\_bop.obs\_value ) group by time, ref\_area**

ds_agg		
TIME	REF_AREA	OBS_VALUE
2010	EU25	20
2010	BG	1
2010	RO	1
2010	EU27	23
2011	EU25	20
2011	BG	1
2011	RO	-1
2011	EU27	20
2012	LU	40
2012	EU25	30

4281  
4282  
4283

4284 3) ds\_agg := **avg** ( ds\_bop )  
 4285

ds_agg
<b>OBS_VALUE</b>
15.5

4286  
 4287 4) ds\_agg := **max** ( ds\_bop.obs\_value ) as max\_value, **min** ( ds\_bop.obs\_value ) as min\_value **group by** time  
 4288

ds_agg			
TIME	REF_AREA	MIN_VALUE	MAX_VALUE
2010	EU25	23	1
2011	RO	20	-1
2012	EU25	40	30

4289

## 4290 Time aggregate functions

### 4291 *Semantics*

4292 The *time aggregate functions* represent a set of statistical functions used to aggregate data on the time  
 4293 dimension.

### 4294 *Syntax*

```
4295 aggregateFunction ( ds
4296     , freqFrom
4297     , freqTo
4298     { , minPeriods }
4299     { , timePeriodName }
4300     { , timeFormatFrom }
4301     { , timeFormatTo }
4302 )
4303
```

### 4304 *Parameters*

4305 ds : dataset {identifier <IDENT> as scalar-type}+  
 4306 [ {measure <IDENT> as numeric}+ | {attribute <IDENT> as numeric}+ ]  
 4307 {measure <IDENT> as scalar-type}\* {attribute <IDENT> as scalar-type}\*  
 4308

- 4309 • *aggregateFunction* - is one of the aggregate functions described in the paragraph "Aggregate functions".
- 4310 • *ds* - is the input Dataset to which the aggregate function is applied.
- 4311 • *freqFrom* - is the frequency from which the data will be aggregated
- 4312 • *freqTo* - is the frequency to which the data will be aggregated. *freqTo* must be a lower frequency than
- 4313 *freqFrom*
- 4314 • *minPeriods* - is an Integer number describing the minimum number of values required to perform the time
- 4315 based aggregation. If *minPeriods* is omitted then the aggregation is performed only if all the periods needed
- 4316 for the aggregation are present.
- 4317 • *timePeriodName* - is the name of the time period component of the Dataset. Default name is "time".
- 4318 • *timeFormatFrom* - is the format of the time period relative to *freqFrom*. It must be specified only when
- 4319 *freqFrom* is C (custom).
- 4320 • *timeFormatTo* - is the format of the time period relative to *freqTo*. It must be specified only when *freqTo* is C
- 4321 (custom).
- 4322

### 4323 *Constraints*

- 4324 • If *ds* has more than one Measure Component, then a Measure or attribute must be specified using the
- 4325 membership operator on ds.
- 4326 • *timePeriodName* is the name of an Identifier Component owned by *ds*.
- 4327

- 4328 • *freqFrom* must be a higher frequency compared with *freqTo*, e.g. *freqFrom* = “M” and *freqTo* = “Q” is correct,  
4329 while the reverse is not correct.

4330

4331

Possible values for *freqFrom* and *freqTo*:

Frequency symbol	Frequency
A	Annual
S	Semestrial
Q	Quarterly
M	Monthly
W	Weekly
D	Daily

4332

#### 4333 Returns

4334 A Dataset with the Identifier Components of *ds* and a Measure Component containing data of *ds* aggregated from  
4335 *freqFrom* to *freqTo*. All aggregate functions can be used.

4336

#### 4337 Semantic specification

4338 The aggregateFunction first partitions the data set in groups of data having the frequency *freqFrom*, then the  
4339 data are aggregated to obtain data aggregated having the frequency *freqTo*.

4340 Convert the data contained in *ds* and having the time format specified in *freqFrom* to the format specified in  
4341 *freqTo*. If *freqFrom* and *freqTo* have different value then an aggregation could occur.

4342

4343 Data in *ds* having the frequency different from *freqFrom* will not be involved by the operator and they will be  
4344 discarded from the output Dataset.

4345

4346 By default this operator computes the aggregated value only when the values of all sub-periods exist.

4347 To override this behaviour the user can specify a value for the optional argument *minPeriods*: this is a lower  
4348 bound for the number of periods that must exist in order to produce the aggregation. This means that if the  
4349 optional parameter *minPeriods* is present, then the aggregation can be performed either if the timeseries Dataset  
4350 of input does not contain all the necessary DataPoints needed.

4351

4352

Time format:

Format	Frequency	Example	Frequency	Possible values
yyyy yyyyA	A	2000	Annual	yyyy = 1900, ..., 9999
yyyySs yyyy-Ss	S	2000S1	Semestrial	s = 1, 2
yyyyQq	Q	2000Q1	Quarterly	q = 1, 2, 3, 4
yyyy-Qq	Q	2000Q1	Quarterly	q = 1, 2, 3, 4
yyyyMmm yyyy-mm	M	2000M01	Monthly	mm = 01, 02, ..., 12
yyyyWnn	W	2000D0101	Weekly	nn = 01, 02, ..., 52
yyyyMmmDdd	D	2000D0101	Daily	mm = 01, 02, ..., 12 dd = 01, 02, ..., 31
yyyy-mm-dd	N	2000-01-01	Date (frequency not specified)	mm = 01, 02, ..., 12 dd = 01, 02, ..., 31
Combination of yyyy, mm and dd	C	01-01-2000	Custom date format	yyyy = 1900, ..., 9999 mm = 01, 02, ..., 12 dd = 01, 02, ..., 31

4353

4354

4355 *Examples*

4356 1) ds\_abop := sum ( bp\_qbop) time\_aggregate ( "Q", "A" )

ds_qbop			
TIME	REF_AREA	PARTNER	OBS_VALUE
2010Q1	EU25	CA	20
2010Q2	EU25	CA	20
2010Q3	EU25	CA	20
2010Q4	EU25	CA	20
2010Q1	EU27	CA	30
2010Q2	EU27	CA	30
2010Q3	EU27	CA	30
2010Q4	EU27	CA	30
2010Q1	IT	CA	10
2010Q2	IT	CA	10
2010Q3	IT	CA	10
2010Q4	IT	CA	10

4357

4358 The above operation perform a frequency change from quarterly data, where the date has the pattern "YYYYQN",  
 4359 to annual data with the pattern "YYYY". Due to the pattern of the Component TIME, the frequency is deduced and  
 4360 the *frequency\_name* parameter can be omitted.  
 4361

ds_abop			
TIME	REF_AREA	PARTNER	OBS_VALUE
2010	EU25	CA	80
2011	EU27	CA	120
2010	IT	CA	40

4362

ds_1	
DATE	VALUE
1939-01-01	4400.0
1939-02-01	4400.0
1939-03-01	10600.0
1939-04-01	6800.0

4363

4364 2) ds\_2 := sum ( ds\_1, "M", "A", 4, date, "yyyy-mm-dd", "yyyy-mm-dd" )

4365

4366 Due to the value of *minPeriods* ( 4 ), the annual aggregation is performed when at least 4 data points exist  
 4367 relative to the 1939's months.

ds_2	
DATE	VALUE
1939-01-01	26200.0

4368

4370 

## Analytic functions

4371 

### Semantics

4372 The Analytic functions allow to specify operations to be applied on groups of Data Points within a Dataset. The  
 4373 Data Points of the Dataset are first partitioned into groups. Then, in each group, each Data Point is combined with  
 4374 (some of) the others in a customizable way, and for each input Data Point, an output one is produced.

4375 Groups are determined by a list of names of Identifier Components, in such a way that Data Points having the  
 4376 same values for those Identifiers are assigned to the same group.

4377 A **sliding window** is then declared to define for each input Data Point in the window (**current** Data Point), how  
 4378 to produce the corresponding Data Point in the output, by combining its Measure Components with the ones of  
 4379 the other Data Points in the same window. For each window and for each Data Point, the sliding window spans  
 4380 the Data Points to be combined and while moving from the first to the last Data Point in the group, produces the  
 4381 output Data Points. In other words, for each group, the sliding window determines the moving range of Data  
 4382 Points to be combined for each input one. At one extreme, the sliding window can span one single Data Point at a  
 4383 time, implying that Data Points are not combined with the others, producing independent values of the Measures  
 4384 for all the input Data Points; at the other extreme, the sliding window spans the entire window, producing the  
 4385 same value for all the Data Points.

4386 The size of the sliding window is either based on the number Data Points to be included or the specification of a  
 4387 numerical interval.

4388 Finally, there are a number of possible functions that can be applied to combine the Data Points within a sliding  
 4389 window, such as the average value within a sliding window, the cumulative sum, and so on.

4390

4391 

### Syntax

4392 `analyticFunction ( ds { , extraParams } ) over ( { partitionBy } { orderBy } { windowingClause } )`

4393 `partitionBy ::= partition by c_p { , c_p } *`

4394 `orderBy ::= order by c_o { , c_o } * { [ asc | desc ] }`

4395 `windowingClause ::=`

4396 `[ rows | range ]`

4397 `between`

4398 `[ num preceding | num following | current row | unbounded preceding | unbounded following ]`

4399 `and`

4400 `[ num preceding | num following | current row | unbounded preceding | unbounded following ]`

4401

4402 

### Parameters

- 4403 • *ds* – is the input Dataset.
- 4404 • *extraParams* – additional parameters (depending on the analytic function).
- 4405 • *partitionBy* – partitions *ds* into groups based on the value of one or more Identifier Components. If omitted,  
 4406 the function treats all rows of the Dataset as a single partition.
- 4407 • *c\_p* – are valid Identifier Components of *ds* used for the partitioning expressed by *partitionBy*.
- 4408 • *orderBy* - specifies how Data Points are ordered within each windows (asc is the default).
- 4409 • *c\_o* – are references to valid Components on which the sort is performed within the respective pre-calculated  
 4410 windows.
- 4411 • The keywords *rows* and *range* define for each row a "sliding window" (set of rows) used for calculating the  
 4412 result of the analytic function. The analytic function is then applied to all the Data Points in the sliding  
 4413 window. The sliding window "slides" through the windows from top to bottom. In particular:
  - 4414 ○ *rows* – defines a sliding window using a specified number of preceding and following data points  
 4415 relative to the current data point (according to the orderBy clause)
  - 4416 ○ *range* – defines a sliding window as a numerical offset relative to the current data point (according to  
 4417 the orderBy clause)
- 4418 • *unbounded preceding* - indicates that the sliding window starts at the first Data Point of the window.
- 4419 • *unbounded following* - indicates that the sliding window ends at the last Data Point of the window.
- 4420 • *current row* – specifies that the window starts or ends at the current Data Point.
- 4421 • *preceding* – specifies the start point of the sliding window as number of data points preceding the current  
 4422 data point.
- 4423 • *following* – specifies the end point of the sliding window as number of data points following the current data  
 4424 point.

4425

4426 *Constraints*

- 4427 • *ds* must have at least one Measure Component (as explicated in the syntax).
- 4428 • *Analytic functions* cannot be nested.

4429 *Returns*

4430 A Dataset having the same Identifier, Measure and Attribute Components as the input one, where the Measure  
4431 Components take values depending on the definition of windows, sliding windows and the specific **analytic**  
4432 **function**.

4433 *Semantic specification*

4434 The operator takes a Dataset as input, optionally the specification for the partitioning and the internal order of  
4435 the windows; optionally, it also takes as input the information to define a sliding window. If omitted, there is one  
4436 single sliding window, coinciding with each of the windows. The operator returns a Dataset with the same  
4437 Identifier Components, Measure Components and Attribute Components as *ds*, where the value of all the  
4438 Measure Components take values that depend on the specific **analytic function**, the partitioning criteria and the  
4439 Data Points in each sliding window.

4440 The functions that can be used as analytic functions are the aggregate functions described in the previous  
4441 chapter and some specific functions described below.

4442 *first\_value*

4443 *Semantics*

4444 For each sliding window and for each Measure Component, the operator calculates the first value according to  
4445 the specified order.

