VTL 2.1 DOCS

version

SDMX-TWG

December 19, 2024

Contents

Documentation for VTL v2.1 **1**

User Manual 1

Foreword 1

Introduction 2

Structure of the document 2

General characteristics of the VTL 3

User orientation 3

Integrated approach 3

Active role for processing 4

Independence of IT implementation 5

Extensibility, customizability 6

Language effectiveness 7

Evolution of VTL 2.0 in respect to VTL 1.0 7

The Information Model 7

Structural artefacts and reusable rules 8

The core language and the standard library 8

The user defined operators 8

The VTL Definition Language 8

The functional paradigm 9

The operators 9

Changes for version 2.1 9

VTL Information Model 10

Introduction 10

Generic Model for Data and their structures 11

Data model diagram 12

Explanation of the Diagram 13

Functional Integrity 13

Examples 14

The data artefacts 16

Generic Model for Variables and Value Domains 16

Variable and Value Domain model diagram 17

Explanation of the Diagram 17

Relations and operations between Code Items 19

Conditioned Code Item Relations 21

The historical changes 21

The Variables and Value Domains artefacts 22

Generic Model for Transformations 24

Transformations model diagram 26

Explanation of the diagram 26

Examples 27

Functional paradigm 27

Transformation Consistency 27

VTL Data types 29

Data Types overview 30

Data Types model diagram 30
Explanation of the diagram 30
General conventions for describing the types 31
Scalar Types 31
Basic Scalar Types 31
Value Domain Scalar Types 33
Set Scalar Types 34
External representations and literals used in the VTL Manuals 34
Conventions for describing the scalar types 36
Compound Data Types 37
Component Types 38
Data Set Types 39
Product Types 40
Operator Types 41
Ruleset Types 41
Universal Set Types 42
Universal List Types 42
VTL Transformations 43
The Expression 43
The Assignment 45
The Result 45
The names 46
The artefact names 46
The environment name 47
The connection to the persistent storage 47
VTL Operators 47
The categories of VTL operators 48
The input parameters 48
The invocation of VTL operators 49
Level of operation 50
The Operators' behaviour 51
The Join operators 51
Other operators: default behaviour on Identifiers, Measures and Attributes 52
The Identifier Components and the Data Points matching 52
The operations on the Measure Components 54
Operators which change the basic scalar type 58
Boolean operators 60
Set operators 60
Behaviour for Missing Data 60
Behaviour for Attribute Components 61
The Attribute propagation rule 62
Properties of the Attribute propagation algorithm 64
Governance, other requirements and future work 64
The governance of the extensions 65
Relations with the GSIM Information Model 65
Data Sets and Data Structures 66

Value Domains 67
Transformation model and Business Process Model 67
Annex 1 – EBNF 68
Properties of VTL grammar 68
Reference Manual 68
Foreword 68
Introduction 69
Overwiew of the language and conventions 70
Introduction 70
Conventions for writing VTL Transformations 71
Typographical conventions 72
Abbreviations for the names of the artefacts 72
Conventions for describing the operators' syntax 73
Description of data types of operands and result 74
VTL-ML Operators 75
VTL-ML - Evaluation order of the Operators 81
Description of VTL Operators 82
VTL-DL - Rulesets 83
define datapoint ruleset 83
Semantics 83
Syntax 83
Syntax description 83
Parameters 85
Constraints 85
Semantic specification 85
Examples 86
define hierarchical ruleset 87
Semantics 87
Syntax 87
Syntax description 88
Input parameters type 90
Constraints 91
Semantic specification 91
Examples 95
VTL-DL – User Defined Operators 96
define operator 96
Syntax 96
Syntax description 96
Input parameters type 96
Constraints 96
Semantic specification 97
Examples 97
Data type syntax 97
VTL-ML - Typical behaviours of the ML Operators 98
Typical behaviour of most ML Operators 98
Operators applicable on one Scalar Value or Data Set or Data Set Component 98

Operators applicable on two Scalar Values or Data Sets or Data Set Components 99
Operators applicable on more than two Scalar Values or Data Set Components 101
Behaviour of Boolean operators 102
Behaviour of Set operators 102
Behaviour of Time operators 102
Operators changing the data type 103
Type Conversion and Formatting Mask 104
The Numbers Formatting Mask 104
The Time Formatting Mask 104
Attribute propagation 107
Operators 108
VTL-ML - General Purpose Operators 108
Parentheses: *()* 108
Syntax 108
Input parameters 108
Examples of valid syntaxes 108
Semantics for scalar operations 108
Input parameters type 108
Result type 109
Additional Constraints 109
Behaviour 109
Examples 109
Example 1 109
Persistent assignment: <- 109
Syntax 109
Input parameters 110
Examples of valid syntaxes 110
Semantics for scalar operations 110
Input parameters type 110
Result type 110
Additional Constraints 110
Behaviour 110
Examples 110
Example 1 111
Non-persistent assignment: :=` 111
Syntax 111
Input parameters 111
Examples of valid syntaxes 111
Semantics for scalar operations 111
Input parameters type 111
Result type 111
Additional Constraints 112
Behaviour 112
Examples 112
Example 1 112
Membership: # 112

Syntax 112
Input parameters 113
Semantics for scalar operations 113
Input parameters type 113
Result type 113
Additional Constraints 113
Behaviour 113
Examples 113
Example 1 114
Example 2 114
Example 3 114
Example 4 114
Example 5 115
Example 6 115
User-defined operator call 115
Syntax 115
Input parameters 115
Examples of valid syntaxes 115
Semantics for scalar operations 115
Input parameters type 115
Result type 116
Additional Constraints 116
Behaviour 116
Examples 116
Example 1 116
Evaluation of an external routine: *eval* 116
Syntax 116
Input parameters 116
Examples of valid syntaxes 116
Semantics for scalar operations 117
Input parameters type 117
Result type 117
Additional Constraints 117
Behaviour 117
Examples 117
Type conversion: *cast* 118
Syntax 118
Input parameters 118
Semantics for scalar operations 118
Input parameters type 118
Result type 118
Additional Constraints 118
Behaviour 118
Examples 121
Example 1 121
Example 2 121

Example 3 121
Example 4 121
Example 5 121
Example 6 121
Example 7 122
Example 8 122
Example 9 122
VTL-ML - Join operators 122
Join: *inner_join*, *left_join*, *full_join*, *cross_join* 122
Syntax 122
Input parameters 123
Examples of valid syntaxes 125
Semantics for scalar operations 125
Input parameters type 125
Result type 126
Additional Constraints 126
Behaviour 127
Examples 130
Example 1 131
Example 2 131
Example 3 131
Example 4 131
Example 5 132
Example 6 132
Example 7 132
VTL-ML - String Operators 133
String concatenation: \ 133
Syntax 133
Input parameters 133
Semantics for scalar operations 133
Input parameters type 133
Result type 133
Additional Constraints 133
Behavior 133
Examples 133
Example 1 133
Example 2 134
Whitespace removal: *trim*, *rtrim*, *ltrim* 134
Syntax 134
Input parameters 134
Semantics for scalar operations 134
Input parameters type 134
Result type 134
Additional Constraints 134
Behavior 135
Examples 135

Example 1 135
Example 2 135
Character case conversion: *upper/lower* 135
Syntax 135
Input parameters 135
Examples of valid syntaxes 135
Semantics for scalar operations 136
Input parameters type 136
Result type 136
Additional Constraints 136
Behavior 136
Examples 136
Example 1 136
Example 2 136
Sub-string extraction: *substr* 137
Syntax 137
Input parameters 137
Examples of valid syntaxes 137
Semantics for scalar operations 137
Input parameters type 137
Result type 138
Additional Constraints 138
Behavior 138
Examples 138
Example 1 138
Example 2 138
Example 3 139
String pattern replacement: *replace* 139
Syntax 139
Input parameters 139
Examples of valid syntaxes 139
Semantics for scalar operations 139
Input parameters type 139
Result type 139
Additional Constraints 140
Behaviour 140
Examples 140
Example 1 140
Example 2 140
String pattern location: *instr* 141
Syntax 141
Input parameters 141
Examplesof valid syntaxes 141
Semantics for scalar operations 141
Input parameters type 141
Result type 142

Additional Constraints 142
Behaviour 142
Examples 142
Example 1 142
Example 2 142
Example 3 143
Example 4 143
String length: *length* 143
Syntax 143
Input parameters 143
Examples of valid syntaxes 143
Semantics for scalar operations 143
Input parameters type 143
Result type 144
Additional Constraints 144
Behaviour 144
Examples 144
Example 1 144
Example 2 144
Example 3 145
Example 4 145
VTL-ML - Numeric Operators 145
Unary Plus: + 145
Syntax 145
Input parameters 145
Examples of valid syntaxes 145
Semantics for scalar operations 145
Input parameters type 145
Result type 146
Additional Constraints 146
Behaviour 146
Examples 146
Example 1 146
Example 2 146
Unary Minus: - 147
Syntax 147
Input parameters 147
Examples of valid syntaxes 147
Semantics for scalar operations 147
Input parameters type 147
Result type 147
Additional Constraints 147
Behaviour 147
Examples 147
Example 1 148
Example 2 148

Addition: + 148
Syntax 148
Input parameters 148
examples of valid syntaxes 148
Semantics for scalar operations 148
Input parameters type 148
Result type 149
Additional Constraints 149
Behaviour 149
Examples 149
Example 1 149
Example 2 149
Example 3 150
Subtraction: - 150
Syntax 150
Input parameters 150
Examples of valid syntaxes 150
Semantics for scalar operations 150
Input parameters type 150
Result type 151
Additional Constraints 151
Behaviour 151
Examples 151
Example 1 151
Example 2 151
Example 3 152
Multiplication: * 152
Syntax 152
Input parameters 152
Examples of valid syntaxes 152
Semantics for scalar operations 152
Input parameters type 152
Result type 153
Additional Constraints 153
Behavior 153
Examples 153
Example 1 153
Example 2 153
Example 3 154
Division: / 154
Syntax 154
Input parameters 154
Examples of valid syntaxes 154
Semantics for scalar operations 154
Input parameters type 154
Result type 154

Additional Constraints 155
Behavior 155
Examples 155
Example 1 155
Example 2 156
Example 3 156
Modulo: *mod* 156
Syntax 156
Input parameters 156
Examples of valid syntaxes 156
Semantics for scalar operations 156
Input parameters type 157
Result type 157
Additional Constraints 157
Behavior 157
Examples 157
Example 1 157
Example 2 158
Example 3 158
Rounding: *round* 158
Syntax 158
Input parameters 158
Examples of valid syntaxes 158
Semantics for scalar operations 159
Input parameters type 159
Result type 159
Additional Constraints 159
Behavior 159
Examples 159
Example 1 160
Example 2 160
Example 3 160
Truncation: *trunc* 160
Syntax 160
Input parameters 160
Examples of valid syntaxes 161
Semantics for scalar operations 161
Input parameters type 161
Result type 161
Additional Constraints 161
Behavior 161
Examples 161
Example 1 162
Example 2 162
Example 3 162
Ceiling: *ceil* 163

Syntax 163
Input parameters 163
Examples of valid syntaxes 163
Semantics for scalar operations 163
Input parameters type 163
Result type 163
Additional Constraints 163
Behaviour 163
Examples 163
Example 1 164
Example 2 164
Floor: *floor* 164
Syntax 164
Input parameters 164
Examples of valid syntaxes 164
Semantics for scalar operations 164
Input parameters type 165
Result type 165
Additional Constraints 165
Behaviour 165
Examples 165
Example 1 165
Example 2 165
Absolute value: *abs* 166
Syntax 166
Input parameters 166
Examples of valid syntaxes 166
Semantics for scalar operations 166
Input parameters type 166
Result type 166
Additional Constraints 166
Behavior 166
Examples 166
Example 1 167
Example 2 167
Exponential: *exp* 167
Syntax 167
Input parameters 167
Examples of valid syntaxes 167
Semantics for scalar operations 167
Input parameters type 168
Result type 168
Additional Constraints 168
Behavior 168
Examples 168
Example 1 168

Example 2 168
Natural logarithm: *ln* 169
Syntax 169
Input parameters 169
Examples of valid syntaxes 169
Semantics for scalar operations 169
Input parameters type 169
Result type 169
Additional Constraints 169
Behaviour 169
Examples 169
Example 1 170
Example 2 170
Power: *power* 170
Syntax 170
Input parameters 170
Examples of valid syntaxes 170
Semantics for scalar operations 170
Input parameters type 171
Result type 171
Additional Constraints 171
Behaviour 171
Examples 171
Example 1 171
Example 2 172
Logarithm: *log* 172
Syntax 172
Input parameters 172
Examples of valid syntaxes 172
Semantics for scalar operations 172
Input parameters type 172
Result type 172
Additional Constraints 173
Behavior 173
Examples 173
Example 1 173
Example 2 173
Square root: *sqrt* 173
Syntax 173
Input parameters 174
Examples of valid syntaxes 174
Semantics for scalar operations 174
Input parameters type 174
Result type 174
Additional Constraints 174
Behaviour 174

Examples 174
Example 1 174
Example 2 175
Random: *random* 175
Syntax 175
Input parameters 175
Examples of valid syntaxes 175
Semantics for scalar operations 175
Input parameters type 175
Result type 175
Additional Constraints 176
Behaviour 176
Examples 176
Example 1 176
Example 2 176
VTL-ML - Comparison Operators 177
Equal to: = 177
Syntax 177
Input parameters 177
Examples of valid syntaxes 177
Semantics for scalar operations 177
Input parameters type 177
Result type 177
Additional Constraints 177
Behaviour 177
Examples 177
Example 1 177
Example 2 178
Not equal to: <> 178
Syntax 178
Input parameters 178
Examples of valid syntaxes 178
Semantics for scalar operations 178
Input parameters type 178
Result type 179
Additional Constraints 179
Behaviour 179
Examples 179
Example 1 179
Example 2 179
Greater than: > >= 180
Syntax 180
Input parameters 180
Examples of valid syntaxes 180
Semantics for scalar operations 180
Input parameters type 180