4446 *Examples*

4447 `ds := first_value ( ds_bop ) over ( partition by ref_area , partner order by time )`

4448

ds_bop			
REF_AREA	PARTNER	TIME	OBS_VALUE
LU	CA	1993	3
LU	CA	1994	4
LU	CA	1996	10
LU	CA	1997	20
LU	US	1993	400
LU	US	1996	500
LU	US	1997	600
LU	WORLD	1994	1000

4449

ds			
REF_AREA	PARTNER	TIME	OBS_VALUE
LU	CA	1993	3
LU	CA	1994	3
LU	CA	1996	3
LU	CA	1997	3
LU	US	1993	400
LU	US	1996	400
LU	US	1997	400
LU	WORLD	1994	1 000

4450

4451

4457 lag lead

4458

4459 *Semantics*

4460 The operator swaps the values of all Measure Components of the current Data Point with the ones of the  
4461 corresponding Measure Components of the Data Point that is referred to by the offset. If the offset exceeds the  
4462 boundaries of the sliding window, the default value is used for the swap. If omitted, 0 is implied as the default.

4463

4464 *Parameters*

4465 This analytic function takes as input the following extra parameters:

- 4466 • *offset* - it allows to individuate a Data Point by specifying the relative position from the current Data Point as  
4467 an offset, negative if moving towards from the beginning to the end of the define ordering, positive if moving  
4468 from the end to the beginning of the defined order.
- 4469 • *default* - the value that a Data Point has to take if the Data Point, whose position is calculated using the offset,  
4470 is NULL.

4471

4472 *Examples*

4473

4474 `ds := lag ( ds_bop , -1, -100 ) over ( partition by ref_area , partner order by time )`

4475

ds_bop			
REF_AREA	PARTNER	TIME	OBS_VALUE
LU	CA	1993	3
LU	CA	1994	4
LU	CA	1996	10
LU	CA	1997	20
LU	US	1993	400
LU	US	1996	500
LU	US	1997	600
LU	WORLD	1994	1000

4476

ds			
REF_AREA	PARTNER	TIME	OBS_VALUE
LU	CA	1993	-100
LU	CA	1994	3
LU	CA	1996	4
LU	CA	1997	10
LU	US	1993	-100
LU	US	1996	400
LU	US	1997	500
LU	WORLD	1994	-100

4477

4478 last\_value

4479

4480 *Semantics*

4481 For each sliding window and for each Measure Component, the operator calculates the first value according to  
4482 the specified order.

4483

4484 *Examples*

4485

4486 `ds := last_value ( ds_bop ) over ( partition by ref_area , partner order by time )`

4487

ds_bop			
REF_AREA	PARTNER	TIME	OBS_VALUE
LU	CA	1993	3
LU	CA	1994	4
LU	CA	1996	10
LU	CA	1997	20
LU	US	1993	400
LU	US	1996	500
LU	US	1997	600
LU	WORLD	1994	1000

4488

ds			
REF_AREA	PARTNER	TIME	OBS_VALUE
LU	CA	1993	20
LU	CA	1994	20
LU	CA	1996	20
LU	CA	1997	20
LU	US	1993	600
LU	US	1996	600
LU	US	1997	600
LU	WORLD	1994	1 000

4489

4490

4491

4492

In spite of the general syntax for analytic functions, the following ones do not allow the definition of any sliding window, that is, it always coincide with the entire window.

4493

### ntile

4494

4495

#### Semantics

4496

4497

4498

4499

4500

4501

For each Data Point of each window, the operator produces a Data Point where the values of the numeric Measures Components are set to a unique window number. The values of the non-numeric Measure Components are just copied. For each windows a unique number (incrementally generated, starting with 1) is assigned. Note that the order by clause of analytic functions operates within each window; therefore, the windows are not mutually ordered.

4502

#### Examples

4503

4504

4505

ds := ntile ( ds\_bop ) over ( partition by REF\_AREA, partner order by time )

ds_bop			
REF_AREA	PARTNER	TIME	OBS_VALUE
LU	CA	1993	3
LU	CA	1994	4
LU	CA	1996	10
LU	CA	1997	20
LU	US	1993	400



LU	US	1996	500
LU	US	1997	600
LU	WORLD	1994	1000

4506

ds			
REF_AREA	PARTNER	TIME	OBS_VALUE
LU	CA	1993	1
LU	CA	1994	1
LU	CA	1996	1
LU	CA	1997	1
LU	US	1993	2
LU	US	1996	2
LU	US	1997	2
LU	WORLD	1994	3

4507

4508

### percent\_rank

4509

4510

#### Semantics

4511

4512

4513

4514

4515

4516

The operator calculates the percent rank of each Data Point with respect to the other Data Points of the same window. For each Data Point and for each numeric Measure Component, the percent rank is calculated as the rank of that Data Point minus one divided by the number of total Data Points in the partition. Data Points with equal values for the ranking criteria receive the same percent rank. The values of each numeric Measure Component are assigned to the respective percent rank. All the values of the non-numeric Measure Components are just copied.

4517

#### Examples

4518

ds := **percent\_rank** ( ds\_bop ) **over** ( **partition by** ref\_area, partner **order by** time )

4519

ds_bop			
REF_AREA	PARTNER	TIME	OBS_VALUE
LU	CA	1993	3
LU	CA	1994	4
LU	CA	1996	10
LU	CA	1997	20
LU	US	1993	400
LU	US	1996	500
LU	US	1997	600
LU	WORLD	1994	1000

4520

ds			
REF_AREA	PARTNER	TIME	OBS_VALUE
LU	CA	1993	0
LU	CA	1994	0.25
LU	CA	1996	0.5
LU	CA	1997	0.75

LU	US	1993	0
LU	US	1996	0.33
LU	US	1997	0.67
LU	WORLD	1994	0

4521  
4522  
4523  
4524  
4525  
4526  
4527  
4528  
4529  
4530  
4531

## rank

### Semantics

The operator calculates the rank of each Data Point with respect to the other Data Points in the same window.

For each Data Point and for each numeric Measure Component, the rank is calculated as the relative position of the Data Point in the window. The values of each numeric Measure Component is assigned to the respective rank. All the values of the non-numeric Measure Components are just copied.

### Examples

`ds_1 := rank ( ds_bop ) over ( partition by ref_area, partner order by time )`

ds_bop			
REF_AREA	PARTNER	TIME	OBS_VALUE
LU	CA	1993	3
LU	CA	1994	4
LU	CA	1996	10
LU	CA	1997	20
LU	US	1993	400
LU	US	1996	500
LU	US	1997	600
LU	WORLD	1994	1000

4532

ds_1			
REF_AREA	PARTNER	TIME	OBS_VALUE
LU	CA	1993	1
LU	CA	1994	2
LU	CA	1996	3
LU	CA	1997	4
LU	US	1993	1
LU	US	1996	2
LU	US	1997	3
LU	WORLD	1994	1

4533  
4534  
4535  
4536  
4537  
4538

## ratio\_to\_report

### Semantics

The operator calculates the percentage amount of the value of each Data Point in the respective window (ratio to report).

For each Data Point and for each numeric Measure Component, the ratio to report is calculated as the percentage amount of the value of Measure Component in the sum of the values for the same Measure Component of the other Data Points in the window. The values of each numeric Measure Component is assigned to the respective ratio to report. All the values of the non-numeric Measure Components are just copied.

4542

4543 *Examples*  
 4544 `ds_1 := ratio_to_report ( ds_bop ) over ( partition by REF_AREA, partner)`  
 4545

ds_bop			
REF_AREA	PARTNER	TIME	OBS_VALUE
LU	CA	1993	3
LU	CA	1994	4
LU	CA	1996	10
LU	CA	1997	20
LU	US	1993	400
LU	US	1996	500
LU	US	1997	600
LU	WORLD	1994	1000

4546

ds_1			
REF_AREA	PARTNER	TIME	OBS_VALUE
LU	CA	1993	0.08108
LU	CA	1994	0.10810
LU	CA	1996	0.27027
LU	CA	1997	0.54054
LU	US	1993	0.26667
LU	US	1996	0.33333
LU	US	1997	0.40000
LU	WORLD	1994	0.10000

4547

## 4548 hierarchy

### 4549 *Semantics*

4550 VTL foresees the existence of set relations among Code Items in Code List.

4551 Many Enumerated Value Domains have an intrinsic Boolean algebraic structure, in the sense that a Boolean algebra can be defined on the respective Code Items.

4552 In general, a Boolean algebra is an algebraic structure (elements and operators having some properties) that summarizes the properties of set operators (*union, intersection, complement*) and logical operators (or, and, not).

4553 Elements of a Boolean Value Domain can be combined with the elementary set operators<sup>3</sup>; e.g. the item C is the union of the items D and E; the item K is the complement of S with respect to J (so the elements in J that are not in S), the item A is the intersection of B and C and so on.

4554 Only considering the set union, there are two possible organizations for hierarchical Code Lists: classifications and free hierarchies.

4555 In **classifications**, every element is uniquely classified by a single partition, so that the overall structure is a tree and, consequently, every Code Item can be given a specific level. An example is the usual geographical classification of the world into continents, each partitioned into nations.

4562

<sup>3</sup> Here we refer to elementary set operations and not to any operator of the language.

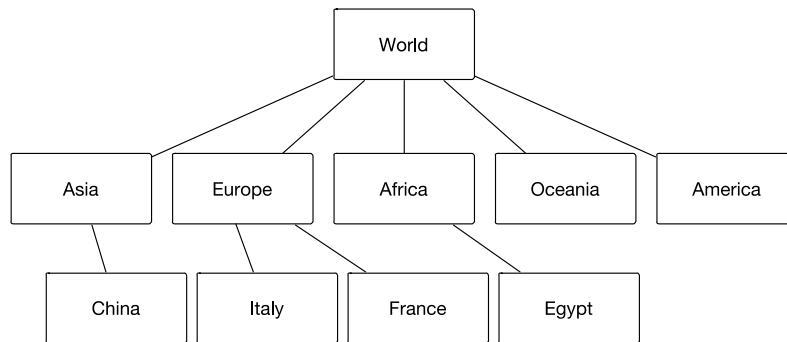
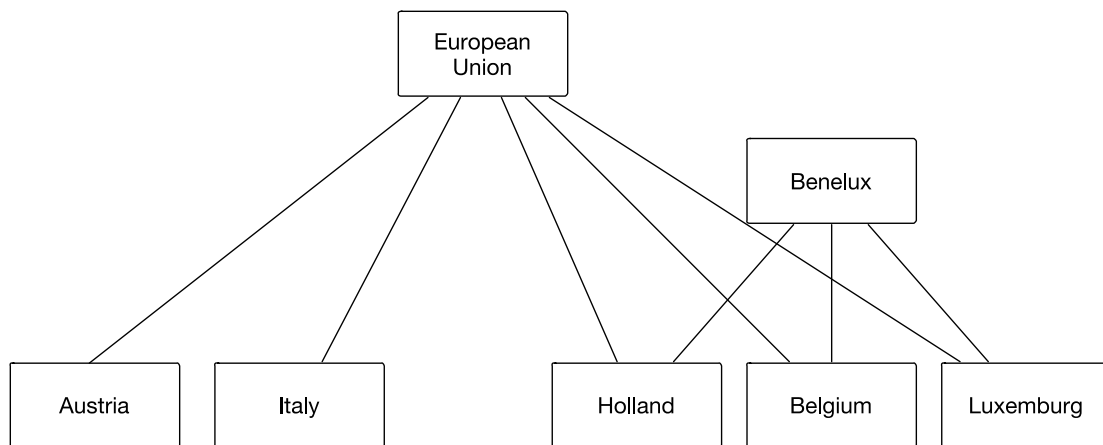


Figure 1 - example of hierarchical Code List

4563  
4564

4565  
4566  
4567  
4568  
4569  
4570

In **free hierarchies** each item can be partitioned according to multiple criteria; in turn, every element can be used to compose multiple other elements. The overall structure is not a tree and isolated Items can be present. An example is the hierarchy of European countries, where Belgium, Holland and Luxembourg contribute to the “European Countries” Code Item and to the “Benelux” Code Item.



4571  
4572

4573

Figure 2 - Example of free hierarchy

4574 More sophisticated hierarchical organizations can indeed exist if intersection and complement set operators are  
4575 also considered. For example the element “Benelux” could also be defined as the complement of the European  
4576 Union with respect to all the countries except Holland, Belgium and Luxembourg.

4577 Therefore we would have:

4578  $BENELUX = EU - (ITALY \cup AUSTRIA \cup \dots)$

4579 In order to support multiple classifications of Code Items in Code Lists and allow for the adoption of all the set  
4580 operators, we introduce two concepts *hierarchical aggregations* and *mappings* (that will be used in the  
4581 **hierarchy** operator).

4582 A *hierarchical aggregation* is a set of *mappings*, each transforming a Code Item into another Code Item. All the  
4583 *mappings* within a *hierarchical aggregation* associate Code Items of the same Code List with Code Items of a  
4584 single Code List.

4585 Hierarchical aggregations and mappings can be easily expressed in tabular form and referred to in the language  
4586 (in **hierarchy** operator) by an identifier (*hierarchyName*). However, there is also an inline syntactical form.

4587 Suppose we want to express the hierarchical relationship  $BENELUX = Belgium \cup Holland \cup Luxembourg$ . There  
4588 will be a hierarchical aggregation **Benelux\_aggr**, with the following mappings:

4589

HIERARCHY NAME	MAPS FROM	MAPS TO	START DATE	END DATE	SIGN	LEVEL	OUTPUT
<b>Benelux_aggr</b>	Belgium	Benelux	...	...	+	1	true
<b>Benelux_aggr</b>	Holland	Benelux	...	...	+	1	true
<b>Benelux_aggr</b>	Luxembourg	Benelux	...	...	+	1	true

4590  
4591  
4592  
4593  
4594  
4595  
4596  
4597  
4598

It maps each component item into the compound one.  
Each mapping has a *Sign*. It specifies if the contribution of the MAPS FROM Code Item in the composition is positive (UNION) or negative (COMPLEMENT). Notice that there is not a particular convention to represent intersection, as it can be obtained with an appropriate composition of UNION and COMPLEMENT.  
For instance, if we want to define in the element **EuropeWithoutItaly**, a possible definition could be the one complementing Italy, with respect to the entire Europe (i.e. subtracting Italy from Europe), as shown in the following table:

HIERARCHY NAME	MAPS FROM	MAPS TO	START DATE	END DATE	SIGN	LEVEL	OUTPUT
Eu_no_Italy_aggr	Europe	EuropeWith outItaly	....	...	+	1	true
Eu_no_Italy_aggr	Italy	EuropeWith outItaly	...	...	-	1	true

4599  
4600  
4601  
4602  
4603  
4604  
4605

Moreover, mappings are divided into levels, in the sense that a complete tree can be embedded into one hierarchy. The OUTPUT property, for each MAPS TO, indicates if the value must be preserved in the output or is only to be used for aggregations at a higher level.  
For instance, suppose we want to express the hierarchical relationship in Figure2, there will be a hierarchy **World\_aggr**, with the following correspondences:

HIERARCHY NAME	MAPS FROM	MAPS TO	START DATE	END DATE	SIGN	LEVEL	OUTPUT
World_aggr	Italy	Europe	....	...	+	1	false
World_aggr	France	Europe	...	...	+	1	false
World_aggr	Luxembourg	Europe	...	...	+	1	false
World_aggr	...	Europe			+	1	false
World_aggr	Asia	World			+	2	true
World_aggr	Europe	World			+	2	true
World_aggr	America	World			+	2	true
World_aggr	Oceania	World			+	2	true
World_aggr	Africa	World			+	2	true

4606  
4607  
4608  
4609  
4610  
4611  
4612  
4613  
4614  
4615  
4616  
4617  
4618  
4619  
4620  
4621  
4622  
4623  
4624

Output *false* for LEVEL 1, indicates that the aggregations into Europe (and into the other countries) are only functional to LEVEL 2 and the MAPS TO values with *false* will not be in the output.  
Summarizing, the set of mappings within a hierarchy has to be interpreted as follows.  
For every level, from the lowest to the highest, each MAPS TO Code Item is the set difference between the UNION of all the corresponding MAPS FROM Code Items with positive SIGN and the UNION of all the corresponding MAPS FROM Code Items with the negative SIGN.  
*Rules* inherently represent hierarchies in Code Lists, but, at the same time, only refer to Code Items. Thus, can be applied on different *Identifier Components* referring to different Code Lists.

*Syntax*

```

hierarchy
  ( {dataset=} ds_1, {component=} comp,
    [
      {hierarchy_name=} hierarchyname |
      ( ( ( {from=} maps_from, {level=} level, {sign=} [+|-] )
        { ( {from=} maps_from, {level=} level, {sign=} [+|-] ) * )
        to {to=} maps_to } + ) as hierarchyname
    ],
  )

```

```
4625     {isFilter=} isFilter
4626     {, {aggregation=} [sum|prod] }
4627 )
```

#### 4629 *Parameters*

4630 *ds\_1* : Dataset<?,MeasureComponent<Numeric>+>

4631 *comp* : Identifier Component

4632 *hierarchyname* : string

4633 *maps\_from, maps\_to* : Constant

4634 *level* : integer

4635 *hierarchyname* : string

4636 *isFilter* : boolean

4637

4638 *ds\_1* – the Dataset to be aggregated

4639 *comp* – the *IdentifierComponent* to aggregate upon

4640 *hierarchyname* – the name of the hierarchical aggregation of the information model, which can be optionally replaced by an inline specification of the rule

4642 *maps\_from* – an input value in an inline aggregation rule

4643 *level* – the level of hierarchy of a correspondence in an aggregation rule

4644 *sign* – the sign of the contribution of a *maps\_from* value in an aggregation rule

4645 *maps\_to* – an output value in an inline aggregation rule

4646 *isFilter* – if the aggregation must be interpreted as a filter (excluding non matching records)

4647

#### 4648 *Constraints*

4649 • *hierarchyname* denotes an hierarchical aggregation either externally configured in a table or embedded in the operator with the inline notation (static).

4651 • level > 0 (static).

4652 • All the *MeasureComponents* of *ds\_1* must be Numeric (static).

4653

#### 4654 *Semantic specification*

4655 It applies a hierarchical aggregation with name *hierarchyname* on an Identifier Component *comp* in a Dataset *ds\_1*, aggregating all the Measure Components according to an aggregation function (algebraic sum or product).

4657 *hierarchyname* can be the identifier of an aggregation that is externally configured in a table, with an associated set of mappings, each with a sign and a level.

4659 Alternatively, a hierarchical aggregation can be expressed in an inline fashion, where *maps\_from* constants are mapped into *maps\_to* ones in the specific level and sign.

4661 For the given aggregation, for each level, all the mappings are considered and orderly applied with the following logic.

4663 For each value of the *IdentifierComponent* of all the records of the considered Dataset, if the value is present in *maps\_from* for any mapping, it is turned into the respective *maps\_to* value; if the value is not present in *maps\_from* for any correspondence, the entire record is discarded if *isFilter* is TRUE, the original value is preserved if *isFilter* is false.

4667 Aggregations are typically hierarchical, in the sense that they map many *maps\_from* values into fewer *maps\_to* values: often, multiple component Code Items collapse into the same compound Code Item.

4669 Therefore, at this stage, there may be multiple records having the same values for all the *IdentifierComponents*, as the differentiating ones have been aggregated into the same one.

4671 The records having the same value for all the *IdentifierComponents* are aggregated by algebraic sum (**sum**) or product (**prod**) of their *MeasureComponents*. If the aggregation function is omitted, sum is implied.

4673 Sum implies that the *MeasureComponents* have to be algebraically summed, considered with positive or negative sign depending on the Sign of the used mapping.

4675 Prod implies that the *MeasureComponents* have to be multiplied, considered with -1 exponent when the negative Sign is used in the mapping.

4677 Notice that the use of prod as aggregation function is meaningful only when the *IdentifierComponent* is a measure dimension.

#### 4679 **Hierarchies and measure dimensions**

4680 As it is well known, an *IdentifierComponent* in a Dataset can play the role of *measure dimension*, meaning that the Dataset is indeed multi-measure, but represented as mono-measure with a further, measure-qualifying dimension.

4683 *IdentifierComponents* in hierarchy can also be a measure dimension, although in this case the aggregation inherently assumes a different meaning.

4684

4685 In facts, Data Points with all coinciding *IdentifierComponents*, except for the measure dimension, are nothing but  
 4686 an expression of different measures for the same data point. An aggregation over the *measure dimension* is  
 4687 conceptually an operation involving the measures of the same Data Point (algebraic sum or multiplication).

4688 **Hierarchies as transcodings**

4689 The presented mapping method can be used intuitively to express a transcoding in a synthetic way. In this case,  
 4690 the involved Component value is mapped into another one, which is not hierarchically related with the first, but  
 4691 simply represents the same value expressed in another coding standard.

4692 **Hierarchies as filters**

4693 *isFilter* parameter in *hierarchy* allows choosing whether an input Data Point is to be kept in the output even if  
 4694 there are not mappings having the *maps\_from* value corresponding to the Identifier Component of that Data  
 4695 Point.

4696 Indeed, this mechanism lends itself to the construction of reusable filters, independent of the specific Dataset  
 4697 and Identifier Component they are applied on.

4698 Such rules comprise a set of ( $X \rightarrow X$ ,  $Y \rightarrow$ , ...,  $Z \rightarrow Z$ ) correspondences preserving the values X, Y and Z of the  
 4699 *IdentifierComponents* and filtering out the Data Points having a value for the *IdentifierComponent* different from  
 4700 X, Y or Z.

4701 *Returns*

4702 The aggregated Dataset

4703 *Examples*

4704 1) The expression:

4705 Income\_by\_country\_and\_nace\_ISO := **hierarchy**("Income\_by\_country\_and\_nace\_UN", GEO, un\_to\_ISO\_aggr, false)

4706 or its equivalent inline form:

4707 Income\_by\_country\_and\_nace\_ISO := **hierarchy**("Income\_by\_country\_and\_nace\_UN", GEO,

4708 (("ITA",1,+) to "IT", (("ALB",1,+) to AL, (("BEL",1,+) to "BE") as un\_to\_ISO\_aggr, false)

4709 Converts the values of the GEO Identifier Component from the United Nations 3-letter standard into the ISO 2-  
 4710 letter one.

4711

HIERARCHY NAME	MAPS FROM	MAPS TO	START DATE	END DATE	SIGN	LEVEL	OUTPUT
un_to_ISO_aggr	ITA	IT	....	...	+	1	TRUE
un_to_ISO_aggr	ALB	AL	...	...	+	1	TRUE
un_to_ISO_aggr	BEL	BE	...	...	+	1	TRUE

4712

4713 2) The expression:

4714 Income\_by\_continent\_and\_nace := **hierarchy**("Income\_by\_state\_and\_nace", GEO, Continent\_aggr, false)

4715 Takes as input the Dataset of the income, broken down by states and NACE and aggregates to continent level.

4716 The expression:

4717

HIERARCHY NAME	MAPS FROM	MAPS TO	START DATE	END DATE	SIGN	LEVEL	OUTPUT
Continent_aggr	Italy	Europe	....	...	+	1	TRUE
Continent_aggr	France	Europe	...	...	+	1	TRUE
Continent_aggr	Luxembourg	Europe	...	...	+	1	TRUE
Continent_aggr	...	Europe			+	1	TRUE
Continent_aggr	China	Asia			+	1	TRUE
Continent_aggr	India	Asia			+	1	TRUE
Continent_aggr	USA	America			+	1	TRUE
Continent_aggr	...	...			+	1	TRUE

4721

Income_by_state_and_nace		
GEO	NACE	VALUE
Italy	IND	10
Italy	TECH	20
France	IND	31
France	TECH	50
Spain	IND	30
Spain	TECH	15
China	IND	250
China	TECH	250
India	IND	30
India	TECH	100
Luxembourg	IND	10
Luxembourg	TECH	12

4722

4723

INCOME_BY_CONTINENT_AND_NACE		
GEO	NACE	VALUE
Europe	IND	81
Europe	TECH	97
Asia	IND	280
Asia	TECH	359

4724

4725

4726

4727

4728

4729

4730

3) The expression:

Income\_by\_world\_and\_nace := **hierarchy**("Income\_by\_state\_and\_nace", GEO, World\_aggr, false)

Takes as input the Dataset of the income, broken down by states and NACE and aggregates to the world level. World\_aggr is a 2 levels Rule. LEVEL 1 is only a temporary output and its MAPS TO Code Items do not appear in the final result.