Result type 180
Additional Constraints 180
Behaviour 180
Examples 180
Example 1 181
Example 2 181
Example 3 181
Less than < <= 182
Syntax 182
Input parameters 182
Examples of valid syntaxes 182
Semantics for scalar operations 182
Input parameters type 182
Result type 182
Additional Constraints 182
Behaviour 182
Examples 182
Example 1 183
Between *between* 183
Syntax 183
Input parameters 183
Examples of valid syntaxes 183
Semantics for scalar operations 183
Input parameters type 184
Result type 184
Additional Constraints 184
Behaviour 184
Examples 184
Example 1 184
Element of *in* / *not_in* 184
Syntax 184
Input parameters 185
Examples of valid syntaxes 185
Semantics for scalar operations 185
Input parameters type 185
Result type 185
Additional Constraints 185
Behavior 185
Examples 186
Example 1 186
Example 2 186
Example 3 186
Match characters: *match_characters* 187
Syntax 187
Input parameters 187
Examples of valid syntaxes 187

Semantics for scalar operations 187
Input parameters type 187
Result type 187
Additional Constraints 188
Behaviour 188
Examples 188
Example 1 188
Is null: *isnull* 188
Syntax 188
Input parameters 188
Examples of valid syntaxes 188
Semantics for scalar operations 188
Input parameters type 188
Result type 189
Additional Constraints 189
Behaviour 189
Examples 189
Example 1 189
Example 2 189
Exists in: *exists_in* 190
Syntax 190
Input parameters 190
Examples of valid syntaxes 190
Semantics for scalar operations 190
Input parameters type 190
Result type 190
Additional Constraints 190
Behaviour 191
Examples 191
Example 1 191
Example 2 192
Example 3 192
VTL-ML - Boolean Operators 192
Logical conjunction: *and* 192
Syntax 192
Input parameters 192
Examples of valid syntaxes 192
Semantics for scalar operations 192
Input parameters type 193
Result type 193
Additional Constraints 193
Behavior 193
Examples 193
Example 1 193
Example 2 194
Logical disjunction: *or* 194

Syntax 194
Input parameters 194
Examples of valid syntaxes 194
Semantics for scalar operations 194
Input parameters type 194
Result type 195
Additional Constraints 195
Behavior 195
Examples 195
Example 1 195
Example 2 196
Exclusive disjunction: *xor* 196
Syntax 196
Input parameters 196
Examples of valid syntaxes 196
Semantics for scalar operations 196
Input parameters type 196
Result type 197
Additional Constraints 197
Behaviour 197
Examples 197
Example 1 197
Example 2 198
Logical negation: *not* 198
Syntax 198
Input parameters 198
Examples of valid syntaxes 198
Semantics for scalar operations 198
Input parameters type 198
Result type 198
Additional Constraints 199
Behaviour 199
Examples 199
Example 1 199
Example 2 199
VTL-ML - Time Operators 200
Period indicator: *period_indicator* 200
Syntax 200
Input parameters 200
Examples of valid syntaxes 200
Semantics for scalar operations 200
Input parameters type 200
Result type 201
Additional Constraints 201
Behaviour 201
Examples 201

Example 1 201
Example 2 201
Fill time series: *fill_time_series* 202
Syntax 202
Input parameters 202
Examples of valid syntaxes 202
Semantics for scalar operations 202
Input parameters type 202
Result type 202
Additional Constraints 202
Behaviour 202
Examples 203
Example 1 204
Example 2 204
Example 3 205
Example 4 205
Example 5 205
Example 6 206
Example 7 206
Example 8 207
Flow to stock: *flow_to_stock* 207
Syntax 207
Input parameters 207
Examples of valid syntaxes 207
Semantics for scalar operations 207
Input parameters type 207
Result type 208
Additional Constraints 208
Behavior 208
Examples 208
Example 1 209
Example 2 210
Example 3 210
Example 4 210
Stock to flow: *stock_to_flow* 211
Syntax 211
Input parameters 211
Examples of valid syntaxes 211
Semantics for scalar operations 211
Input parameters type 211
Result type 211
Additional Constraints 211
Behaviour 211
Examples 212
Example 1 213
Example 2 213

Example 3 214
Example 4 214
Time shift: *timeshift* 215
Syntax 215
Input parameters 215
Examples of valid syntaxes 215
Semantics for scalar operations 215
Input parameters type 215
Result type 215
Additional Constraints 215
Behavior 215
Examples 215
Example 1 217
Example 2 217
Example 3 218
Example 4 218
Time aggregation: *time_agg* 218
Syntax 218
Input parameters 218
Examples of valid syntaxes 219
Semantics for scalar operations 219
Input parameters type 219
Result type 219
Additional Constraints 219
Behaviour 219
Examples 220
Example 1 220
Example 2 220
Example 3 220
Example 4 220
Actual time: *current_date* 221
Syntax 221
Input parameters 221
Examples of valid syntaxes 221
Semantics for scalar operations 221
Input parameters type 221
Result type 221
Additional Constraints 221
Behavior 221
Examples 221
Example 1 221
Example 2 221
Days between two dates: *datediff* 221
Syntax 221
Input parameters 221
Examples of valid syntaxes 222

Semantics for scalar operations 222
Input parameters type 222
Result type 222
Additional Constraints 222
Behaviour 222
Examples 222
Example 1 222
Example 2 223
Add a time unit to a dete: *dateadd* 223
Syntax 223
Input parameters 223
Examples of valid syntaxes 223
Semantics for scalar operations 223
Input parameters type 223
Result type 223
Additional Constraints 224
Behaviour 224
Examples 224
Example 1 224
Example 2 224
Extract time period from a date: *year, month, dayofmonth, dayofyear* 224
Syntax 224
Input parameters 224
Examples of valid syntaxes 225
Semantics for scalar operations 225
Input parameters type 225
Result type 225
Additional Constraints 225
Behaviour 225
Examples 225
Example 1 225
Example 2 226
Number of days to duration: *daytoyear, daytomonth* 226
Syntax 226
Input parameters 226
Examples of valid syntaxes 226
Semantics for scalar operations 226
Input parameters type 226
Result type 226
Additional Constraints 226
Behaviour 226
Examples 227
Example 1 227
Example 2 227
Example 2 227
Example 3 227

Duration to number of days: *yeartoday, monthtoday* 227
Syntax 227
Input parameters 228
Examples of valid syntaxes 228
Semantics for scalar operations 228
Input parameters type 228
Result type 228
Additional Constraints 228
Behaviour 228
Examples 228
Example 1 229
Example 2 229
Example 3 229
VTL-ML - Set Operators 229
Union: *union* 229
Syntax 229
Input parameters 229
Examples of valid syntaxes 229
Semantics for scalar operations 229
Input parameters type 229
Result type 229
Additional Constraints 230
Behaviour 230
Examples 230
Example 1 230
Example 2 231
Intersection: *interesect* 231
Syntax 231
Input parameters 231
Examples of valid syntaxes 231
Semantics for scalar operations 231
Input parameters type 231
Result type 231
Additional Constraints 231
Behavior 231
Examples 232
Example 1 232
Set difference: *setdiff* 232
Syntax 232
Input parameters 232
Examples of valid syntaxes 232
Semantics for scalar operations 232
Input parameters type 233
Result type 233
Additional Constraints 233
Behavior 233

Examples 233
Example 1 234
Example 2 234
Symmetric difference: *symdiff* 234
Syntax 234
Input parameters 234
Semantics for scalar operations 234
Input parameters type 234
Result type 234
Additional Constraints 235
Behaviour 235
Examples 235
Example 1 235
VTL-ML - Hierarchical aggregation 236
Hierarchical roll-up: *hierarchy* 236
Syntax 236
Input parameters 236
Examples of valid syntaxes 236
Semantics for scalar operations 236
Input parameters type 236
Result type 237
Additional Constraints 237
Behaviour 237
Examples 239
Example 1 240
Example 2 240
Example 3 241
VTL-ML - Aggregate and Analytic operators 241
Aggregate invocation 242
Syntax 242
Input parameters 242
Examples of valid syntaxes 244
Semantics for scalar operations 244
Input parameters type 244
Result type 244
Additional Constraints 245
Behaviour 245
Examples 245
Example 1 246
Example 2 246
Example 3 247
Example 4 247
Analytic invocation 247
Syntax 247
Input parameters 248
Examples of valid syntaxes 249

Semantics for scalar operations 249
Input parameters type 250
Result type 250
Additional Constraints 250
Behaviour 250
Examples 251
Example 1 251
Counting the number of data points: *count* 252
Syntax 252
Input parameters 252
Semantics for scalar operations 252
Input parameters type 252
Result type 252
Additional Constraints 252
Behaviour 252
Examples 252
Example 1 253
Example 2 253
Minimun value: *min* 253
Syntax 253
Input parameters 253
Semantics for scalar operations 253
Input parameters type 254
Result type 254
Additional Constraints 254
Behaviour 254
Examples 254
Example 1 254
Maximum value: *max* 254
Syntax 254
Input parameters 255
Semantics for scalar operations 255
Input parameters type 255
Result type 255
Additional Constraints 255
Behaviour 255
Examples 255
Example 1 256
Median value: *median* 256
Syntax 256
Input parameters 256
Semantics for scalar operations 256
Input parameters type 256
Result type 256
Additional Constraints 256
Behaviour 257

Examples 257
Example 1 257
Sum: *sum* 257
Syntax 257
Input parameters 257
Semantics for scalar operations 257
Input parameters type 258
Result type 258
Additional Constraints 258
Behaviour 258
Examples 258
Example 1 258
Average value: *avg* 258
Syntax 258
Input parameters 259
Semantics for scalar operations 259
Input parameters type 259
Result type 259
Additional Constraints 259
Behaviour 259
Examples 259
Example 1 259
Population standard deviation: *stddev_pop* 260
Syntax 260
Input parameters 260
Semantics for scalar operations 260
Input parameters type 260
Result type 260
Additional Constraints 260
Behaviour 260
Examples 261
Example 1 261
Sample standard deviation: *stddev_samp* 261
Syntax 261
Input parameters 261
Semantics for scalar operations 261
Input parameters type 261
Result type 262
Additional Constraints 262
Behaviour 262
Examples 262
Example 1 262
Population variance: *var_pop* 262
Syntax 262
Input parameters 263
Semantics for scalar operations 263

Input parameters type 263
Result type 263
Additional Constraints 263
Behaviour 263
Examples 263
Example 1 263
Sample variance: *var_samp* 264
Syntax 264
Input parameters 264
Semantics for scalar operations 264
Input parameters type 264
Result type 264
Additional Constraints 264
Behaviour 264
Examples 265
Example 1 265
First value: *first_value* 265
Syntax 265
Input parameters 265
Semantics for scalar operations 265
Input parameters type 265
Result type 265
Additional Constraints 266
Behaviour 266
Examples 266
Example 1 266
Last value: *last_value* 267
Syntax 267
Input parameters 267
Semantics for scalar operations 267
Input parameters type 267
Result type 267
Additional Constraints 267
Behaviour 267
Examples 267
Example 1 268
Lag: *lag* 268
Syntax 268
Input parameters 268
Semantics for scalar operations 268
Input parameters type 268
Result type 269
Additional Constraints 269
Behaviour 269
Examples 269
Example 1 269

Lead: *lead* 270
Syntax 270
Input parameters 270
Semantics for scalar operations 270
Input parameters type 270
Result type 270
Additional Constraints 271
Behaviour 271
Examples 271
Example 1 271
Rank: *rank* 272
Syntax 272
Input parameters 272
Semantics for scalar operations 272
Input parameters type 272
Result type 272
Additional Constraints 272
Behaviour 272
Examples 272
Example 1 273
Ratio to report: *ratio_to_report* 273
Syntax 273
Input parameters 273
Semantics for scalar operations 273
Input parameters type 273
Result type 273
Additional Constraints 274
Behaviour 274
Examples 274
Example 1 274
VTL-ML - Data Validation Operators 275
Check datapoint: *check_datapoint* 275
Syntax 275
Input parameters 275
Examples of valid syntaxes 275
Semantics for scalar operations 275
Input parameters type 275
Result type 276
Additional Constraints 276
Behaviour 276
Examples 276
Example 1 276
Example 2 277
Check hierarchy: *check_hierarchy* 277
Syntax 277
Input parameters 277

Examples of valid syntaxes 278
Input parameters type 278
Result type 278
Additional Constraints 278
Behaviour 279
Examples 282
Example 1 283
Check : *check* 283
Syntax 283
Input parameters 283
Examples of valid syntaxes 284
Input parameters type 284
Result type 284
Additional Constraints 284
Behaviour 284
Examples 284
Example 1 285
VTL-ML - Conditional Operators 286
if-then-else: *if* 286
Syntax 286
Input parameters 286
Examples of valid syntaxes 286
Semantics for scalar operations 286
Input parameters type 286
Result type 286
Additional Constraints 287
Behaviour 287
Examples 287
Example 1 288
Case: *case* 288
Syntax 288
Input parameters 288
Examples of valid syntaxes 288
Semantics for scalar operations 289
Input parameters type 289
Result type 289
Additional Constraints 289
Behaviour 289
Examples 290
Example 1 290
Nvl: *nvl* 290
Syntax 290
Input parameters 290
Examples of valid syntaxes 290
Semantics for scalar operations 290
Input parameters type 291

Result type 291
Additional Constraints 291
Behaviour 291
Examples 291
Example 1 291
VTL-ML - Clause Operators 292
Filtering Data Points: *filter* 292
Syntax 292
Input parameters 292
Examples of valid syntaxes 292
Semantics for scalar operations 292
Input parameters type 292
Result type 292
Additional Constraints 292
Behavior 292
Examples 293
Example 1 293
Calculation of a Component: *calc* 293
Syntax 293
Input parameters 293
Examples of valid syntaxes 293
Semantics for scalar operations 293
Input parameters type 294
Result type 294
Additional Constraints 294
Behavior 294
Examples 294
Example 1 294
Example 2 295
Aggregation: *aggr* 295
Syntax 295
Input parameters 295
Examples of valid syntaxes 297
Semantics for scalar operations 297
Input parameters type 297
Result type 297
Additional Constraints 297
Behaviour 297
Examples 298
Example 1 298
Example 2 298
Example 3 299
Maintaining Components: *keep* 299
Syntax 299
Input parameters 299
Examples of valid syntaxes 299

Semantics for scalar operations 299
Input parameters type 299
Result type 299
Additional Constraints 300
Behaviour 300
Examples 300
Example 1 300
Removal of Components: *drop* 300
Syntax 300
Input parameters 300
Examples of valid syntaxes 300
Semantics for scalar operations 300
Input parameters type 301
Result type 301
Additional Constraints 301
Behaviour 301
Examples 301
Example 1 301
Change of Component name: *rename* 301
Syntax 301
Input parameters 302
Examples of valid syntaxes 302
Semantics for scalar operations 302
Input parameters type 302
Result type 302
Additional Constraints 302
Behaviour 302
Examples 302
Example 1 303
Pivoting: *pivot* 303
Syntax 303
Input parameters 303
Examples of valid syntaxes 303
Semantics for scalar operations 303
Input parameters type 303
Result type 303
Additional Constraints 304
Behaviour 304
Examples 304
Example 1 304
Unpivoting: *unpivot* 304
Syntax 304
Input parameters 305
Examples of valid syntaxes 305
Semantics for scalar operations 305
Input parameters type 305

Result type 305
Additional Constraints 305
Behaviour 305
Examples 306
Example 1 306
Subspace: *sub* 306
Syntax 306
Input parameters 306
Examples of valid syntaxes 306
Semantics for scalar operations 306
Input parameters type 306
Result type 307
Additional Constraints 307
Behaviour 307
Examples 307
Example 1 308
Example 2 308
Example 3 308

Documentation for VTL v2.1

User Manual

Foreword

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

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

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

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

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

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

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

The second step, taking also into consideration that some VTL implementation initiatives are already in place, is aimed at acknowledging and fixing other features considered of minor impact and priority, without affecting the features already published or the possible relevant implementations. In parallel with the work for designing the new VTL version, the task force has been involved in the SDMX implementation of VTL, aiming at defining formats for exchanging rules and developing web services to retrieve them; the new features have been included in the SDMX 3.0 package.

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

a. The User Manual, highlighting the main characteristics of VTL, its core assumptions and the information model the language is based on;

b. The Reference Manual, containing the full library of operators ordered by category, including examples;

c. eBNF notation (extended Backus-Naur Form) which is the technical notation to be used as a test bed for all the examples;

d. A Technical Notes document, containing some guidelines for VTL implementation.

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

Acknowledgements

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

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

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

Introduction

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

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

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

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

Although VTL is developed under the umbrella of the SDMX governance, DDI and GSIM users may also be highly interested in adopting a language for validation and transformation. In particular, organizations involved in the SDMX, DDI and GSIM communities and in the Modernisation of Official Statistics (HLG-MOS) expressed their wish of adopting VTL as a unique language, usable in SDMX, DDI and GSIM.

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

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

Structure of the document

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

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

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

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

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

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

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

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

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

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

General characteristics of the VTL

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

User orientation

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

Integrated approach

- The language is independent of the statistical domain of the data to be processed;
    - VTL has no dependencies on the subject matter (the data content);

    - VTL is able to manipulate statistical data in relation to their structure.
- The language is suitable for the various typologies of data of a statistical environment (for example dimensional data, survey data, registers data, micro and macro, quantitative and qualitative) and is supported by an information model (IM) which covers these typologies;
    - The IM allows the representation of the various typologies of data of a statistical environment at a conceptual/logical level (in a way abstract from IT and from the physical storage);
    - The various typologies of data are described as much as possible in an integrated way, by means of common IM artefacts for their common aspects;
    - The principle of the Occam's razor is applied as an heuristic principle in designing the conceptual IM, so keeping everything as simple as possible or, in other words, unifying the model of apparently different things as much as possible.
- The language (and its IM) is independent of the phases of the statistical process and usable in any one of them;
    - Operators are designed to be independent of the phases of the process, their syntax does not change in different phases and is not bound to some characteristic restricted to a specific phase (operators' syntax is not aware of the phase of the process);
    - In principle, all operators are allowed in any phase of the process (e.g. it is possible to use the operators for data validation not only in the data collection but also, for example, in data compilation for validating the result of a compilation process; similarly it is possible to use the operators for data calculation, like the aggregation, not only in data compilation but also in data validation processes);
    - Both collected and calculated data are equally permitted as inputs of a calculation, without changes in the syntax of the operators/expression;
    - Collected and calculated data are represented (in the IM) in a homogeneous way with regard to the metadata needed for calculations.
- The language is designed to be applied not only to SDMX but also to other standards;
    - VTL, like any consistent language, relies on a specific information model, as it operates on the VTL IM artefacts to produce other VTL IM artefacts. In principle, a language cannot be applied as-is to another information model (e.g. SDMX, DDI, GSIM); this possibility exists only if there is an unambiguous correspondence between the artefacts of those information models and the VTL IM (that is if their artefacts correspond to the same mathematical notion);
    - The goal of applying the language to more models/standards is achieved by using a very simple, generic and conceptual Information Model (the VTL IM), and mapping this IM to the models of the different standards (SDMX, DDI, GSIM, …); to the extent that the mapping is straightforward and unambiguous, the language can be inherited by other standards (with the proper adjustments);
    - To achieve an unambiguous mapping, the VTL IM is deeply inspired by the GSIM IM and uses the same artefacts when possible [1]; in fact, GSIM is designed to provide a formal description of data at business level against which other information models can be mapped; a very small subset of the GSIM artefacts is used in the VTL IM in order to keep the model and the language as simple as possible (Occam's razor principle); these are the artefacts strictly needed for describing the data involved in Transformations, their structure and the variables and value domains;
    - GSIM artefacts are supplemented, when needed, with other artefacts that are necessary for describing calculations; in particular, the SDMX model for Transformations is used;
    - As mentioned above, the definition of the VTL IM artefacts is based on mathematics and is expressed at an abstract user level.

Active role for processing

- The language is designed to make it possible to drive in an active way the execution of the calculations (in addition to documenting them)
- For the purpose above, it is possible either to implement a calculation engine that interprets the VTL and operates on the data or to rely on already existing IT tools (this second option requires a translation from the VTL to the language of the IT tool to be used for the calculations)

- The VTL grammar is being described formally using the universally known Backus Naur Form notation (BNF), because this allows the VTL expressions to be easily defined and processed; the formal description allow the expressions:
    - To be parsed against the rules of the formal grammar; on the IT level, this requires the implementation of a parser that compiles the expressions and checks their correctness;
    - To be translated from the VTL to the language of the IT tool to be used for the calculation; on the IT level, this requires the implementation of a proper translator;
    - To be translated from/to other languages if needed (through the implementation of a proper translator).
- The inputs and the outputs of the calculations and the calculations themselves are artefacts of the IM
    - This is a basic property of any robust language because it allows calculated data to be operands of further calculations;
    - If the artefacts are persistently stored, their definition is persistent as well; if the artefacts are non-persistently stored (used only during the calculation process like input from other systems, intermediate results, external outputs) their definition can be non-persistent;
    - Because the definition of the algorithms of the calculations is based on the definition of their input artefacts (in particular on the data structure of the input data), the latter must be available when the calculation is defined;
    - The VTL is designed to make the data structure of the output of a calculation deducible from the calculation algorithm and from the data structure of the operands (this feature ensures that the calculated data can be defined according to the IM and can be used as operands of further calculations);
    - In the IT implementation, it is advisable to automate (as much as possible) the structural definition of the output of a calculation, in order to enforce the consistency of the definitions and avoid unnecessary overheads for the definers.
- The VTL and its information model make it possible to check automatically the overall consistency of the definitions of the calculations, including with respect to the artefact of the IM, and in particular to check:
    - the correctness of the expressions with respect to the syntax of the language
    - the integrity of the expressions with respect to their input and output artefacts and the corresponding structures and properties (for example, the input artefacts must exist, their structure components referenced in the expression must exist, qualitative data cannot be manipulated through quantitative operators, and so on)
    - the consistency of the overall graph of the calculations (for example, in order to avoid that the result of a calculation goes as input to the same calculation, there should not be cycles in the sequence of calculations, thus eliminating the risk of producing unpredictable and erroneous results).

Independence of IT implementation

- According to the “user orientation” above, the language is designed so that users are not required to be aware of the IT solution;
    - To use the language, the users need to know only the abstract view of the data and calculations and do not need to know the aspects of the IT implementation, like the storage structures, the calculation tools and so on.
- The language is not oriented to a specific IT implementation and permits many possible different implementations (this property is particularly important in order to allow different institutions to rely on different IT environments and solutions);
    - The VTL provides only for a logical/conceptual layer for defining the data transformations, which applies on a logical/conceptual layer of data definitions
    - The VTL does not prescribe any technical/physical tool or solution, so that it is possible to implement the VTL by using many different IT tools
    - The link between the logical/conceptual layer of the VTL definitions and the IT implementation layer is out of the scope of the VTL;

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

Extensibility, customizability

- The language is made of few “core” constructs, which are the fundamental building blocks into which any operation can be decomposed, and a “standard library”, which contains a number of standard operators built from the core constructs; these are the standard parts of the language, which can be extended gradually by the VTL maintenance body, enriching the available operators according to the evolution of the business needs, so progressively making the language more powerful;
- Other organizations can define additional operators having a customized behaviour and a functional syntax, so extending their own library by means of custom-designed operators. As obvious, these additional operators are not part of the standard VTL library. To exchange VTL definitions with other institutions, the possible custom libraries need to be pre-emptively shared.
- In addition, it is possible to call external routines of other languages/tools, provided that they are compatible with the IM; this requisite is aimed to fulfil specific calculation needs without modifying the operators of the language, so exploiting the power of the other languages/tools if necessary for specific purposes. In this case:
    - The external routines should be compatible with, and relate back to, the conceptual IM of the calculations as for its inputs and outputs, so that the integrity of the definitions is ensured
    - The external routines are not part of the language, so their use is subject to some limitations (e.g. it is impossible to parse them as if they were operators of the language)
    - The use of external routines compromises the IT implementation independence, the abstraction and the user orientation. Therefore external routines should be used only for specific needs and in limited cases, whereas widespread and generic needs should be fulfilled through the operators of the language;
- Whilst an Organisation adopting VTL can extend it by defining customized parts, on its own total responsibility, in order to improve the standard language for specific purposes (e.g. for supporting possible algorithms not permitted by the standard part), it is important that the customized parts remain compliant with the VTL IM and the VTL fundamentals. Adopting Organizations are totally in charge of any activity for maintaining and sharing their customized parts. Adopting Organizations are also totally in charge of any possible maintenance activity to maintain the compliance between their customized parts and the possible VTL future versions.

Language effectiveness

- The language is oriented to give full support to the various typologies of data of a statistical environment (for example dimensional data, survey data, registers data, micro and macro, quantitative and qualitative, …) described as much as possible in a coherent way, by means of common IM artefacts for their common aspects, and relying on mathematical notions, as mentioned above. The various types of statistical data are considered as mathematical functions, having independent variables (Identifiers) and dependent variables (Measures, Attributes [2]), whose extensions can be thought as logical tables (DataSets) made of rows (Data Points) and columns (Identifiers, Measures, Attributes).
- The language supports operations on the Data Sets (i.e. mathematical functions) in order to calculate new Data Sets from the existing ones, on their structure components (Identifiers, Measures, Attributes), on their Data Points.
- The algorithms are specified by means of mathematical expressions which compose the operands (Data Sets, Components …) by means of operators (e.g. +,-,*,/,>,<) to obtain a certain result (Data Sets, Components …);
- The validation is considered as a kind of calculation having as an operand the Data Sets to be validated and producing a Data Set containing information about the result of the validation;
- Calculations on multiple measures are supported by most operators, as well as calculations on the attributes of the Data Sets and calculations involving missing values;
- The operations are intended to be consistent with the real world historical changes which induce changes of the artefacts (e.g. of the code lists, of the hierarchies …); however, because different standards may represent historical changes in different ways, the implementation of this aspect is left to the standards (e.g. SDMX, DDI …), to the institutions and to the implementers adopting the VTL and therefore the VTL specifications does not prescribe any particular methodology for representing the historical changes of the artefacts (e.g. versioning, qualification of time validity);
- Almost all the VTL operators can be nested, meaning that in the invocation of an operator any operand can be the result of the invocation of other operators which calculate it;
- The results of the calculations can be permanently stored or not, according to the needs.

1 See the section “Relationships between VTL and GSIM”

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

Evolution of VTL 2.0 in respect to VTL 1.0

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

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

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

The main improvements are described below.

The Information Model

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

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

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

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

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

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

Structural artefacts and reusable rules

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

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

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

The core language and the standard library

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

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

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

The user defined operators

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

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

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

The VTL Definition Language

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

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

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

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

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

i. needed in the implementations;

ii. influenced by other implementation-specific aspects;

iii. in real applications, bound to be extended by means of other context-related metadata and adapted to the specific environment.

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

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

The functional paradigm

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

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

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

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

The operators

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

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

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

Changes for version 2.1

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

i. typos and errors in the text and/or in the examples have been fixed;

ii. new operators have been defined: time operators (datediff, dateadd, year/month/quarter/dayofmonth/dayofyear, daystoyear, daystomonth, durationtodays), case operator (simple extension of if-then-else), random operator (generating a random decimal number >= 0 and < 1)

ii. some changes have been introduced: the cast operator will have only explicit or implicit mask (no optional mask not allowed), some assumptions have been taken in the ordering for some use cases, the default window clause for analytic operators has been changed to be compliant with the SQL standard behaviour.

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

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

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

VTL Information Model

Introduction

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

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

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

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

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

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

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

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

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

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

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

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

Level 0 – the real world

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

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

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

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

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

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

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

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

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

The VTL IM is illustrated in the following sections.