HIERARCHY NAME	MAPS FROM	MAPS TO	START DATE	END DATE	SIGN	LEVEL	OUTPUT
World_aggr	Italy	Europe	...	...	+	1	false
World_aggr	France	Europe	...	...	+	1	false
World_aggr	Luxembourg	Europe	...	...	+	1	false
World_aggr	...	Europe			+	1	false
World_aggr	...	Europe			+	1	false
World_aggr	China	Asia			+	1	false
World_aggr	India	Asia			+	1	false
World_aggr	USA	America			+	1	false
World_aggr	Asia	World			+	2	true



<b>World_aggr</b>	Europe	World			+	2	true
<b>World_aggr</b>	America	World			+	2	true
<b>World_aggr</b>	Oceania	World			+	2	true
<b>World_aggr</b>	Africa	World			+	2	true

4731  
4732

Income_by_state_and_nace		
GEO	NACE	VALUE
Italy	IND	10
Italy	TECH	20
France	IND	31
France	TECH	50
Spain	IND	30
Spain	TECH	15
China	IND	250
China	TECH	250
India	IND	30
India	TECH	100
Luxembourg	IND	10
Luxembourg	TECH	12

4733  
4734

Income_world		
GEO	NACE	VALUE
WORLD	IND	361
WORLD	TECH	444

4735

4) The expression:

4736

Income\_world\_and\_nace\_par := **hierarchy**("Income\_by\_state\_and\_nace", World\_aggr\_par, false)

4737

Or its equivalent inline version:

4738

Income\_world\_and\_nace\_par := **hierarchy** ( Income\_by\_state\_and\_nace, GEO,  
 ((“Italy”,1,+),(“France”,1,+),(“Luxembourg”,1,+) to “Europe”,  
 ((“China”,1,+),(“India”,1,+)) to “Asia”,  
 ((“USA”,1,+) to “America”,  
 ((“Asia”,2,+),(“Europe”,2,+),(“America”,2,+),  
 (“Oceania”,2,+),(“Africa”,2,+)) to “World”)  
 as World\_aggr\_par,  
 false)

4739

4740

4741

4742

4743

4744

4745

4746

4747

4748

4749

Takes as input the Dataset of the income, broken down by states and NACE and aggregates to the World level.  
 Differently from example 2, it preserves the first level in the output.

HIERARCHY NAME	MAPS FROM	MAPS TO	START DATE	END DATE	SIGN	LEVEL	OUTPUT
<b>World_aggr_par</b>	Italy	Europe	....	...	+	1	true
<b>World_aggr_par</b>	France	Europe	...	...	+	1	true

World_aggr_par	Luxembourg	Europe	...	...	+	1	true
World_aggr_par	...	Europe			+	1	true
World_aggr_par	...	Europe			+	1	true
World_aggr_par	China	Asia			+	1	true
World_aggr_par	India	Asia			+	1	true
World_aggr_par	USA	America			+	1	true
World_aggr_par	Asia	World			+	2	true
World_aggr_par	Europe	World			+	2	true
World_aggr_par	America	World			+	2	true
World_aggr_par	Oceania	World			+	2	true
World_aggr_par	Africa	World			+	2	true

4750  
4751

Income_by_state_and_nace		
GEO	NACE	VALUE
Italy	IND	10
Italy	TECH	20
France	IND	31
France	TECH	50
Spain	IND	30
Spain	TECH	15
China	IND	250
China	TECH	250
India	IND	30
India	TECH	100
Luxembourg	IND	10
Luxembourg	TECH	12

4752  
4753

Income_world_and_nace_par		
GEO	NACE	VALUE
World	IND	361
World	TECH	444
Europe	IND	81
Europe	TECH	97
Asia	IND	280
Asia	TECH	359

4754  
4755  
4756  
4757

5) The expression:  
Italian\_income\_by\_nace:= **hierarchy**("Income\_by\_state\_and\_nace", Italy\_filter, true)

4758 Takes as input the Dataset of the income, broken down by states and NACE and filters out all the incomes that do  
 4759 not refer to Italy.

HIERARCHY NAME	MAPS FROM	MAPS TO	START DATE	END DATE	SIGN	LEVEL	OUTPUT
<b>Italy_filter</b>	Italy	Italy	....	...	+	1	TRUE

4760  
4761

Income_by_state_and_nace		
GEO	NACE	VALUE
Italy	IND	10
Italy	TECH	20
France	IND	31
France	TECH	50
Spain	IND	30
Spain	TECH	15
China	IND	250
China	TECH	250
India	IND	30
India	TECH	100
Luxembourg	IND	10
Luxembourg	TECH	12

4762  
4763

Italian_income_by_nace		
GEO	NACE	VALUE
Italy	IND	10
Italy	TECH	20

4764  
4765

6) The expression:

income\_prod\_2:= **hierarchy**("Income\_by\_state\_and\_nace\_mm", GEO, mult\_rule, false, prod) .

4766 Takes as input the Dataset of the income, broken down by states and NACE in a measure dimension form and  
 4767 aggregates by calculating INC2 / INC1 into INC.  
 4769

HIERARCHY NAME	MAPS FROM	MAPS TO	START DATE	END DATE	SIGN	LEVEL	OUTPUT
<b>Mult_measure</b>	INC1	INC	....	...	-	1	TRUE
<b>Mult_measure</b>	INC2	INC	....	...	+	1	TRUE

4770

Income_by_state_and_nace			
GEO	NACE	MEASURE	VALUE
Italy	IND	INC1	10
Italy	IND	INC2	20
Italy	TECH	INC1	20

Italy	TECH	INC2	40
France	IND	INC1	31
France	IND	INC2	61
France	TECH	INC1	50
France	TECH	INC2	100
China	IND	INC1	250
China	IND	INC2	500
China	TECH	INC1	250
China	TECH	INC1	500
India	IND	INC1	30
India	IND	INC2	60
India	TECH	INC1	100
India	TECH	INC2	200

4771

4772

Income_prod2			
GEO	NACE	MEASURE	VALUE
Italy	IND	INC	2
Italy	TECH	INC	2
France	IND	INC	2
France	TECH	INC	2
China	IND	INC	2
China	TECH	INC	2
India	IND	INC	2
India	TECH	INC2	2

4773

4774

4776 **check**

4777

4778 **check** (with datapoint rulesets)4779 *Semantics*4780 This **check** operator applies one or more datapoint Ruleset on a Dataset.

4781

4782 *Syntax*4783 **check** (4784 *ds*,4785 *dpr*+4786 { , **not valid** | **valid** | **all** }4787 { , **condition** | **measures** }

4788 )

4789 *Parameters*4790 *ds* : dataset-type

4791

- 4792 •
- ds*
- is the Dataset to check

- 4793 •
- dpr*
- is a data point Ruleset

- 4794 •
- valid**
- returns the valid data points of
- ds*
- according to
- dpr*

- 4795 •
- not valid**
- returns the not valid data points of
- ds*
- according to
- dpr*
- (default)

- 4796 •
- all**
- returns all data points of
- ds*
- , independently of whether a specific rule of a Ruleset is respected or not

- 4797 •
- condition**
- returns a Boolean Measure named
- CONDITION**
- with true values for the Data Points that satisfy
- 
- 4798 the a specific rule of a Ruleset and false otherwise

- 4799 •
- measures**
- returns the original Measures and attributes of
- ds*
- (default). The parameter mMasures cannot be
- 
- 4800 used in combination with
- all**
- .

4801

4802 *Constraints*4803 *ds* has exactly the variables that are defined in the signature of *dpr*.

4804

4805 *Returns*4806 It Returns a Dataset having the same identifiers as the input Dataset plus the RULE\_ID identifier. The values of  
4807 RULE\_ID are the concatenation of the name of the Data Point ruleset applied (which is meaningful, since the  
4808 operator can apply multiple rules to the same Dataset) and the name of the rule within the Ruleset. The  
4809 Measures returned depend on the specified option:

- 4810 • With the
- condition**
- option: returns a Dataset having all Identifier Components of
- ds*
- , the Identifier Component
- 
- 4811 RULE\_ID, the Measure
- CONDITION**
- with a Boolean value holding the outcome of the validation and the
- 
- 4812 attributes ERRORMESSAGE and ERRORLEVEL. If ERRORMESSAGE and ERRORLEVEL are not specified in the
- 
- 4813 datapoint (horizontal) rule the values of the related attributes will be NULL for all the data points .

- 4814 • With the
- measures**
- option: returns a Dataset having all Identifier Components of
- ds*
- , the Identifier RULE\_ID,
- 
- 4815 all the original Measures of
- ds*
- and the attributes ERRORMESSAGE and ERRORLEVEL. If ERRORMESSAGE and
- 
- 4816 ERRORLEVEL are not specified in the datapoint ruleset the values of the related attributes will be NULL for
- 
- 4817 all the data points.

4818 The attributes of the Dataset will be:

- 4819 • ERRORMESSAGE, containing the error message specified in the rule

- 4820 • ERRORLEVEL, containing the error level (severity) specified in the rule

4821 If **ERRORMESSAGE** and **ERRORLEVEL** are not specified in the datapoint (horizontal) rule the values of the  
 4822 related attributes will be **NULL** for all the data points .  
 4823 If **not valid** is specified then **check** returns the data points of *ds* that do not satisfy at least a rule of *dpr*. If a data  
 4824 point in *ds* does not satisfy several data rules then several data points are returned, one for each rule that is not  
 4825 satisfied, with the associated rule\_id, error message and error level.  
 4826 If **valid** is specified then **check** returns the data points of *ds* that do satisfy all rules of *dpr*.  
 4827 If **all** is specified then **check** returns all data points of *ds*. This option is normally used in combination with  
 4828 **condition**.  
 4829 See also the examples under **define datapoint ruleset**.

4830 *Examples*

4831

ds_labour			
TIME	PERSON_ID	AGE	EDU
2011	1	15	5
2011	2	20	3
2011	3	17	6
2011	4	19	4
2011	5	32	6
2011	6	17	14
2011	7	25	14
2011	8	18	10
2011	9	15	3
2011	10	40	5

4832 Where the variable edu is coded using the following classification:  
 4833

edu	descr
1	No title
2	Elementary License / Certificate of final evaluation
3	Middle School (or professional training)/ Diploma in Education secondary level
4	Professional qualifications Diploma of secondary school 2-3 years which does not allow enrollment at the University
5	High School Diploma / Secondary Education degree higher than 4-5 years which allows enrollment at the University
6	Academy Diploma (Fine Arts, Dramatic Arts National, National Dance), Higher Institute of Artistic Industries, State Conservatory of Music
7	University degree of two / three years, direct school for special purposes, school equivalent education
8	bachelor's degree (three years)
9	Specialist / Master's degree (two years)
10	4-6 years Degree: Bachelor's degree from the old system or graduate specialization / teaching single-cycle
13	VET Certificate of professional qualification (operator) / Professional diploma IFP technical

	(Three-year pathways / four-year education and training)
14	higher technical specialization certificate (HTE)
15	Higher Technical Diploma (ITS)

```

4834
4835 define datapoint ruleset dpr_labour ( AGE as a ) (
4836     rule_1 when a between 14 and 17 then edu <> 5 ;
4837     rule_2 when a between 16 and 19 then edu <> 6 ;
4838     rule_3 when a between 17 and 20 then edu <> 7 and edu <> 8 ;
4839     rule_4 when a between 18 and 21 then edu <> 10 ;
4840     rule_5 when a between 14 and 16 then edu <> 13 ;
4841     rule_6 when a between 16 and 18 then edu <> 14 ;
4842     rule_7 when a between 17 and 20 then edu <> 15 ;
4843 );
4844
4845 ds_validation_report := check ( ds_labour, dpr_labour, not valid, condition )
4846

```

ds_validation_report							
TIME	PERSON_ID	RULE_ID	AGE	EDU	CONDITION	ERRORMESSAGE	ERRORLEVEL
2011	1	hr_labour_rule_1	15	5	FALSE		
2011	3	hr_labour_rule_2	17	6	FALSE		
2011	6	hr_labour_rule_6	17	14	FALSE		
2011	8	hr_labour_rule_4	18	10	FALSE		

```

4847
4848
4849 check \(with hierarchical rulesets\)
4850

```

#### Semantics

This **check** operator applies one or more hierarchical (vertical) ruleset on a Dataset.