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

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

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

Generic Model for Data and their structures

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

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

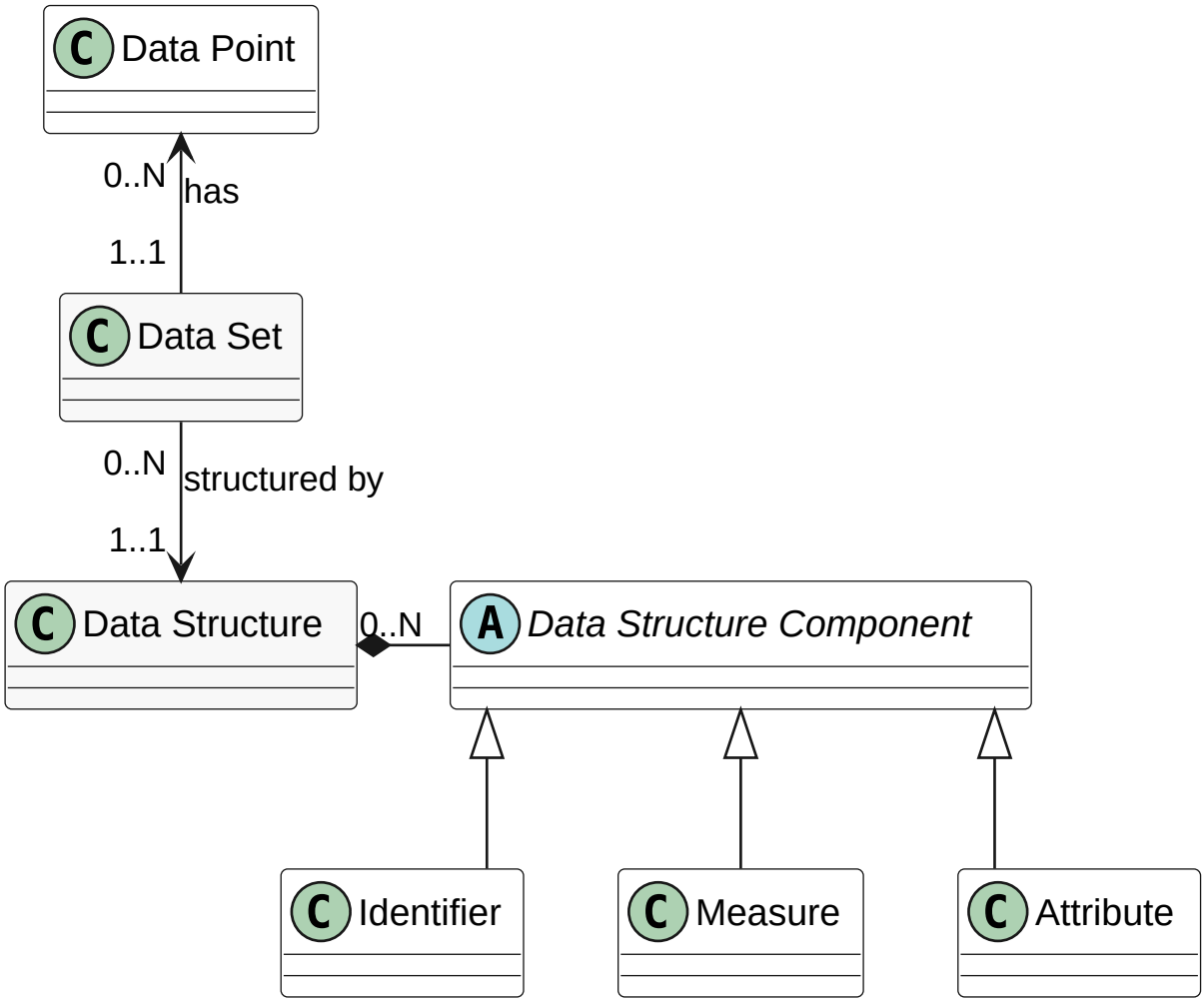
A mathematical function can be seen as a **logical table made of rows and columns**. Each column holds the values of a variable (either independent or dependent); each row holds the association between the values of the independent variables and the values of the dependent variables (in other words,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

In this way, the manipulation of any kind of data (unit and dimensional) is brought back to the manipulation of very simple and well-known objects, which can be easily understood and managed by users. According to these assumptions, there would no longer be the need of distinguishing between unit and dimensional data, and in fact VTL does not introduces such a distinction at all. Nevertheless, even if such a distinction is not part of the VTL IM, this aspect is illustrated below in this document in order to make it easier to map the VTL IM to the GSIM IM and the DDI IM, which have such a distinction.

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

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

Data model diagram



White box: same artefact as in GSIM 1.1

Light grey box: similar to GSIM 1.1

Explanation of the Diagram

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

Data Point: a single value of the function, i.e. a single association between the values of the independent variables and the values of the dependent variables. A Data Point corresponds to a row of the logical table that describes the function; therefore, the extension of the function (Data Set) is a set of Data Points. Some Data Points of the function can be unknown (i.e. missing or null), for example, the possible ones relevant to future dates. The single Data Points do not need to be individually defined, because their definition is the definition of the function (i.e. the Data Set definition).

Data Structure: the structure of a mathematical function, having independent and dependent variables. The independent variables are called “Identifier components”, the dependent variables are called either “Measure Components” or “Attribute Components”. The distinction between Measure and Attribute components is conventional and essentially based on their meaning: the Measure Components give information about the real world, while the Attribute components give information about the function itself.

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

> **Identifier Component** (or simply Identifier): a component of the data structure that is an independent variable of the function.
>
> **Measure Component** (or simply Measure): a component of the data structure that is a dependent variable of the function and gives information about the real world.
>
> **Attribute Component** (or simply Attribute): a component of the data structure that is a dependent variable of the function and gives information about the function itself. In case the automatic propagation of the Attributes is Attributes can be further classified in normal Attributes (not automatically propagated) and Viral Attributes (automatically propagated).

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

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

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

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

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

Functional Integrity

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

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

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

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

Examples

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

Production of the American Countries

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

This table is equivalent to a proper mathematical function: in fact, it fulfils the functional integrity requirements above. The Table can be defined as a Data Set, whose name can be "Production of the American Countries". Each row of the table is a Data Point belonging to the Data Set. The Data Structure of this Data Set has five Data Structure Components:

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

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

Institutional Unit Data

Row Type	I.U. ID	Ref. Date	I.U. Name	I.U. Sector	Assets	Liabilities
I	A	###	AAAAA	Private	###	###
II	A	2013	###	###	1000	800
II	A	2014	###	###	1050	750
I	B	###	BBBBB	Public	###	###

II	B	2013	###	###	1200	900
II	B	2014	###	###	1300	950
I	C	###	CCCCC	Private	###	###
II	C	2013	###	###	750	900
II	C	2014	###	###	800	850
…	…	…	…	…	…	…

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

Row Type I - Institutional Unit register

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

Row Type II - Institutional Unit Assets and Liabilities

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

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

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

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

Data Structure 1:

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

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

Data Structure 2:

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

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

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

The data artefacts

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

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

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

Data Set

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

Data Structure

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

(Data) Structure Component

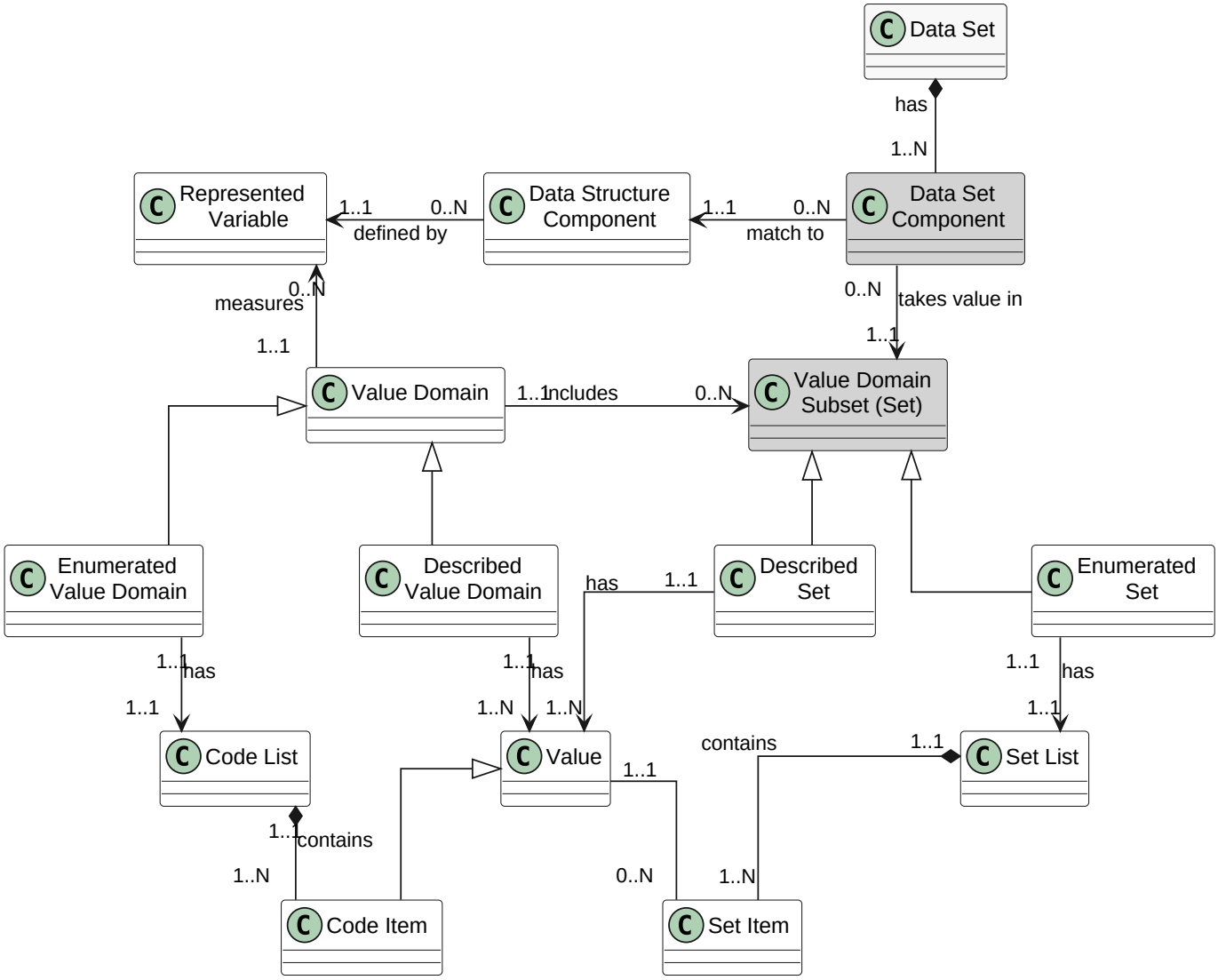
Data Structure name	*the data structure, which the Data Structure Component belongs to*
Component name	*the name of the Component*
Component Role	*IDENTIFIER or MEASURE or ATTRIBUTE (or also VIRAL ATTRIBUTE if the automatic propagation is supported)*
Represented Variable	*the Represented Variable which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

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

Generic Model for Variables and Value Domains

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

Variable and Value Domain model diagram



White box: same as in GSIM 1.1

Light grey: similar to GSIM 1.1

Dark grey additional detail (in respect to GSIM 1.1)

Explanation of the Diagram

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

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

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

information of the set of allowed values of the Component, which can be different Data Set by Data Set even if their data structure is the same.

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

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

Represented Variable: a characteristic of a statistical population (e.g. the country of birth) represented in a specific way (e.g. through the ISO code). A represented variable may take value in (or may be measured by) just one Value Domain.

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

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

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

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

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

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

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

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

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

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

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

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

Relations and operations between Code Items

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

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

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

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

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

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

UnitedKingdom < Europe means (UnitedKingdom implies Europe)

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

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

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

  Benelux = Belgium + Luxembourg + Netherlands

  This means that if a point of space belongs to Belgium OR Luxembourg OR Netherlands then it also belongs to Benelux and that if a point of space belongs to Benelux then it also belongs either to Belgium OR to Luxembourg OR to Netherlands (disjunction). In other words, the statement above says that territories of

Belgium, Netherland and Luxembourg are non-overlapping and their union is the territory of Benelux. Consequently, as for the additive measures (and being equal the other possible Identifiers), the sum of the measure values referred to Belgium, Luxembourg and Netherlands is equal to the measure value of Benelux.

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

  EUwithoutUK = EuropeanUnion - UnitedKingdom

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

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

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

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

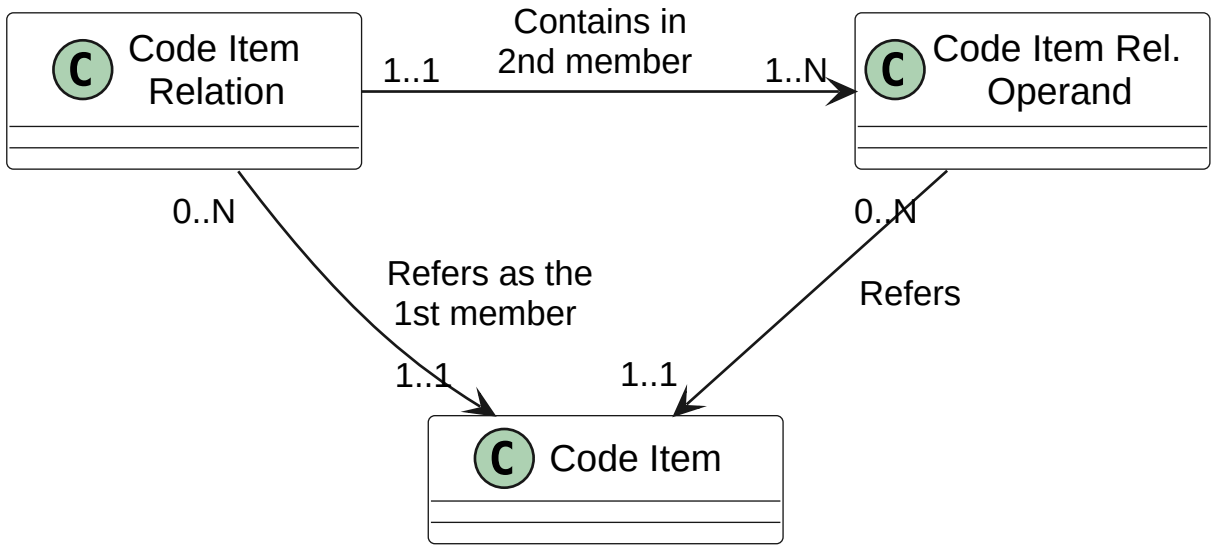
When these relations are applied to additive numeric Measures (e.g. the population relevant to geographical areas), they allow to obtain the Measure Values of the left member Code Items (i.e. the population of Benelux and EUwithoutUK) by summing or subtracting the Measure Values relevant to the component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

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

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

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

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



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

Conditioned Code Item Relations

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

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

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

The historical changes

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

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

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

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

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

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

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

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

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

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

b. Different Identifiers are assigned to the abstraction when some aspects of the abstraction change in time. For example, more Identifiers (e.g. EU1, … EU9) represent the European Union, one for each period during which its participant countries remain stable. This option is based on the conceptualization in which the abstractions are identified in connection with the time period in which they do not change, so that an Code Item (e.g. EU1) corresponds to an abstraction (e.g. the European Union) only for the time period in which the abstraction remain stable (e.g. EU1 represents the European Union from when it was created by the founder countries, to the first time it changed composition). An example of adoption of this option b) is the common practice of giving versions to Code Lists or Code Items for representing time changes (e.g. EUv_1, … , EUv_9 where v=version), being each version assumed as invariable.

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

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

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

The Variables and Value Domains artefacts

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

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

(Represented) Variable

Variable name	*name of the Represented Variable*
Value Domain name	*reference to the Value Domain that measures the Variable, i.e. in which the Variable takes values*

(Data Set) **Component**

Data Set name	*the Data set which the Component belongs to*
Component name	*the name of the Component*
(Sub) Set name	*reference to the (sub)Set containing the allowed values for the Component*

Value Domain

Value Domain name	*name of the Value Domain*
Value Domain sub-class	*if it is an Enumerated or Described Value Domain*
Basic Scalar Type	*the basic scalar type of the Values of the Value Domain, for example string, number … and so on (see also the section "VTL data types")*
Value Domain Criterion	*a criterion for restricting the Values of a basic scalar type, for example by specifying a max length of the representation, an upper or/and a lower value, and so on*

Code List *this artefact is comprised in the previous one, in fact it corresponds one to one to the enumerated Value Domain (see above)*

Value *this artefact has no explicit representation, because the Values of described Value Domains are not represented by definition, while the Values of the enumerated Value Domains are represented through the Code Item artefact (see below)*

Code Item *this artefact defines the Code Items of the Enumerated Value Domains*

Value Domain name	*the Value Domain, which the Value belongs to*
Value	*the univocal name of the Value within the Value Domain it belongs to*

(Value Domain Sub) **Set**

Value Domain name	*the Value Domain, which the Value belongs to*
Set name	*the name of the Set, which must be univocal within the Value Domain*
Set sub-class	*if it is an Enumerated or Described Set*
Set Criterion	*a criterion for identifying the Values belonging to the Set*

Set List *this artefact is comprised in the previous one, in fact it corresponds one to one to the enumerated Set*

Set Item *this artefact specifies the Code Items of the Enumerated Sets*

Value Domain name	*reference to the Value Domain which the Set and the Value belongs to*
Set name	*the Set that contains the Value*
Value	*Value element of the Set*

Code Item Relation

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

Code Item Relation Operand

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

Generic Model for Transformations

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

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

This model is essentially derived from the SDMX IM [14], as DDI and GSIM do not have an explicit transformation model at the present time [15]. In its turn, the SDMX model for Transformations is similar in scope and content to the Expression metamodel that is part of the Common Warehouse Metamodel (CWM) [16] developed by the Object Management Group (OMG).

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

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

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

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

The general form of a Transformation is the following:

result assignment_operator expression

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

Dr <- D1 (*Dr , D1* are assumed to be Data Sets)

In this Transformation the Data Set *D1* is assigned without changes (i.e. is copied) to *Dr*, which is persistently stored.

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

For example: *Dr := D1 + D2* (*Dr , D1 , D2* are assumed to be Data Sets)

In this example, the measure values of the Data Set *Dr* are calculated as the sum of the measure values of the Data Sets *D1* and *D2*, by composing the Data Points having the same Values for the Identifiers. In this case, *Dr* is not persistently stored.

A validation is intended to be a kind of Transformation. For example, the simple validation that *D1 = D2* can be made through an "If" operator, with an expression of the type:

Dr := If (*D1 = D2* , then TRUE, else FALSE)

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

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

The *expression* in the right-hand side of a Transformation must be written according to a formal language, which specifies the list of allowed operators (e.g. sum, product …), their syntax and semantics, and the rules for composing the expression (e.g. the default order of execution of the operators, the use of parenthesis to enforce a certain order …). The Operators of the language have Parameters [17], which are the a-priori unknown inputs and output of the operation, characterized by a given role (e.g. dividend, divisor or quotient in a division).

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

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

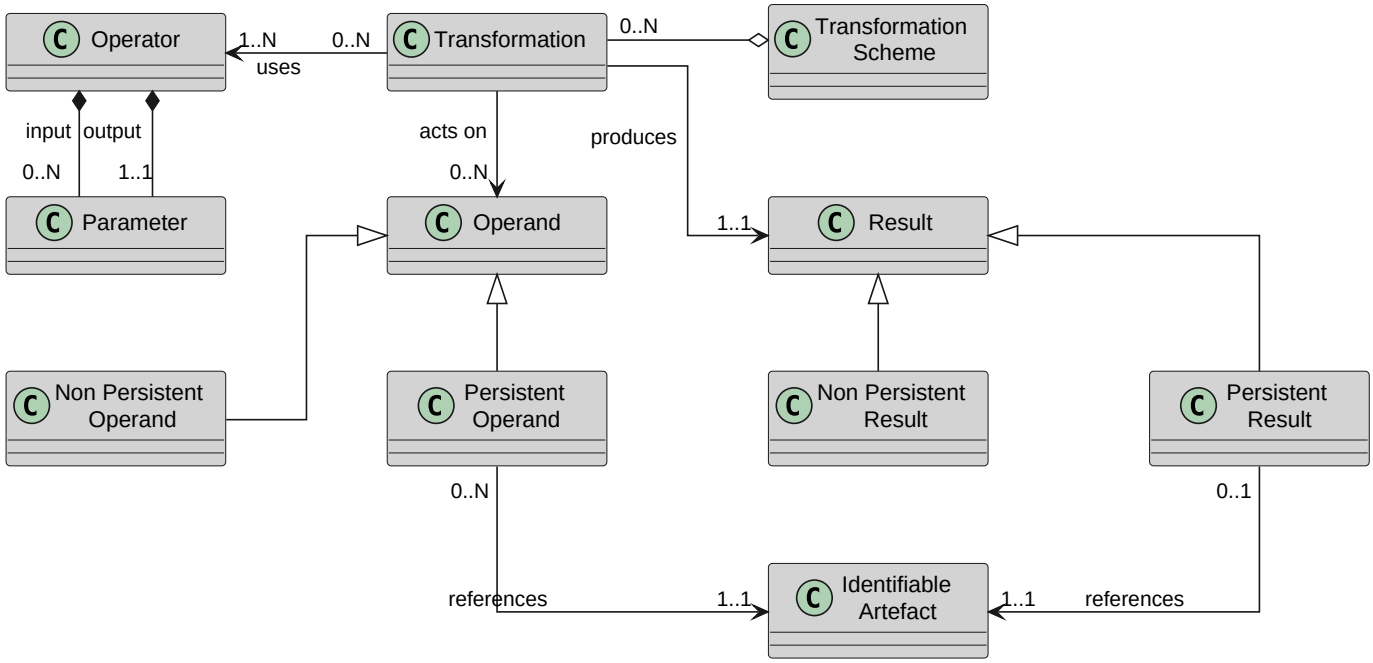
In the practical use of the language, Transformations can be composed one with another to obtain the desired outcomes. In particular, the result of a Transformation can be an operand of other Transformations, in order to define a sequence of calculations as complex as needed.

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

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

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

Transformations model diagram



White box: same as in GSIM 1.1

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

Explanation of the diagram

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

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

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

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

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

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

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

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

Examples

Imagine that *D1, D2* and *D3* are Data Sets containing information on some goods, specifically: *D1* the stocks of the previous date, *D2* the flows in the last period, *D3* the current stocks. Assume that it is desired to check the consistency of the Data Sets using the following statement:

Dr := If ((*D1* + *D2*) = *D3* , then “true”, else “false”)

In this case:

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

- *Dr* is the Result
- *D1*, *D2 and D3* are the Operands
- If ((*D1* + *D2*) = *D3* , then TRUE, else FALSE) is the Expression
- “:=”, “If”, “+” , “=” are Operators

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

- input parameters of “+”: two numeric Data Sets (to be summed)
- output parameters of “+”: a numeric Data Sets (resulting from the sum)
- input parameters of “=”: two Data Sets (to be compared)
- output parameter of “=”: a Boolean Data Set (resulting from the comparison)
- input parameters of “If”: an Expression defining a condition, i.e. (*D1*+*D2*)=*D3*
- output parameter of “If”: a Data Set (as resulting from the “then”, “else” clauses)

Functional paradigm

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

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

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

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

Transformation Consistency

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

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

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

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

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

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

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

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

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

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

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

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

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

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

7 Hyperlink “https://unece.org/statistics/modernstats/gsim”

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

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

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

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

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

13 According to the probability theory, a random experiment is a procedure that returns a result belonging a predefined set of possible results (for example, the determination of the “geographic location” may be considered as a random experiment that returns a point of the Earth surface as a result). The “space of results” is the space of all the possible results. Instead an “event” is a set of results (going back to the example of the geographic location, the event “Europe” is the set of points of the European territory and more in general an “event” corresponds to a “geographical area”). The “space of events” is the space of all the possible “events” (in the example, the space of the geographical areas).

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

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

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

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

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

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

VTL Data types

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

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

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

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

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

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

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

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

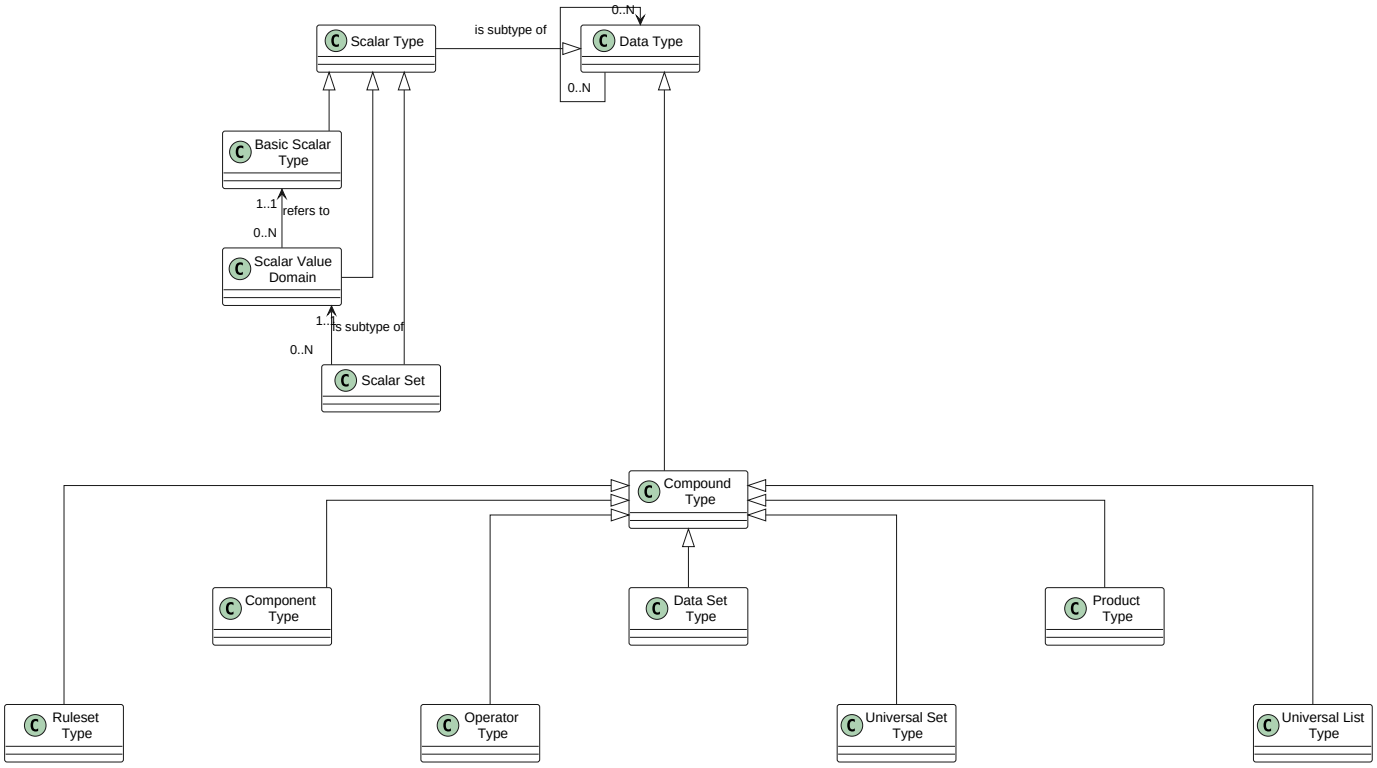
For the sake of simplicity, hereinafter the term “data type” is sometimes abbreviated to “type” and the term “scalar type” to “scalar”.