#### Syntax

```

4855 check (
4856     ds,
4857     hr+
4858     { , threshold ( threshold ) }
4859     { , not valid | valid | all }
4860     { , measures | condition }
4861 )

```

#### Parameters

*ds* : dataset-type

*threshold* : numeric-constant

- 4867 • *ds* is the Dataset to check
- 4868 • *hr* is a code item compatibility ruleset
- 4869 • *threshold* is the threshold (tolerance value) to be applied as the upper limit of the difference between the left and right side of the rules. In the simplest case *threshold* is a numeric constant. A more sophisticated form exists where *threshold* is an expression involving the following predefined values:
  - 4872 ○ *left\_side*    the value of the left-hand side of the rule

4873           ○ `right_side`   the value of the right-hand side of the rule (the value computed by VTL)

4874           ○ `nr_items`     the number of items in the right-hand side of the rule

4875           Examples of possible threshold expressions:

4876           `threshold ( 3 )`

4877           `threshold ( abs ( left_side - right_side ) > 3 )`       equivalent to the above \*/

4878           `threshold ( abs ( left_side / right_side ) > abs ( 50 * left_side ) )`    can differ up to 50 % of left side\*/

4879           `threshold ( abs ( left_side - right_side ) > 0.5 * nr_items )`   can differ up to 0.5 for each item % \*/

4880           • **valid** returns the valid data points of `ds` according to `vr`

4881           • **not valid** return the non valid data points of `ds` according to `hr` (default)

4882           • **all** returns all data points of `ds`, independently of whether a specific rule of a Ruleset is respected or not

4883           • **condition** returns a Boolean Measure Boolean attribute "condition" named **CONDITION** with true values for

4884           the Data Points that satisfy a specific rule of a ruleset and false otherwise

4885           • **measures** returns the original Measures and attributes of `ds` (default). The parameter **measures** cannot be

4886           used in combination with **all**.

4887           *Constraints*

4888           • `ds` has the variables specified in the code item compatibility ruleset `hr`.

4889           • The ruleset must be explicitly defined for validation purposes.

4890           *Returns*

4891           It returns a Dataset having all Identifier Components of `ds` plus the Identifier `RULE_ID`, which allows to

4892           distinguish between validation rules in the various rulesets when ambiguities arise. The values of `RULE_ID` are

4893           built as the concatenation of the ruleset identifier and the rule identifier.

4894           The Measures of the Dataset returned depend on the specified option:

4895           • With the **condition** option: it returns a Dataset having the Measure **CONDITION** (true/false)

4896           • With the **measures** option: it returns a Dataset having all the Measures of `ds`

4897           Additional Measures are:

4898           • `value_CONDITION`: containing the value computed by executing the rule

4899           • `rule`: containing the returning text related to the rule

4900           The attributes of the Dataset will be:

4901           • `errorcode`, containing the error message specified in the rule

4902           • `errorlevel`, containing the error level (severity) specified in the rule

4903           If `errormessage` and `errorlevel` are not specified in the ruleset, then the values of the related attributes will be

4904           NULL for all the data points .

4905

4906           *Examples*

4907           See to the examples for hierarchical (vertical) rulesets.

4908

4909           `check ( single rule)`

4910

4911           *Semantics*

4912           The **check** operator takes as input a Boolean VTL expression and uses it as the indication of a validation.

4913           *Syntax*

4914           **check** (

4915                `ds`

4916                { , **threshold** (`threshold`) }

4917                { , **not valid** | **valid** | **all** }



```

4918     { , measures | condition }
4919     { , imbalance ( imbalance ) }
4920     { , errorcode ( errorcode ) }
4921     { , errorlevel ( errorlevel ) }
4922 )

```

#### 4923 *Parameters*

```

4925 ds : dataset {identifier <IDENT> as scalar-type; }+ {measure <IDENT> as numeric; }+
4926       {measure <IDENT> as boolean-type; }* {attribute <IDENT> as scalar-type; }*
4927 threshold : scalar-constant
4928 imbalance : numeric-constant
4929 errormessage : string-constant
4930 errorlevel : integer-constant
4931

```

- 4932 • *ds* is the Boolean VTL expressions, hence yielding a Boolean Dataset, that represents the validation.
- 4933 • *threshold* is a tolerance number. It requires the presence of an imbalance. If this latter value is below the
- 4934 *threshold*, then the data point is considered valid, thus the Boolean Measure **CONDITION** is true; false in all
- 4935 the other cases.
- 4936 • **valid** returns only the valid data points
- 4937 • **not valid** returns only the non valid data points
- 4938 • **all** returns all data points, independently of their validity
- 4939 • **condition** returns a Boolean Measure "**CONDITION**" with values true for the data points that satisfy the
- 4940 ruleset and false otherwise
- 4941 • **measures** returns a Boolean Measure "**CONDITION**" with values true for the data points that satisfy the
- 4942 ruleset and false otherwise (default)
- 4943 • *imbalance* is the imbalance to be computed. Imbalance has a number datatype. If not specified in the check, it
- 4944 will be not in the output.
- 4945 • *errorcode* is the error code to be produced when the validation fails.
- 4946 • *errorlevel* is the error level (severity) of the validation rule. Errorlevel has a string datatype. If not specified
- 4947 in the check, it will be not in the output.

#### 4948 *Constraints*

- 4949 • The input Dataset must have all Boolean Measure Components.
- 4950 • When a threshold is specified, the imbalance must be specified as well.

#### 4951 *Returns*

4952 It returns a Dataset with all the Identifier Components of *ds* .

4953 The measures of the Dataset returned, depend on the option specified:

- 4954 • With the **condition** option: returns the Measure **CONDITION** (true/false) .
- 4955 • With the **measures** option: returns all the Measures of *ds* .

4956 Additional measures are:

- 4957 • **CONDITION**, when the the Boolean value computed by executing the rule (true/false) (depending on the
- 4958 optional parameter (condition or Measures):
- 4959 • *imbalance*, imbalance to be computed

4960 The attributes of the Dataset will be:

- 4961 • *errorcode*, containing the error code specified in the rule
- 4962 • *errorlevel* , containing the error level (severity) specified in the rule

4963 *Semantic specification*

4964 It takes as input a Boolean VTL expression and uses it as the indication of a validation. It returns an output  
4965 Dataset that specifies the outcome of the validation. It can report in the output Dataset what are the violations,  
4966 what are the original data, what is the imbalance between the expected values and the actual ones also applying  
4967 thresholds. It can also link the failed validations to specific error codes and error levels for further processing.  
4968

4969 *Examples*

ds_bop				
TIME	REF_AREA	PARTNER	FLOW	OBS_VALUE
2010	IT	US	NET	10
2011	IT	US	NET	20
2012	IT	US	NET	50
2013	IT	US	NET	40
2014	IT	US	NET	50
2015	IT	US	NET	60
2010	DE	US	NET	25
2011	DE	US	NET	35
2012	DE	US	NET	45
2013	DE	US	NET	55
2014	DE	US	NET	65
2015	DE	US	NET	75

4970  
4971 Check that the difference between each value and the average of that value, its preceding value and following  
4972 value is lower than 10 (in absolute value).

```

4973
4974 ds_moving_average := avg ( ds_bop ) over (
4975     partition by ref_area, partner
4976     order by time
4977     rows between 1 preceding and 1 following ) ;
4978 ds_outliers := check ( abs ( ds_bop.obs_value - ds_moving_average.obs_value ) <= 10 ) ;
4979

```

ds_outliers							
TIME	REF_AREA	PARTNER	FLOW	OBS_VALUE	IMBALANCE	ERRORMESSAGE	ERRORLEVEL
2012	IT	US	NET	50			

4980  
  
4981 **check value domain subset**

4982 *Semantics*

4983 The **check\_value\_domain\_subset** operator checks if the values of the specified Components owned by *the*  
4984 Dataset are part of the restricted domain of the ValueDomain.

4985 *Syntax*

```

4986 check_value_domain_subset ( ds, [ components | { compList ( {compIdent}+ ), valueDomain }, vds );
4987

```

4988 *Parameters*

```

4989 ds : dataset {identifier <IDENT> as scalar-type}+
4990     {measure <IDENT> as scalar-type}* {attribute <IDENT> as scalar-type}*
4991

```

4992 *vds* : valueDomainSubset-ref  
 4993 *components* : list <component-ref>  
 4994 *compList* : list<list<component-ref>>  
 4995 *compIndentm*: list <component-ref>  
 4996 *valueDomainm*: list <dimension-ref>

- 4997
- 4998 • *ds* – is the starting Dataset.
  - 4999 • *components* - is the List containing the Components owned by *ds* to be validated.
  - 5000 • *compList* – is the list containing the Components (divided in lists *compIndent*) that must be checked according to the restrictions defined in the ValueDomainSubset *vds* passed as input.
  - 5001
  - 5002 • *valueDomain* – is the list containing the dimension of the ValueDomainSubset that are used to validate the Components referred in *compList* and *compIndent*
  - 5003
  - 5004 • *vds* – is the ValueDomainSubset containing the restrictions to be verified in *ds*.

5005

5006 **Constraints**

- 5007 • if only components are defined then *vds* must be mono-dimensional.
- 5008 • if both *compList* and *compIndent* are defined they must be of the same dimension.

5009

5010 **Returns**

5011 A Dataset with all the Identifier, Measure and Attribute Components of the input one enriched by a Boolean Measure Component for each Component specified in *components* or *compList*, that contains the result of the check, against the ValueDomainSubset restrictions, for the respective values.

5012

5013

5014

5015 **Semantic specification**

5016 The operator checks if the values of the specified Components owned by *ds* are part of the restricted domain of the ValueDomain in *vds*, returning a Dataset with the same structure of the input one and a Boolean Measure Component for each Component specified in the signature (the name of the new Component is “COMPONENT\_NAME” + “\_CONDITION”). For each Datapoint the new Boolean Measure Component assumes the value TRUE if the value of the respective Component is part of the restricted domain of the respective ValueDomain in *vds*, FALSE otherwise.

5021 The operator can work in two mode: mono-dimensional and multi-dimensional mode.

5022 In the mono-dimensional version (only *components* defined) it takes as input a Dataset, a List of Components and a mono-dimensional ValueDomainSubset. It evaluates if all the values inside the specified Components of *ds* are part of the restricted domain defined by the mono-dimensional ValueDomainSubset.

5023 In the multi-dimensional version (both *compList* and *valueDomains* defined) it takes as input a Dataset, two Lists and a multi-dimensional ValueDomainSubset. The first list is a List of Lists containing names of components owned by *ds*, the second List contains reference to the ValueDomain owned by *vds*. The Components specified in the first element of *compList* will be checked against the ValueDomain specified in the first element of *valueDomains*, and so on (it follows that the two Lists must have the same size and order of the elements matters).