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

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

Data Types overview

Data Types model diagram



Explanation of the diagram

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

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

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

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

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

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

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

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

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

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

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

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

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

General conventions for describing the types

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

  operand :: string

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

  operand :: scalar | component | dataset

  means that "operand" can be either *scalar*, or *component*, or *dataset.*
- The angular parenthesis **< type2 >** indicates that type 2 (included in the parenthesis) restricts the specification of the preceding type, for example:

  operand :: component <string>

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

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

  operand :: component

  means "the operand is a component without other specifications" and therefore it can be of any *scalar* type, just the same as writing operand :: component<scalar> (in fact as already said, "scalar" means "any *scalar* type").
- The underscore _ indicates that the preceding type appears just one time, for example:

  measure<string> _

  indicates just one Measure having the scalar type *string*; the underscore also mean that this is a non-predetermined generic element, which therefore can be any (in the example above, the string Measure can be any).
- A specific element_name in place of the underscore denotes a predetermined element of the preceding type, for example:

  measure<string not null> my_text

  means just one Measure Component, which is a not-null *string* type and whose name is "my_text".
- The symbol _**+** means that the preceding type may appear from 1 to many times, for example:

  measure<string> _+

  means one or more generic Measures having the scalar type *string* (these Measures are not predetermined).
- The symbol _***** means that the preceding type may appear from 0 to many times, for example:

  measure<string> _*

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

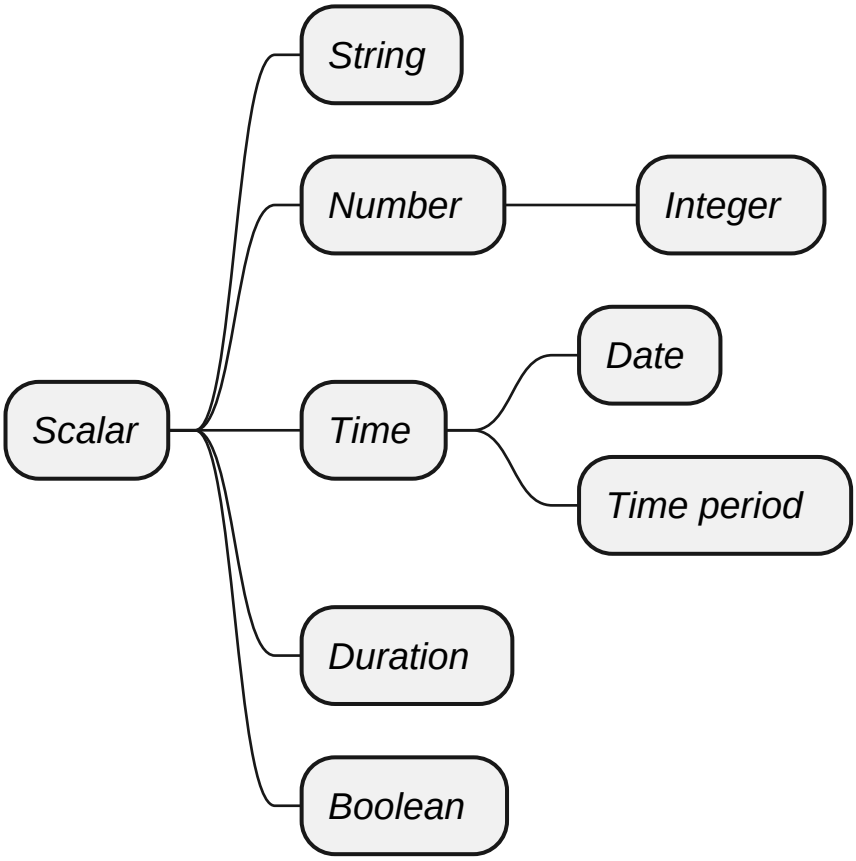
Scalar Types

Basic Scalar Types

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

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

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



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

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

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

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

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

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

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

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

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

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

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

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

Examples of possible different choices for external representations:

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

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

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

Value Domain Scalar Types

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

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

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

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

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

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

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

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

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

Set Scalar Types

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

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

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

External representations and literals used in the VTL Manuals

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

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

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

The representations adopted in these manuals are described below.

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

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

- A fixed-point *number begins* with the integer part, which is a sequence of numeric characters from 0 to 9 (at least one digit) optionally prefixed by plus or minus for the sign (no symbol means plus), a dot is always present in the end of the integer part and separates the (possible) fractional part, which is another sequence of numeric characters.

- A floating point *number*, has a mantissa written like a fixed-point number, followed by the capital letter E (for "Exponent") and by the exponent, written like a fixed-point integer;

For example:

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

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

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

For example:

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

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

YYYY-MM-DD/YYYY-MM-DD

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

2000-01-01/2000-12-31 the whole year 2000

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

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

YYYY-MM-DD

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

2000-12-31 the 31st December of the year 2000

2010-01-01 the first of January of the year 2010

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

YYYYPppp

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

2000M12 the month of December of the year 2000

2010Q1 the first quarter of the year 2010

2020Y the whole year 2010

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

Code	***Duration***
D	Day
W	Week
M	Month
Q	Quarter

S	Semester
A	Year (Annual)

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

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

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

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

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

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

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

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

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

- The string literals are written between double quotes, for example the literal “123456” is a string, even if its characters are all numeric, as well as “I am a string! “.
- The numeric literals are assumed to have some pre-definite patterns, which are the numeric patterns used for the external representation of the numbers described above.

  A literal having one of these patterns is assumed to be a number.
- The boolean literals are assumed to be the values TRUE and FALSE (capital letters without quotes).

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

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

Conventions for describing the scalar types

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

  operand :: string literal not null

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

- An **expression within square brackets** following the previous keywords, means that the preceding scalar type is restricted by the expression. This is a VTL *boolean* expression [22] (whose result can be TRUE or FALSE) which specifies a membership criterion, that is a condition that discriminates the values which belong to the restriction (sub-type) from the others (the value is assumed to belong to the sub-type only if the expression evaluates to TRUE). The keyword "value" stands for the generic value of the preceding scalar type and is used in the expression to formulate the restrictive condition. For example:

  integer [value <= 6]

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

  The following examples show some particular cases:

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

  integer [between (value, 1, 100)]

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

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

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

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

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

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

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

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

  integer [value >= 1]

  means integer equal to or greater than 1, while "integer[value < 100]" means an integer lesser than 100.

- The generic expression **[value in { v1, … , vN }]** [24] restricts a scalar type by specifying explicitly a set of possible values, for example

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

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

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

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

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

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

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

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

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

Compound Data Types

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

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

The compound Data Types are described in the following sections.

Component Types

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

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

The main sub-types of the Component Type depend on the role of the Component in the data structure and are the *identifier*, *measure* and *attribute* types (if the automatic propagation of the Attributes is supported, another sub-type is the *viral attribute*). These types reflect the fact that the VTL behaves differently on Components of different roles. Their common super- type is *component*, which means “a Component having any role”.

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

role_type < scalar_type >

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

component (or component<scalar>) any Component

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

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

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

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

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

  operand :: component

  means that “operand” may be any component.

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

  operand :: measure

  means that the operand must be a Measure.

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

- By default, a Component can be either specified directly through its name or indirectly through a sub-expression that calculates it.
- The optional keyword **name** following the type keyword means that a component name must be specified and that the component cannot be obtained through a sub-expression; For example:

operand :: measure name <string>

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

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

  operand :: component < string >

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

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

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

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

Data Set Types

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

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

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

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

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

  Some commonly used representations of the periodicity are the following:

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

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

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

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

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

dataset { type_constraint } or dataset_ts { type_constraint }

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

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

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

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

  operand:: dataset name

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

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

  operand :: dataset { measure<string> _ }

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

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

  operand:: dataset { identifier<st_Id1> Id1, …, identifier<st_IdN> IdN, measure<st_Me1> Me1, … , measure<st_MeL> MeL, attribute<st_At1> At1, … , attribute<st_AtK> AtK }

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

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

  operand:: dataset { me<st_Me1 > Me1, … , me<st_MeL > MeL }

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

Product Types

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

*string * integer * boolean*

includes elements like [30] (“PfgTj”, 7, true), (“kj-o”, 80, false), (“”, 4, false) but does not include for example (“qwe”, 2017-12-31, true), (“kj-o”, 80, 92).

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

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

i. the natural expression of exclusion or inclusion criteria (i.e., constraints) over values of two or more dataset components;

ii. the definition of the domain of the Operators in term of types of their Parameters

ii. the definition of more complex data types.

Operator Types

This is the class of the Operators’ types, i.e., the higher-levels functions that allow transformations from the type *t1* (the type of the input Parameters), to the type *t2* (the type of the output Parameter). An Operator Type is written in the form ‘*t1 -> t2*’, where *t1* and *t2* are arbitrary types. For example, the type of the following operator says that it takes as input two integer Parameters and returns a number:

Op1 :: *integer * integer -> number*

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

Ruleset Types

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

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

ruleset

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

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

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

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

datapoint on value domains {(geo_area * sector * time_period * numeric_value)}

datapoint on variables {(ref_date * import_currency * import_country)}

datapoint on value *domains* {date * _+}

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

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

{value_domain * (conditioningValueDomain1 * … * conditioningValueDomainN)}

{variable * (conditioningVariable1 * … * conditioningVariableN)}.

Examples of *hierarchical* types are:

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

hierarchical on value domains { _ * (reference_date)}

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

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

Universal Set Types

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

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

*set < string * integer * boolean >*

includes the sets [29]:

{ (“PfgTj”, 7, true), (“kj-o”, 80, false), (“”, 4, false) }

{ (“duo9”, 67, true), (“io/p”, 540, true) }

But does not includes the sets:

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

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

{ (“”, 4, false), (“F”, 8, true), (“”, 4, false) } in fact (“”, 4, false) is repeated

Universal List Types

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

For instance the following Universal List type:

list < component>

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

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

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

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

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

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

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

26 See the section “Behaviour for Attribute Components”

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

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

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

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

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

VTL Transformations

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

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

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

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

- *D1 := D2*
- *D3 := D4 + D5*

Assuming that *E* is the expression, *R* is the result and *IOi (i=1,… n)* the input Operands, the mathematical form of a Transformation based on *E* can be written as follows:

R := E (IO1, IO2, … , IOn)

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

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

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

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

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

The Expression

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

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

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

An expression is built from the following ingredients:

- **Operators**, which specify the operation to be performed (e.g. +, - and so on). As mentioned, the standard VTL operators are described in detail in the Reference Manual, moreover the VTL allows defining and then invoking "user defined operators" (see the Reference Manual, the VTL-DL for the definition and the VTL-ML for the invocation). Each operator envisages a certain number of input parameters of definite data types and produces an outcome having a definite data type (the types parameter are described in detail in the Reference Manual for each operator).
- **Operands**, which are the actual arguments passed to the invoked Operators, for example writing *D1 + D2* the Operator "+" is invoked and the Operands *D1* and *D2* are passed to it. The Operands can be:
  - **Named artefacts**, which are VTL artefacts specified through their names. Their actual values are obtained either referring to an external persistent source (persistent artefacts) or as result of previous Transformations (non-persistent artefacts) of the same Transformation Scheme; they are identified by means of a symbolic name (e.g. in *D1 + D2* the Operands *D1* and *D2* are identified by the names *D1* and *D2*). Examples of identified artefacts are the Data Sets (like *D1* and *D2* above) and the Data Set Components (like *D1#C1, D1#C2, D1#C3*, where # means that *Cj* is a Component of the Data Set *Di*).
  - **Literals**, which are VTL artefacts whose actual values are directly written in the expression; for example, in the invocation *D1 + 7* the second operand (7) is a literal, in this case a scalar literal. Also other kind of artefacts can be written in the expressions, for example the curly brackets denote the value of a Set (for example *{1, 2, 3, 4, 5, 6}* is the set of the integers from 1 to 6) and the square brackets denote a list (for example *[7, 5, 3, 6, 3]* is a list of numbers).
- **Parenthesis**, which specify the order of evaluation of the operators; for example in the expression *D1 * (D2 + D3*) first the sum *D2 + D3* is evaluated and then their product for *D1*. In case the parenthesis are not used, the default order of evaluation (described in the Reference Manual) is applied (in the example, first the product and then the sum).

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

R := O1 + O2 / (O3 – O4 / O5)

can be calculated in the following steps:

I. *(O4 / O5)*

II *(O3 - I)*

II *(O2 / II)*

I.

IV *(O1 + III)*

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

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

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

power(base, exponent),

the invocation *power(D1 + D2 , 2)* is allowed and means that first *D1 + D2* is calculated and then the result is squared. As usual, the data type of the calculated operand must comply with the allowed data types of the corresponding Parameter (in the example above, *D1 + D2* must have a numeric data type, otherwise it cannot be squared).

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

Option 1:

Dr := power(D1 + D2 , 2)

Option 2:

D3 := D1 + D2

Dr := power(D3 , 2)

In both cases, in fact, first *D1 + D2* is evaluated and then the *power* operator is applied to obtain *Dr*.

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

The Assignment

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

<- persistent assignment

:= non-persistent assignment

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

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

For example, taking again the example of Transformation above:

Dr := power(D1 + D2 , 2)

The result *Dr* can be declared as persistent by writing:

Dr <- power(D1 + D2 , 2)

Instead, to make persistent also the intermediate result of *D1 + D2* it is necessary to split the Transformation like in the option 2 above:

D3 <- D1 + D2

Dr <- power(D3 , 2)

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

The Result

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

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

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

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

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

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

The names

The artefact names

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

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

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

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

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

The **regular names**:

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

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

The regular names are:

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

The non-regular names:

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

The non-regular names are:

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

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

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

- the keywords of the VTL-ML and VTL-DL operators and of their parameters (e.g. <, := , # , inner_join, as, using, filter, apply, rename, to, + , - , power, and, or, not, group by, group except, group all, having …)

- the names of the classes of VTL artefacts of the VTL-IM (e.g., value, value domain, value domain subset, set, variable, component, data set, data structure, operator, operand parameter, transformation …)
- additional keywords for possible future use like get, put, join, map, mapping, merge, transcode and the names of commonly used mathematical and statistical functions.