5032

5033 **Examples**

5034 1)

5035

ds_1				
TIME	REF_AREA	PARTNER	OBS_VALUE	OBS_STATUS
2010	EU25	CA	20	D
2010	BG	CA	1	P
2010	RO	CA	1	P
2010	EU27	CA	23	P

5036

5037 *l\_1* = list<components-ref> (REF\_AREA)

5038

5039 *ds\_1* := **check\_value\_domain\_subset** (*ds\_1*, *l\_1*, *vds\_1*)

5040

5041 vds\_1 is a mono-dimensional enumerated ValueDomainSubset, the CodeList referenced by its ValueDomain  
 5042 contains the values: ["EU25","EU27","EU28"]  
 5043

ds_2					
TIME	REF_AREA	REF_AREA_CONDITION	PARTNER	OBS_VALUE	OBS_STATUS
2010	EU25	TRUE	CA	20	D
2010	BG	FALSE	CA	1	P
2010	RO	FALSE	CA	1	P
2010	EU27	TRUE	CA	23	P

5044  
 5045 2)  
 5046 compList := list<list<component-ref>>(list<component-ref>(REF\_AREA), list<component-ref>(OBS\_VALUE))  
 5047 ds\_2 := **check\_dataset\_values** (ds\_1, compList, list<valueDomain-ref>(D1, D2), vds\_1)  
 5048

5049 vds\_1 is a multi-dimensional ValueDomainSubset with two dimensions D1 and D2. D1 take its domain from the  
 5050 values of a CodeList defined as ["EU25","EU27","EU28"], D2 is a numeric restricted domain that allows only  
 5051 positive integers.

5052  
 5053 Returned Dataset:

ds_2						
TIME	REF_AREA	REF_AREA_CONDITION	PARTNER	OBS_VALUE	OBS_VALUE_CONDITION	OBS_STATUS
2010	EU25	TRUE	CA	20	TRUE	D
2010	BG	FALSE	CA	1	TRUE	P
2010	RO	TRUE	CA	1	TRUE	P
2010	EU27	FALSE	CA	23	TRUE	P

5054  
 5055

5056

## VTL-ML - Time series functions

5057

### fill\_time\_series

5058

#### Semantics

5059

The operator **fill\_time\_series** replaces each missing data point in the input Dataset (from the lowest to the highest time period found in the Dataset) with a data point having the values of Measures and attributes set to null.

5061

5062

5063

#### Syntax

5064

**fill\_time\_series** ( *ds*, *freq*, { , *timePeriodName* { , *timeFormat* } )

5065

5066

#### Parameters

5067

*ds* : dataset {identifier <IDENT> as scalar-type}+ {identifier <IDENT> as date}

5068

{measure <IDENT> as numeric}+

5069

{measure <IDENT> as scalar-type}\* {attribute <IDENT> as scalar-type}\*

5070

5071

*ds* – is the input Dataset whose missing data points in the series will be filled in.

5072

5073

#### Constraints

5074

The Dataset *ds* must have the specified the Identifier Component *timePeriodName* or the default "time".

5075

5076

#### Returns

5077

A Dataset having the same Identifier, Measure and Attribute Components as the input one. The missing data points in each series will be filled in.

5079

5080

#### Semantic specification

5081

fill\_time\_series allows to fill in all series of *ds* (no need to process the series one by one).

5082

The time format can be specified as described in the table under "Time aggregate functions".

5083

5084

#### Examples

ds_bop				
TIME	REF_AREA	PARTNER	FLOW	OBS_VALUE
2010	IT	US	NET	10
2012	IT	US	NET	50
2010	DE	US	NET	25

5085

5086

ds\_fill\_ts := fill\_time\_series ( ds\_bop, "A", time )

ds_fill_ts				
TIME	REF_AREA	PARTNER	FLOW	OBS_VALUE
2010	IT	US	NET	10
2011	IT	US	NET	null
2012	IT	US	NET	50
2010	DE	US	NET	25
2011	DE	US	NET	null
2012	DE	US	NET	null

5087

5088

5089

5090

Note: the Dataset contains data from 2010 to 2012 therefore 1 data point is inserted for the series (IT, US, NET) and 2 data points are inserted for the series (DE, US, NET).

## 5091 flow\_to\_stock

### 5092 *Semantics*

5093 The operator **flow\_to\_stock** consists in the transformation from a flow interpretation of a Dataset (with one  
5094 single date Identifier Component), where the numeric Measures represent relative modifications of the stock  
5095 level (flow), to the corresponding stock interpretation of it, where the numeric Measures represent stock levels  
5096 Measured at a specific time (stock).

5097

### 5098 *Syntax*

5099 **flow\_to\_stock** ( *ds* )

5100

### 5101 *Parameters*

5102 *ds* : dataset {identifier <IDENT> as scalar-type}+ {identifier <IDENT> as date}

5103 {measure <IDENT> as numeric}+

5104 {measure <IDENT> as scalar-type}\* {attribute <IDENT> as scalar-type}\*  
5105

5106 *ds* – the input Dataset containing time series.

5107

### 5108 *Constraints*

- 5109 • *ds* must be a Dataset that contains only one date Identifier Component (as in the syntax).
- 5110 • *ds* must be regular, that is, once the Data Points have been ordered by the only date Identifier, for each pair  
5111 of consecutive Data Points, the distance in time between the respective date values must be constant (and  
5112 typically one year, one quarter, one day and so on).

5113

### 5114 *Returns*

5115 A Dataset having the same Identifier, Measure and Attribute Components as the input one. The values of the  
5116 numeric Measure Components are computed as the stock interpretation of the respective input values, which are  
5117 considered according to a stock interpretation.

5118

### 5119 *Semantic specification*

5120 The operator takes as input a Dataset *ds* and returns another one with the same Identifier, Measure and  
5121 Attribute Components as the input one. We say that two Data Points *dp1* and *dp2* of *ds* are consecutive if they  
5122 have the same values for all the Identifier Components but the one with date data type and, once all the Data  
5123 Points with the same values for all the Identifier Components have been ordered by date Component, they are  
5124 adjacent.

5125 The Data Points of the output are calculated as follows. Data Points in *ds* are partitioned in blocks having the  
5126 same values for all the Identifier Components but the date one. For each block, the first Data Point is copied into  
5127 the output. Then, for each pair of consecutive Data Points *dp1* and *dp2* (that is, *dp2* follows *dp1*) of *ds*, a new data  
5128 Point appears in the output. The value of all the Identifier Components, non-numeric Measure Components and  
5129 Attribute Components of the output Dataset are copied from *dp2*. The value of each numeric Measure  
5130 Component is calculated as the sum of the value in the output of the previously copied Data Point and the value  
5131 of the Measure Component of *dp2*. Note that the operator actually performs the cumulative sum and no Data  
5132 Points are neglected.

5133

### 5134 *Examples*

ts_1	
DATE	VALUE
1939-01-01	4400.0
1939-02-01	0.0
1939-03-01	6200.0
1939-04-01	-38000.0

5135

5136 **ts\_2 := flow\_to\_stock ( ts\_1 )**

5137

ts_2	
DATE	VALUE
1939-01-01	4400.0
1939-02-01	4400.0
1939-03-01	10600.0
1939-04-01	6800.0

5138

## 5139 stock\_to\_flow

### 5140 *Semantics*

5141 The operator **stock\_to\_flow** consists in the transformation from a stock interpretation of a Dataset (with one  
5142 single date Identifier Component), where the numeric Measures represent stock levels measured at a specific  
5143 time (stock), to the corresponding flow interpretation of it, where the numeric Measures represent relative  
5144 modifications of the stock level (flow).

5145

### 5146 *Syntax*

5147 **stock\_to\_flow** ( *ds* )

5148

### 5149 *Parameters*

5150 *ds* : dataset {identifier <IDENT> as scalar-type}+ {identifier <IDENT> as date}

5151           {measure <IDENT> as numeric}+

5152           {measure <IDENT> as scalar-type}\* {attribute <IDENT> as scalar-type}\*  
5153

5154 *ds* – the input Dataset.  
5155

### 5156 *Constraints*

- 5157 • *ds* must be a Dataset that contains only one date Identifier Component (as in the syntax).
- 5158 • *ds* must be regular, that is, once the Data Points have been ordered by the only date Identifier, for each pair  
5159 of consecutive Data Points, the distance in time between the respective date values must be constant (and  
5160 typically one year, one quarter, one day and so on).  
5161

### 5162 *Returns*

5163 A Dataset having the same Identifier, Measure and Attribute Components as the input one. The values of the  
5164 numeric Measure Components are computed as the flow interpretation of the respective input values, which are  
5165 considered according to a stock interpretation.  
5166

### 5167 *Semantic specification*

5168 The operator takes as input a Dataset *ds* and returns another one with the same Identifier, Measure and  
5169 Attribute Components as the input one. We say that two Data Points *dp1* and *dp2* of *ds* are consecutive if they  
5170 have the same values for all the Identifier Components but the one with date data type and, once all the Data  
5171 Points with the same values for all the Identifier Components have been ordered by date Component, they are  
5172 adjacent.

5173 The Data Points of the output are calculated as follows. Data Points in *ds* are partitioned in blocks having the  
5174 same values for all the Identifier Components but the date one. For each block, the first Data Point is copied into  
5175 the output. Then, for each pair of consecutive Data Points *dp1* and *dp2* (that is, *dp2* follows *dp1*) of *ds*, a new data  
5176 Point appears in the output. The value of all the Identifier Components, non-numeric Measure Components and  
5177 Attribute Components of the output Dataset are copied from *dp2*. The value of each numeric Measure  
5178 Component is calculated as the difference of the respective numeric Measure.  
5179

5179

5180

5181

5182

5183 *Examples*

ts_1	
DATE	VALUE
1939-01-01	4400.0
1939-02-01	4400.0
1939-03-01	10600.0
1939-04-01	6800.0

5184

5185 `ts_2 := stock_to_flow( ts_1 )`

5186

ts_2	
DATE	VALUE
1939-01-01	4400.0
1939-02-01	0
1939-03-01	6200.0
1939-04-01	-3800.0

5187

## 5188 timeshift

5189 *Semantics*

5190 The operator **timeshift** returns the input Dataset with its time component shifted by the amount of time  
5191 specified as parameter.

5192

5193 *Syntax*

5194 **timeshift** ( *ds*, *timeId*, **unit** = [A|M|Q|D], *lag* )

5195

5196 *Parameters*

5197 *ds* : dataset {identifier <IDENT> as scalar-type}+

5198        {[identifier|measure] <IDENT> as date}

5199        {measure <IDENT> as scalar-type}\* {attribute <IDENT> as scalar-type}\*

5200 *timeId* : component-ref

5201 *lag* : integer

5202

5203        • *ds* – the input Dataset containing time series.

5204        • *timeId* – is the reference to a valid Component representing the time of the time series of *ds*, that is the  
5205        Component on which the shift operation must be performed.

5206        • *unit* – represents the unit of time to be shifted. The possibilities are: Y=year, M=Month, Q=Quarter,  
5207        D=Day

5208

5209 *Constraints*

5210 *timeId* must refer to a Component of *ds*, whose type is date (as in the syntax).

5211

5212 *Returns*

5213 A Dataset having the same Identifier, Measure and Attribute Components as the input one, with each value of the  
5214 *timeId* Component modified of *lag* units.