The environment name

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

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

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

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

*Namespace**Name*

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

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

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

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

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

The connection to the persistent storage

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

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

VTL Operators

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

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

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

Assuming that F is the function corresponding to an operator, that P_0 is its output parameter and that $P_{i\ (i=1,\dots\ n)}$ are its input parameters, the mathematical form of an operator can be written as follows:

$$P_0 = F(P_1, \dots, P_n)$$

The function F composes the Parameters P_i to obtain P_0 (as mentioned, $P_{i\ (i=1,\dots,n)}$ and P_0 must be first order functions). In the common case in which the Parameters are Data Sets, F composes the Data Points of the input Data Sets $D_{i\ (i=1,\dots,n)}$ to obtain the Data Points of the output Data Set D_0.

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

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

The categories of VTL operators

The VTL operators are classified according to the following categories.

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

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

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

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

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

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

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

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

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

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

The input parameters

The input parameters may have various goals and in particular:

- identify the model artefacts to be manipulated

- specify possible options for the operator behaviour
- specify additional scalar values required to perform the operator's behaviour.

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

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

Unary having just one input parameter

Binary having two input parameters

N-ary having more input parameters

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

The invocation of VTL operators

Operators have different invocation styles:

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

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

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

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

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

  Operand [Operator]

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

  Postfix operators are also called "clause operators" or simply "clauses".

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

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

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

  The comma (",") is the separator between the operands. Parameter binding is fully positional: in the invocation, actual parameters are passed to the Operator in the same positional order as the corresponding formal parameters in the Operator syntax. Parameters can be mandatory or optional: usually the mandatory ones are in the first positions and the optional ones in the last positions. An underscore ("_") must be used to denote that optional operand is omitted in the invocation; for example, this is a possible invocation of *Operator1(*P_1, P*:sub:*2*, P_3 *)*, where P_2, P_3 are optional and P_2 is omitted:

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

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

  Operator1 (IO_1 **)**.

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

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

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

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

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

Level of operation

The VTL Operators can operate at various levels:

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

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

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

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

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

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

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

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

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

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

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

The Operators' behaviour

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

The Join operators

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

In particular, the Join operators allow to:

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

  inner_join D_1, D_2 *using [reference_date, geo_area]*

  matches the Data Points of D*:sub:*1*, *D*:sub:`2 *which have the same values for the Identifiers *reference_date* and *geo_area*.

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

  filter D_1 *#* M_1 *> 0*

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

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

  group by [Id_1 *,* Id_2 *]*

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

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

  apply D_1 *+* D_2

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

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

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

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

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

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

maintains only the specified measures and attributes, instead the sentence

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

drops only the specified measures and attributes

- rename the specified Components, for example:

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

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

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

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

Other operators: default behaviour on Identifiers, Measures and Attributes

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

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

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

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

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

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

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

The Identifier Components and the Data Points matching

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

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

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

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

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

D_1 = United States Data

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

D_2 = European Union Data

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

The desired result of the sum is the following:

D_r = United States + European Union

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

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

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

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

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

D_1 = European Countries

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

D_2 = Europe

Ref.Date	Population
2012	480
2013	500

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

$D_r := D_1 / D_2 * 100$

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

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

D_r = European Countries / Europe * 100

Ref.Date	Country	Population
2012	U.K.	12.5
2012	Germany	16.7
2013	U.K.	12.4
2013	Germany	16.2

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

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

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

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

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

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

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

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

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

inner_join (D_1 , D_2 apply $D_1 + D_2$)

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

full_join (D_1 , D_2 apply $D_1 + D_2$)

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

The operations on the Measure Components

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

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

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

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

D_1 = Salary of Employees

Employee ID	Salary
A	1000
B	1200
C	800
D	900

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

D_r = Increased Salary of Employees

Employee ID	Salary
A	1100
B	1320
C	880
D	990

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

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

D_1 = Import, Export, Operations

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

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

D_r = 80% of Import & Export

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

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

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

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

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

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

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

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

D_r = 80% of the Import

Ref.Date	Import
2011	800
2012	1040
2013	960

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

dataset_name [keep component_name , component_name …]

dataset_name [drop component_name, component_name …]

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

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

D_r = 80% of the Import

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

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

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

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

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

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

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

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

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

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

D_1 = Births & Deaths of the United States

Ref.Date	Births	Deaths
2011	1000	1200

2012	1300	1100
2013	1200	1300

D_2 = Birth & Deaths of the European Union

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

The Transformation *Dr := D1 + D2* will produce:

D_r = Births & Deaths of United States + European Union

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

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

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

For example, given these two Data Sets:

D_1 (Residents in the United States)

Ref.Date	Residents
2011	1000
2012	1300
2013	1200

D_2 (Inhabitants of the European Union)

Ref.Date	Inhabitants
2011	1100
2012	1200
2013	1050

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

D_r := D_2 *[rename Residents to Population]* + D_2 *[rename Inhabitants to Population]*

The result will be:

D_r (Population of United States + European Union)

Ref.Date	Population
2011	2100
2012	2500
2013	1250

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

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

D_r := D_1 #*Residents* + D_2 #*Inhabitants*

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

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

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

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

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

Operators which change the basic scalar type

Some operators change the basic data type of the input Measure (e.g. from *number* to *string*, from *string* to *date*, from *number* to *boolean* …). Some examples are the *cast* operator that converts the data types, the various *comparison* operators whose output is always *boolean*, the *length* operator which returns the length of a string.

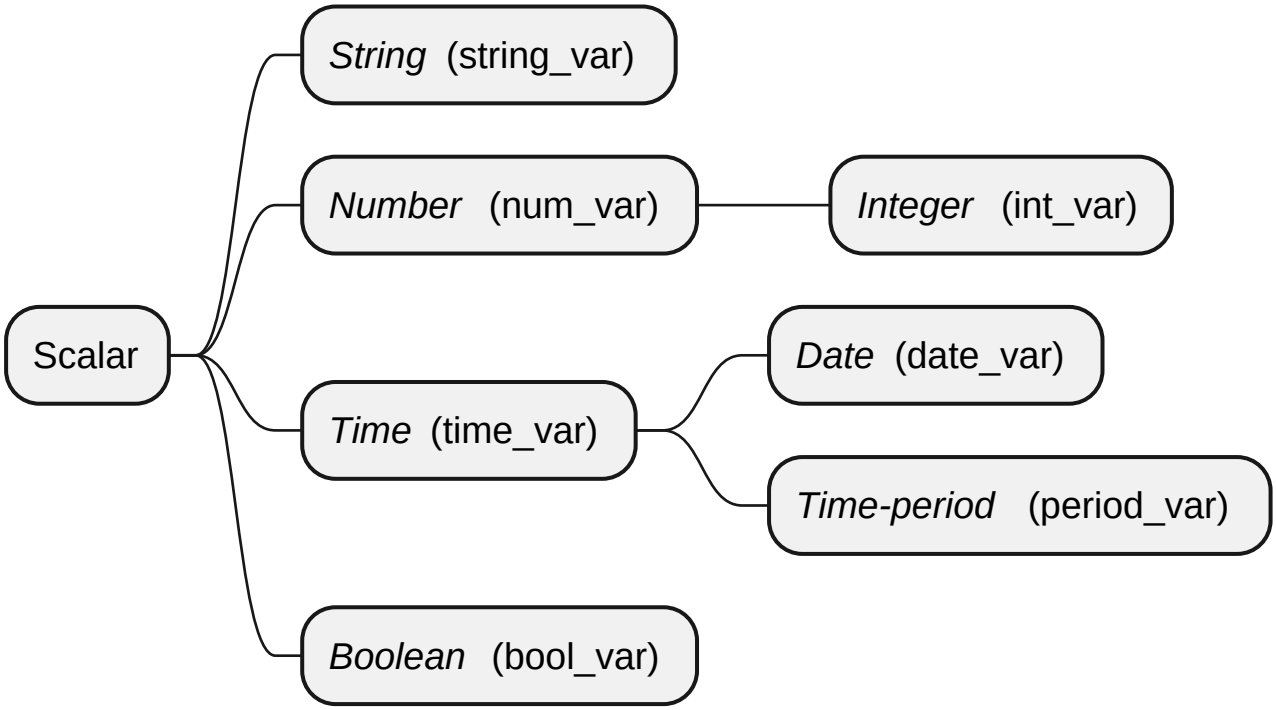
When the basic data type changes, also the Measure must change, because a Variable (in this case used with the role of Measure in a Data Structure) has just one type, which is the same wherever the Variable is used [38].

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

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

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

Basic scalar types with default measure variables



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

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

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

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

D_1 = Import_Export_Operations

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

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

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

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

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

)

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

Boolean operators

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

Set operators

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

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

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

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

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

Behaviour for Missing Data

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

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

Missing data may also show as the absence of some expected Data Point in the Data Set. For example, given a Data Set containing the reports to an international organization relevant to different countries and different dates, and having as Identifier Components the Country and the Reference Date, this Data Set may lack the Data Points relevant to some dates (for example the future dates) or some countries (for example the countries that didn’t send their data) or some combination of dates and countries.

The absence of Data Points, however, does not necessarily denote that the phenomenon under measure is unknown. In some cases, in fact, it means that a certain phenomenon did not happen.

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

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

The general properties of the NULL are the following ones:

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

- **Arithmetic operations**. Whenever a NULL value is involved in a mathematical operation (+, -, *, /, …), the result is NULL.
- **String operations**. In operations on Strings, NULL is considered an empty String (“”).
- **Boolean operations**. VTL adopts 3VL (three-valued logic). Therefore the following deduction rules are applied:

  TRUE *or* NULL → TRUE

  FALSE *or* NULL → NULL

  TRUE *and* NULL → NULL

  FALSE *and* NULL → FALSE

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

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

For example, suppose that in a specific case the NULL values of the Measure Component *M1* of the Data Set *D1* have to be considered equivalent to the number 1, the following Transformation can be used to multiply the Data Sets *D1* and *D2*, preliminarily converting NULL values of *D1.M1* into the number 1. For detailed explanations of *calc* and *is null* refer to the specific sections in the Reference Manual.

$D_r := D_1 [M_1 := \text{if } M_1 \text{ is null then 1 else } M_1] * D_2$

Behaviour for Attribute Components

Given an invocation of one Operator F, which can be written as $D_r := F(D_1 , D_2 , \dots , D_n)$, and considering that the input Data Sets D_i *(i=1,… n)* may have any number of Attribute Components, there can be the need of calculating the desired Attribute Components of D_r. This Section describes the general VTL assumptions about how Attributes are handled (the specific behaviours of the various operators are described in the Reference Manual).

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

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

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

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

The Attribute propagation rule

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

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

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

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

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

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

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

D_1 = Salary of Employees

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

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

D_r = Increased Salary of Employees

Employee ID	Salary	Currency	Scale
A	1100	U.S. $	Unit

B	1320	€	Unit
C	880	yen	Thousands
D	990	U.K. Pound	Unit

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

Another example can be given for operations involving more input Data Sets (e.g. $D_r := D_1 + D_2$). Let us assume that D_1 and D_2 contain the births and the deaths of the United States and the Europe respectively, plus a viral Attribute that qualifies if the Value is estimated or not (having values *True* or *False*).

D_1 = Births & Deaths of the United States

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

D_2 = Births & Deaths of the European Union

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

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

- each value of the Attribute is associated to a default weight;
- the result of the combination is the value having the highest weight;
- if multiple values have the same weight, the result of the combination is the first in lexicographical order.

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

D_r = Births & Deaths of United States + European Union

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

Note also that:

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

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

For example, recalling the example already given example:

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

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

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

Properties of the Attribute propagation algorithm

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

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

Commutative law (1)

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

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

Associative law (2)

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

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

Reflexive law (3)

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

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

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

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

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

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

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

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

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

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

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

Governance, other requirements and future work

The SDMX Technical Working Group, as mandated by the SDMX Secretariat, is responsible for ensuring the technical maintenance of the Validation and Transformation Language through a dedicated VTL task force. The VTL task force is open to the participation of experts from other standardisation communities, such as DDI and GSIM, as the language is designed to be usable within different standards.

The governance of the extensions

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

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

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

In order to fulfil specific calculation features not yet supported, the VTL provides for an operator which allows to define new custom operators by means of the existing ones and another operator (Evaluate) whose purpose is to invoke an external calculation function (routine), provided that this is compatible with the VTL IM, basic principles and data types.