5215

5216 *Semantic specification*



5217 The operator takes as input a Dataset, a valid date Component of *ds (timeId)* and the specification of the **unit** and  
 5218 amount of time (*lag*) to shift. It returns a Dataset with the same structure as the input one. For each Data Point in  
 5219 *ds*, the result contains the same Data Points (so with the same values for all the Identifier, Measure and Attribute  
 5220 Components), except for the values of the Identifier Component *timeId*, which are modified by summing the  
 5221 relative amount **unit** x *lag* (note that *lag* may also be negative).  
 5222  
 5223

*Examples*

ts_1	
DATE	VALUE
1939-01-01	4400.0
1939-02-01	4400.0
1939-03-01	10600.0
1939-04-01	6800.0

5224  
 5225 `ts_2 := timeshift( ts_1, A, 1 )`  
 5226 equivalent forms:  
 5227 `ts_2 := timeshift( ts_1, M,12 )`  
 5228 `ts_2 := timeshift( ts_1, Q, 3 )`

ts_2	
DATE	VALUE
1940-01-01	4400.0
1940-02-01	4400.0
1940-03-01	10600.0
1940-04-01	6800.0

5229

5231 **if-then-else**5232 *Semantics*

5233 The operator **if-then-else** returns one of two possible dataset or values depending on the input conditions.

5234

5235 *Syntax*

5236 **if** *ds\_cond\_1* **then** *ds\_1* { **elseif** *ds\_cond\_2* **then** *ds\_2* }\* **else** *ds\_3*

5237

5238 *Parameters*

5239 *ds\_cond\_1*, *ds\_cond\_2*

5240 : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as boolean}]boolean]

5241 *ds\_1*, *ds\_2*, *ds\_3*

5242 : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as scalar-type}+|constant]

5243

5244 • *ds\_cond\_1* – is the first Boolean condition

5245 • *ds\_1* – can be:

5246 ○ a Dataset from which the Data Points are retrieved anytime that *ds\_cond\_1* evaluate true.

5247 ○ a constant constant returned if *ds\_cond\_1* evaluated true.

5248 • *ds\_cond\_2* – is an optional Boolean condition

5249 • *ds\_2* – can be:

5250 ○ a Dataset from which the Data Points are retrieved anytime that *ds\_cond\_2* evaluate true

5251 ○ a constant constant returned if *ds\_cond\_2* evaluated true.

5252 • *ds\_3* – can be:

5253 ○ a Dataset from which the Data Points are retrieved anytime that *ds\_cond\_2* not evaluate true.

5254 ○ a constant constant returned if *ds\_cond\_2* not evaluated true

5255

5256 *Constraints*

5257 • If *ds\_1*, *ds\_2* and *ds\_3* are constant values, they must have the same type.

5258 • If *ds\_cond\_1*, *ds\_cond\_2*, *ds\_1*, *ds\_2* and *ds\_3* are Datasets, they must have the same Identifier Components, in name and type.

5259 • If *ds\_cond\_1* and *ds\_cond\_2* are Datasets, they must have only one boolean Measure Component (as expressed in the syntax).

5260 • If *ds\_1*, *ds\_2* and *ds\_3* are Datasets, they must have the same Measure Components.

5263

5264 *Returns*

5265 If the input parameters are Boolean scalars then the operator returns the constant of the first evaluated true condition. If no condition evaluates true then *c\_3* is returned.

5266 If the input parameters are Datasets then the operator returns a Dataset having all the Identifier and Measure Components of the input ones, composed by Data Points retrieved in the input Datasets when the relative condition on the relative boolean Measure Component evaluate true.

5270

5271 *Semantic specification*

5272 If the input parameters are Boolean scalars then the operator takes as input a series of Boolean condition with the respective values to return in case of positive validation, it returns the constant of the first evaluated true condition or *c\_3* if not a condition evaluate true.

5275 If the input parameters are Datasets then the operator takes as input a number of condition Datasets *ds\_cond\_1*, having exactly one boolean Measure Component and, for each of them a Dataset *ds* to return in case of positive evaluation (**then**) of the condition. Besides it takes in input a default case (**else**) Dataset to be returned if all the previous conditions evaluate to False. Starting from *ds\_cond\_1*, for each Data Point, if the Measure Component is True, it looks up in the corresponding Datasets (*ds\_1*) all the Data Point for the corresponding values of the Identifier Components and returns them in the output Dataset. If the Measure Component is False, it looks up in the following **elseif** Dataset (*ds\_cond\_2*) for the corresponding values of the Identifier Components. If no Data Point is found, the elaboration skips to the next Data Point of *ds\_cond\_1*. If any Data Points are found and *ds\_cond\_2* is True for them, the corresponding **then** Dataset (*ds\_2*) is returned; otherwise, the evaluation continues likewise, until the **else** part is reached (in case every previous conditional Datasets evaluate to False)

5284

5285 and, if any matching Data Points are found in *ds\_3* they are returned. Then the elaboration is repeated for all the  
 5286 Data Points in *ds\_cond\_1*.

5287  
 5288 *Examples*

5289 On scalar  
 5290 1) Expressions evaluating to Component types are typically used to calculate Measure Components or evaluate  
 5291 filters.

5292 Some examples follow:  
 5293  $K1 + K2 < K3$   
 5294  $K1 - K2 > 5.5$   
 5295  $K2 + \text{round}(K2, 3)$   
 5296  $K1 > 3$  and  $k1 < 5$   
 5297 **if**  $k1 > 4$  **then**  $K2$  **else**  $K3 + 3$   
 5298  $K1$  in  $(1,2,3,4)$  and  $K3$  not in  $(\text{'a','b','c'})$

5300 On Datasets  
 5301 2)  $ds\_1 := \text{if}(\text{population.SEX} = \text{"F"}).CONDITION$   
 5302 **then**  $unemp\_rates\_1$   
 5303 **else**  $unemp\_rates\_2$   
 5304

population				
TIME	GEO	AGE	SEX	POPULATION
2012	Belgium	Total	M	5451780
2012	Belgium	Total	F	5643070
2012	Greece	Total	M	5449803
2012	Greece	Total	F	5673231
2012	Spain	Total	M	23099012
2012	Spain	Total	F	23719207
2012	France	Total	M	31616281
2012	France	Total	F	33671580
2012	Italy	Total	M	28726599
2012	Italy	Total	F	30667608
2012	Austria	Total	M	NULL
2012	Austria	Total	F	NULL

5305

unemp_rates_1				
TIME	GEO	AGE	SEX	RATE
2012	Spain	Total	F	25.8
2012	France	Total	F	NULL
2012	Italy	Total	F	20.9
2012	Austria	Total	M	6.3

5306

unemp_rates_2				
TIME	GEO	AGE	SEX	RATE
2012	Belgium	Total	M	0.12
2012	Greece	Total	M	22.5
2012	Spain	Total	M	23.7
2012	Austria	Total	F	NULL

5307  
5308

ds_1				
TIME	GEO	AGE	SEX	RATE
2012	Belgium	Total	M	0.12
2012	Greece	Total	M	22.5
2012	Spain	Total	M	23.7
2012	Spain	Total	F	25.8
2012	France	Total	F	NULL
2012	Italy	Total	F	20.9

5309  
5310  
5311  
5312  
5313  
5314

population.SEX allows to consider SEX into the only Measure Component, which is compared with "F" by the operator "=". Correctly, it acts on the only Measure Component as POPULATION is temporarily not considered. The comparison returns CONDITION as the only boolean Measure Component  
Thus, as the if operators requires a Dataset with a single boolean Measure Component, the membership operator "." is applied again in order to isolate SEX\_ CONDITION as the only boolean Measure Component

5315

## nvl

5316

### Semantics

The operator **nvl** replaces null values with a value given as a parameter.

5318

5319

### Syntax

5320

**nvl** (*ds*, *rep\_value* )

5321

5322

### Parameters

5323

*ds* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as scalar-type}+  
{attribute <IDENT> as scalar-type}\*|scalar-type]

5324

5325

*rep\_value* : scalar-type

5326

5327

- *ds* can be an scalar-type value or an input Dataset
- *rep\_value* is the value that replace the values in *ds* when them are NULL.

5328

5329

5330

### Constraints

5331

- If *ds* is a scalar value, *ds* and *rep\_value* must be equal in type between themselves.
- If *ds* is a Dataset, all its Measure Components must be of the same type and *rep\_value* must be of the same type of the *ds* Measure Components.

5332

5333

5334

5335

### Returns

5336

If *ds* is a scalar value, it returns *ds* if it is not NULL, *rep\_value* otherwise.

5337

If *ds* is a Dataset, it returns a new Dataset having all the Identifier, Measure and Attribute Components of the input one, where the NULL values of the input Dataset Measure Components are replaced with the specified *rep\_value*.

5339

5340

5341

### Examples

5342

On scalar

5343

1) If C is NULL:

5344

A := nvl(C, 5)                      A = 5

5345

2) If COMPX is not NULL and equal to 10:

5346

A := nvl(C, 5)                      A = 10

5347

5348

On Dataset

5349

3) ds\_1 := **nvl**(population,0)

5350

population				
TIME	GEO	AGE	SEX	POPULATION
2012	Belgium	Total	Total	11094850
2012	Greece	Total	Total	11123034
2012	Spain	Total	Total	NULL
2012	Malta	Total	Total	417546
2012	Finland	Total	Total	5401267
2012	NULL	Total	Total	NULL

5351

ds_1				
TIME	GEO	AGE	SEX	POPULATION
2012	Belgium	Total	Total	11094850
2012	Greece	Total	Total	11123034
2012	Spain	Total	Total	0
2012	Malta	Total	Total	417546
2012	Finland	Total	Total	5401267
2012	NULL	Total	Total	0

5352

5354 **rename**5355 *Semantics*

5356 The **rename** operator, allows to change the name and the role of Measures or Attributes component of a dataset

5357

5358 *Syntax*

5359 **ds\_1 [rename k as compName**  
 5360 **{role=[MEASURE|IDENTIFIER|ATTRIBUTE]}**  
 5361 **{ k as compName**  
 5362 **{role=[MEASURE|IDENTIFIER|ATTRIBUTE]}**  
 5363 **}\*]**

5364 *Parameters*

5365 *ds\_1* : Dataset<?>  
 5366 *k* : Measure or Attribute Component  
 5367 *compName* : string

5369

5370 *ds\_1* – the input Dataset  
 5371 *k* – each Component to rename  
 5372 *compName* – the new name for each Component  
 5373 *role* – the new role for each Component

5374

5375 *Returns*

5376 The Dataset with renamed Components and changed roles.

5377

5378 *Constraints*

- 5379 • *k* is a Component expression that can have only Component literals of *ds\_1* (static).
- 5380 • *role* can be one of: "MEASURE", "IDENTIFIER", "ATTRIBUTE" (static).

5381

5382 *Semantics*

5383 It renames each Measure or Attribute in *ds\_1* that is mentioned in the operator with the new name given in *compName*  
 5384 and the role given in *role* variable. If *role* variable is not specified, the role is left unmodified. All the data points in *ds\_1*  
 5385 are copied into *ds\_2*.

5386 Returns a new Dataset *ds\_2* with the same Identifier of *ds\_1*.

5387 The Dataset *ds\_2* will have the same Measure and Attributes Components of *ds\_1* except for those components changed  
 5388 in the role by the rename operator.

5389

5390 *Examples*

5391 *ds\_2* := *ds\_1*[rename M1 as "I1" role IDENTIFIER] 3154

5392 The expression above renames MeasureComponent M1 into I1 and alters its role to IdentifierComponent.

5393 **filter**5394 *Semantics*

5395 The operator **filter** returns the input Dataset filtered by evaluating the boolean component expression specified  
 5396 as a parameter.

5397

5398 *Syntax*

5399 **ds [ filter f / dpr]**

5400

5401 *Parameters*

5402 *ds* : dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as scalar-type}\*  
 5403 {attribute <IDENT> as scalar-type}\*  
 5404 *f* : {role <IDENT> as boolean}

5405 *dpr* : a previously defined datapoint ruleset

5406

- 5407 • *ds* – is the input Dataset.
- 5408 • *f*– is a Boolean expression involving Components of *ds*.

5409

#### 5410 *Constraints*

5411 *f* is a Component expression over the Components of *ds* (static).

5412

#### 5413 *Returns*

5414 A Dataset with the same Identifier, Measure and Attribute Components of the input one, containing only the Data Points of *ds* that satisfy the Boolean expression *f* for the datapoint (horizontal) ruleset

5416

#### 5417 *Semantic specification*

5418 The operator takes as input a Dataset and a Boolean expression involving the Components owned by *ds* and returns another Dataset with the same structure of the input one. For each Data Point the expression *f* is applied; only the Data Points for which the expression is evaluated true will be part of the output Dataset.

5421

5422

#### 5423 *Examples*

5424 Dr:=population1 [**filter** SEX = "F"]

5425

5426

population1				
SEX	AGE	GEO	TIME	POPULATION
M	Y_LT15	BE	2013	970428
M	Y15-64	BE	2013	3678355
M	Y_GE65	BE	2013	838653
F	Y_LT15	BE	2013	927644
F	Y15-64	BE	2013	3625561
F	Y_GE65	BE	2013	1121001
M	Y_LT15	UK	2013	5757444
M	Y15-64	UK	2013	20748657
M	Y_GE65	UK	2013	4917238
F	Y_LT15	UK	2013	5488356
F	Y15-64	UK	2013	20915924
F	Y_GE65	UK	2013	6068452

5427

5428

Dr				
SEX	AGE	GEO	TIME	POPULATION
F	Y_LT15	BE	2013	927644
F	Y15-64	BE	2013	3625561
F	Y_GE65	BE	2013	1121001
F	Y_LT15	UK	2013	5488356
F	Y15-64	UK	2013	20915924
F	Y_GE65	UK	2013	6068452

5429

5430 **keep**

5431 *Semantics*

5432 The operator **keep** returns the input Dataset with only the Identifier and Measures Components specified as  
 5433 parameters.

5434  
 5435 *Syntax*

5436 *ds* [ **keep** *cmp* {, *cmp*} ]

5437  
 5438 *Parameters*

5439 *ds* : dataset {identifier <IDENT> as scalar-type}<sup>+</sup> {measure <IDENT> as scalar-type}<sup>\*</sup>  
 5440 {attribute <IDENT> as scalar-type}<sup>\*</sup>

5441 *cmp* : component-ref

- 5442
- 5443 • *ds* – is the input Dataset.
- 5444 • *cmp* – is an existing Component of *ds*.
- 5445

5446 *Constraints*

- 5447 • *cmp* is a Component expression over the Components of *ds* containing only Component literals (i.e. names of  
 5448 the Components of *ds*) (static).
- 5449 • *cmp* cannot be a reference to an Identifier Component.
- 5450

5451 *Returns*

5452 A Dataset having all the Identifier Components of *ds* and the Measure Components and Attribute Components  
 5453 selected in *cmp*.

5454  
 5455 *Semantic specification*

5456 The operator takes as input a Dataset *ds* and a subset of the Components owned by *ds*, it returns another Dataset  
 5457 having all the Identifier Components of the input one (Identifier Components are not affected by the **keep**) and  
 5458 all the Measure Components and Attribute Components selected in *cmp*.

5459  
 5460 *Examples*

5461 *ds*\_1 := population1[**keep** SEX, GEO, POPULATION ]

5462

population1				
SEX	AGE	GEO	TIME	POPULATION
M	Y_LT15	BE	2013	970428
M	Y15-64	BE	2013	3678355
M	Y_GE65	BE	2013	838653
F	Y_LT15	BE	2013	927644
F	Y15-64	BE	2013	3625561
F	Y_GE65	BE	2013	1121001
M	Y_LT15	UK	2013	5757444
M	Y15-64	UK	2013	20748657
M	Y_GE65	UK	2013	4917238
F	Y_LT15	UK	2013	5488356
F	Y15-64	UK	2013	20915924
F	Y_GE65	UK	2013	6068452

5463

population1		
SEX	GEO	POPULATION
M	BE	970428



M	BE	3678355
M	BE	838653
F	BE	927644
F	BE	3625561
F	BE	1121001
M	UK	5757444
M	UK	20748657
M	UK	4917238
F	UK	5488356
F	UK	20915924
F	UK	6068452

5464 **calc**

5465 *Semantic*

5466 The operator **calc** returns the input Dataset with new components calculated based on the expressions specified  
5467 as parameters.

5468 *Syntax*

5470 ***ds* [calc *k as compName* {role [Measure | Identifier | Attribute]} {viral }  
5471 {, *k as compName* {role [Measure | Identifier | Attribute]} {viral} ]\***  
5472 ]

5473 *Parameters*

5474 *ds* : dataset {identifier <IDENT> as scalar-type}<sup>+</sup> {measure <IDENT> as scalar-type}<sup>\*</sup>  
5475 {attribute <IDENT> as scalar-type}<sup>\*</sup>

5476 *k* : expr

5477 *compName* : ident

5478 *role* : constant

5480

- 5481 • *ds* – is the input Dataset.
- 5482 • *k* – is an expression involving ds Components.
- 5483 • *role* – is the role of the calculated Component.
- 5484 • *compName* – is the name of the new Component.

5485 *Constraints*

- 5486 • *role* can be one of: “Measure”, “Identifier”, “Attribute”.
- 5487 • *k* is an expression on ds Components.

5488 *Return*

5489 A Dataset having the same Identifier and Measure Components of the input one, enriched by others Components  
5490 calculated using the defined *k* expressions.

5491 *Semantic specification*

5492 The operator takes in as input a Dataset *ds* and a series of expressions to calculate new Components, and returns  
5493 a new Dataset with the same Identifier and Measure Components.

5494 It adds to the output Dataset a Component for each Component expression *k* specified in the clause, calculating it  
5495 row-wise according to the Component expression.

5496 The added Component is named *compName* and is given a role (Identifier, Measure or Attribute Component)  
5497 according to *role* Constant. If the role is omitted, “MEASURE” is implied.

5500 If any *k* coincides with the name of an existing Component in *ds* (even with different type), the calculated one  
5501 replaces the former, in name, value and type.

5502 Special care must be paid to the handling of Attribute Components. If a Component expression has the same  
5503 name as an existing Attribute Component, the previous one is overridden, independently of its virality. In this  
5504

5505 sense, **calc** clause overrides virality. On the other hand, if no Attribute Component expressions override an  
5506 existing Component, it will be kept in the result, only if viral, with unaltered virality. In general, when an  
5507 Attribute Component is calculated, its virality can be set by the use of keyword **viral**. If it is omitted, the Attribute  
5508 Component is non **viral** by default. As a special case of this, a **calc** can be also used simply to alter the virality of  
5509 an Attribute Component.

5510

#### 5511 *Examples*

5512 1)

5513 `ds_2 := ds_1[calc M1*M2/3 as "M4" role MEASURE]`

5514 The expression above calculates a Measure Component by combining the ones of the involved Datasets.

5515 2)

5516 `ds_2 := ds_1[calc M1-1 as "M1" role MEASURE, M1+M2 as M2, if M2>3 then M2 else M3 as M3]`

5517 Like the preceding example, but with a conditional logic.

5518 3)

5519 `ds_2 := ds_1[calc A1 + A2 as "A3" role ATTRIBUTE viral]`

5520 The expression above calculates Attribute Component A3 as a combination of A1 and A2.

5521

## 5522 **attrcalc**

5523

#### 5523 *Semantics*

5524 The operator **attrcalc** returns the input Dataset with new Attribute components calculated based on the

5525 expressions specified as parameters.

5526

#### 5527 *Syntax*

5528 `ds [attrcalc k as compName {viral } {, k as compName {viral } }*]`

5529

#### 5530 *Parameters*

5531 `ds` : dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as scalar-type}\*  
5532 {attribute <IDENT> as scalar-type}+

5533

5534 `k` : expr

5535

5534 `compName` : ident

5536

- `ds` – is the input Dataset, containing Attribute Components.
- `k` – is an expression involving `ds` Components.
- `role` – is the role of the calculated Component.
- `compName` – is the name of the new Attribute Component.

5539

5540

#### 5541 *Constraints*

5542 `k` is an expression on `ds` Components or on Components used to calculate `ds` properly qualified (static).

5543

#### 5544 *Returns*

5545 A Dataset with all the Identifier and Measure Components of the input one, and an Attribute Component, for each

5546 expression `k` specified, named `compName`.

5547

#### 5548 *Semantic specification*

5549 The operator takes as input a Dataset `ds` (which, in general, can be a complex expression of type Dataset) and  
5550 returns a new Dataset `ds` with the same Identifier Components and Measure Components.

5551

5552 The output Dataset has an Attribute Component named `compName` for each Attribute Component expression `k`  
5553 specified in the clause. `k` is a component expression, evaluated row-wise.

5554

5555 Special care must be paid to the handling of Attribute Components. If a Component expression has the same  
5556 name as an existing Attribute Component, the previous one is overridden, independently of its virality. In this  
5557 sense, **attrcalc** clause overrides virality. On the other hand, if no Attribute Component expressions override an  
5558 existing Component, it will be kept in the result, only if viral, with unaltered virality. In general, when an  
5559 Attribute Component is calculated, its virality can be set by the use of keyword **viral**. If it is omitted, the Attribute  
5560 Component is non viral by default. As a special case of this, an **attrcalc** can be also used simply to alter the  
virality of an Attribute Component.

5560

5561 *Examples*

5562 1) `ds_2 := ds_1[attrcalc QUALITY+1 as QUALITY]`

5563 The expression calculates `ds_2`, keeping the QUALITY Attribute Component in `ds_1`, but adding 1 to its value.

5564

ds_1			
K1	K2	M1	QUALITY
1	A	1	1
2	B	3	2
3	C	5	3

5565

ds_2			
K1	K2	M1	QUALITY
1	A	1	2
2	B	3	3
3	C	5	4

5566

5567 2) `ds_r := (ds_1 + ds_2)[attrcalc ds_1.QUALITY + ds_2.QUALITY as QUALITY]`

5568

5569 The expression calculates `ds_r` as the sum of Datasets `ds_1` and `ds_2`. Besides, it calculates the QUALITY attribute

5570 as the sum of the two.

5571

ds_1			
K1	K2	M1	QUALITY
1	A	1	1
2	B	3	2
3	C	5	3

5572

ds_2			
K1	K2	M1	QUALITY
1	A	6	2
2	B	7	3
3	C	8	4

5573

ds_r			
K1	K2	M1	QUALITY
1	A	7	3
2	B	10	5
3	C	13	7

5574

5575 3)

5576 `ds_r := (ds_1 + ds_2)`

5577 [

5578 `attrcalc if ds_1.QUALITY="A" and ds.2QUALITY="B" then "C"`

5579 `elseif ds_1.QUALITY="K" and ds.2QUALITY="K" then "M"`

5580 `else "Z" AS AGGREGATED_QUALITY`

5581 ]

5582 The expression calculates the AGGREGATED\_QUALITY attribute as a combination of QUALITY Attribute  
 5583 Components of the operands according to a decision rule.  
 5584 In particular, if *ds\_1* quality is "A" and *ds\_2* quality is "B", then the AGGREGATED\_QUALITY will be "C". Else, if *ds\_1*  
 5585 quality is "K" and *ds\_2* quality is "K", then "M" is returned. Otherwise "Z" is the AGGREGATED\_QUALITY value.  
 5586

ds_1			
K1	K2	M1	QUALITY
1	A	1	A
2	B	3	B
3	C	5	K

5587

ds_2			
K1	K2	M1	QUALITY
1	A	6	B
2	B	7	A
3	C	8	K

5588

ds_r			
K1	K2	M1	QUALITY
1	A	7	C
2	B	10	Z
3	C	13	M

5589

5590

5591

5592

5593

5594

4)

$ds_r := (ds_1 + ds_2)[\text{attrcalc } ds_1.QUALITY_1 + ds_2.QUALITY_1 \text{ as } QUALITY_1, ds_2.QUALITY_2 \text{ as } QUALITY_2]$

The expression sums two multi-measure Datasets and calculates two Attribute Components with different formulas: QUALITY\_1 is the sum of *ds\_1*.QUALITY\_1 and *ds\_2*.QUALITY\_1, while QUALITY\_2 is simply copied from *ds\_2*.

ds_1					
K1	K2	M1	M2	QUALITY_1	QUALITY_2
1	A	1	5	1	2
2	B	3	3	2	7
3	C	5	1	3	4

5595

ds_2					
K1	K2	M1	M2	QUALITY_1	QUALITY_2
1	A	6	1	2	1
2	B	7	1	3	3
3	C	8	1	4	1

5596

ds_r					
K1	K2	M1	M2	QUALITY_1	QUALITY_2
1	A	7	6	3	1
2	B	10	4	5	3
3	C	13	2	7	1

5597