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

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

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

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

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

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

Relations with the GSIM Information Model

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

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

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

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

Data Sets and Data Structures

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

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

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

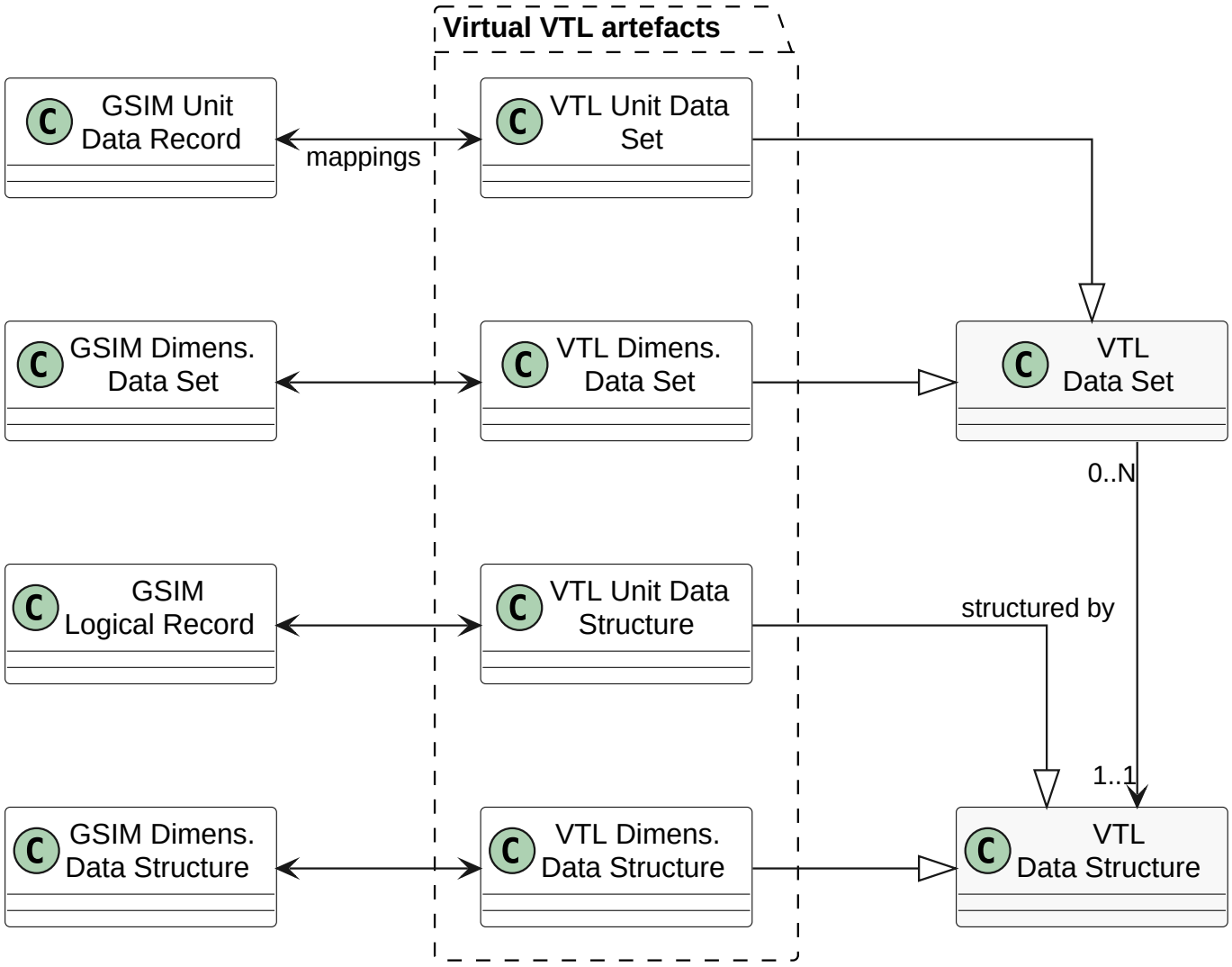
A similar relationship exists between VTL and GSIM Data Structures. In particular, introducing in VTL the virtual distinction between Unit and Dimensional Data Structures, while a GSIM Data Structure may be a GSIM Dimensional Data Structure or a GSIM Unit Data Structure, a VTL Data Structure may (virtually)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

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

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

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

GSIM – VTL mapping diagram about data structures:



Value Domains

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

Transformation model and Business Process Model

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

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

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

Annex 1 – EBNF

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

EBNF is a standard [40] meta-syntax notation, typically used to describe a Context-Free grammar and represents an extension to BNF (Backus-Naur Form) syntax. Indeed, any language described with BNF notation can also be expressed in EBNF (although expressions are typically lengthier).

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

More details can be found at http://en.wikipedia.org/wiki/Extended_Backus–Naur_Form

Properties of VTL grammar

VTL can be described in terms of a Context-Free grammar [41], with productions of the form $V \blacksquare w$, where V is a single non-terminal symbol and w is a string of terminal and non-terminal symbols.

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

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

These devices are reasonably good to guarantee the absence of ambiguity in VTL grammar. Indeed, real parser generators (for instance YACC [42]), can effectively exploit them, in particular using the mentioned associativity and precedence constrains as well as the relative ordering of the productions in the grammar itself, which solves ambiguity by default.

40	ISO/IEC 14977
41	http://en.wikipedia.org/wiki/Context-free_grammar
42	http://en.wikipedia.org/wiki/Yacc

PDF Version

Reference Manual

Foreword

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

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

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

An important point that emerged is that several standards are interested in such a kind of language. However, each standard operates on its model artefacts and produces artefacts within the same model (property of closure). To cope with this, VTL has been built upon a very basic information model (VTL IM), taking the common parts of GSIM, SDMX and DDI, mainly using artefacts from GSIM, somewhat simplified and with some additional detail. In this way, existing standards (GSIM, SDMX, DDI, others) would be allowed to adopt VTL by mapping their information model

against the VTL IM. Therefore, although a work-product of SDMX, the VTL language in itself is independent of SDMX and will be usable with other standards as well. Thanks to the possibility of being mapped with the basic part of the IM of other standards, the VTL IM also makes it possible to collect and manage the basic definitions of data represented in different standards.

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

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

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

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

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

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

a. The User Manual, highlighting the main characteristics of VTL, its core assumptions and the information model the language is based on;

b. The Reference Manual, containing the full library of operators ordered by category, including examples;

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

d. A Technical Notes document, containing some guidelines for VTL implementation.

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

Acknowledgements

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

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

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

Introduction

This document is the Reference Manual of the Validation and Transformation Language (also known as ‘VTL’) version 2.1.

The VTL 2.1 library of the Operators is described hereinafter.

VTL 2.1 consists of two parts: the VTL Definition Language (VTL-DL) and the VTL Manipulation Language (VTL-ML).

This manual describes the operators of VTL 2.1 in detail (both VTL-DL and VTL-ML) and is organized as follows.

First, in the following Chapter “Overview of the language and conventions”, the general principles of VTL are summarized, the main conventions used in this manual are presented and the operators of the VTL-DL and VTL-ML are listed. For the operators of the VTL-ML, a table that summarizes the “Evaluation Order” (i.e., the precedence rules for the evaluation of the VTL-ML operators) is also given.

The following two Chapters illustrate the operators of VTL-DL, specifically for:

- the definition of rulesets and their rules, which can be invoked with appropriate VTL-ML operators (e.g. to check the compatibility of Data Point values …);
- the definition of custom operators/functions of the VTL-ML, meant to enrich the capabilities of the VTL-ML standard library of operators.

The illustration of VTL-ML begins with the explanation of the common behaviour of some classes of relevant VTL-ML operators, towards a good understanding of general language characteristics, which we factor out and do not repeat for each operator, for the sake of compactness.

The remainder of the document illustrates each single operator of the VTL-ML and is structured in chapters, one for each category of Operators (e.g., general purpose, string, numeric …). For each Operator, there is a specific section illustrating the syntax, the semantics and giving some examples.

Overwiew of the language and conventions

Introduction

The Validation and Transformation Language is aimed at defining Transformations of the artefacts of the VTL Information Model, as more extensively explained in the User Manual.

A Transformation consists of a statement which assigns the outcome of the evaluation of an expression to an Artefact of the IM. The operands of the expression are IM Artefacts as well. A Transformation is made of the following components:

- A left-hand side, which specifies the Artefact which the outcome of the expression is assigned to (this is the result of the Transformation);
- An assignment operator, which specifies also the persistency of the left hand side. The assignment operators are two, the first one for the persistent assignment (**<-**) and the other one for the non-persistent assignment (**:=**).
- A right-hand side, which is the expression to be evaluated, whose inputs are the operands of the Transformation. An expression consists in the invocation of VTL Operators in a certain order. When an Operator is invoked, for each input Parameter, an actual argument (operand) is passed to the Operator, which returns an actual argument for the output Parameter. In the right hand side (the expression), the Operators can be nested (the output of an Operator invocation can be input of the invocation of another Operator). All the intermediate results in an expression are non-persistent.

Examples of Transformations are:

```
DS_np := (DS_1 - DS_2 ) * 2;
DS_p <- if DS_np >= 0 then DS_np else DS_1;
```

(DS_1 and DS_2 are input Data Sets, DS_np is a non persistent result, DS_p is a persistent result, the invoked operators (apart the mentioned assignments) are the subtraction (**-**), the multiplication (*****), the choice (**if**…**then**…**else**), the greater or equal comparison (**>=**) and the parentheses that control the order of the operators’ invocations.

Like in the example above, Transformations can interact with one another through their operands and results; in fact the result of a Transformation can be operand of one or more other Transformations. The interacting Transformations form a graph that is oriented and must be acyclic to ensure the overall consistency, moreover a given Artefact cannot be result of more than one Transformation (the consistency rules are better explained in the User Manual, see VTL Information Model / Generic Model for Transformations / Transformations consistency). In this regard, VTL Transformations have a strict analogy with the formulas defined in the cells of the spreadsheets.

A set of more interacting Transformations is usually needed to perform a meaningful and self-consistent task like for example the validation of one or more Data Sets. The smaller set of Transformations to be executed in the same run is called Transformation Scheme and can be considered as a VTL program.

Transformations do not necessarily need to be written in sequence like a classical software program. In fact, they are associated to the Artefacts they calculate, like it happens in the spreadsheets (each spreadsheet's formula is associated to the cell it calculates).

Nothing prevents, however, from writing the Transformations in sequence, taking into account that the Transformations are not necessarily performed in the same order as they are written, because the order of execution depends on their input-output relationships (a Transformation which calculates a result that is operand of other Transformations must be executed first). For example, if the two Transformations of the example above were written in the reverse order:

```
(i) DS_p <- if DS_np >= 0 then DS_np else DS_1;
(ii) DS_np := (DS_1 - DS_2 ) * 2;
```

All the same the Transformation (ii) would be executed first, because it calculates the Data Set DS_np which is an operand of the Transformation (i).

When Transformations are written in sequence, a semicolon (;) is used to denote the end of a Transformation and the beginning of the following one.

Conventions for writing VTL Transformations

When more Transformations are written in a text, the following conventions apply.

Transformations:

- A Transformation can be written in one or more lines, therefore the end of a line does not denote the end of a Transformation.
- The end of a Transformation is denoted by a semicolon (;).

Comments:

Comments can be inserted within VTL Transformations using the following syntaxes.

- A multi-line comment is embedded between /* and */ and, obviously, can span over several lines:

```
/\* multi-line
comment text \*/
```

- A single-line comment follows the symbol // up to the next end of line:

```
// text of a comment on a single line
```

- A sequence of spaces, tabs, end-of-line characters or comments is considered as a single space.
- The characters ***/**** , ****/*** , ***//*** and the whitespaces can be part of a string literal (within double quotes) but in such a case they are part of the string characters and do not have any special meaning.

Examples of valid comments:

Example 1:

```
\* this is a multi-line
Comment */
```

Example 2:

```
// this is single-line comment
```

Example 3:

```
DS_r <- /* A is a dataset */ A + /* B is a dataset */ B ;

(for the VTL this statement is the Transformation DS_r <- A + B ; )
```

Example 4:

```
DS_r := DS_1 // my comment
        * DS_2 ;
```

```
(for the VTL this statement is the Transformation DS_r := DS_1 * DS_2 ; )
```

Typographical conventions

The Reference Manual (this manual) uses the normal font Cambria for the text and the other following typographical conventions:

Convention	*Description*
Italics Cambria	*Basic scalar data types (in the text)* e.g. "…must have one Identifier of type *time_period*. If the Data Set…"
Bold Arial	*Keywords (in the description of the syntax and in the text)* e.g. Rule ::={ ruleName : } { **when** antecedentCondition **then** } consequentCondition { **errorcode** errorCode } { **errorlevel** errorLevel } e.g. "…The **rename** operator allows to rename one or more Components…"
Italics Arial	*Optional Parameter (in the description of the syntax)* e.g. **substr (** op**,** *start***,** *length* **)**
Underlined Arial	*Sub-expressions*
Normal font Arial	• The operator's syntax (excluded the keywords, the optional Parameters and the sub-expressions) e.g. **length (** "Hello, World!" **)** • The examples of invocation of the operators e.g. length ("Hello, World!") • Optional and Mandatory Parameters (in the text) e.g. "…If comp is a Measure in op, then in the result …"

Abbreviations for the names of the artefacts

The names of the artefacts operated by the VTL-ML come from the VTL IM. In their turn, the names of the VTL IM artefacts are derived as much as possible from the names of the GSIM IM artefacts, as explained in the User Manual.

If the complete names are long, the VTL IM suggests also a compact name, which can be used in place of the complete name in case there is no ambiguity (for example, "Set" instead than "Value Domain Subset", "Component" instead than "Data Set Component" and so on); moreover, to make the descriptions more compact, a number of abbreviations, usually composed of the initials (in capital case) or the first letters of the words of artefact names, are adopted in this manual:

Complete name	Compact name	Abbreviation
Data Set	*Data Set*	*DS*
Data Point	*Data Point*	*DP*
Identifier Component	*Identifier*	*Id*
Measure Component	*Measure*	*Me*
Attribute Component	*Attribute*	*At*
Data Set Component	*Component*	*Comp*
Value Domain Subset	*Subset or Set*	*Set*
Value Domain	*Domain*	*VD*

A positive integer suffix (with or without an underscore) can be added in the end to distinguish more than one instance of the same artefact (e.g., DS_1, DS_2, …, DS_N, Me1, Me2, …MeN). The suffix “r” stands for the result of a Transformation (e.g., DS_r).

Conventions for describing the operators’ syntax

Each VTL operator has an explanatory name, which recalls the operator function (e.g., “Greater than”) and a syntactical symbol, which is used to invoke the operator (e.g., “**>**”). The operator symbol may also be alphabetic, always lowercase (e.g., **round**).

In the VTL-DL, the operator symbol is the keyword **define** followed by the name of the object to be defined. The complete operator symbol is therefore a compound lowercase sentence (e.g. **define operator**).

In the VTL-ML, the operator symbol does not contain spaces and may be either a sequence of special characters (like **+**, **-**, **>=**, **<=** and so on) or a text keyword (e.g., **and**, **or**, **not**). The keyword may be compound with underscores as separators (e.g., **exists_in**).

Each operator has a syntax, which is a set of formal rules to invoke the operator correctly. In this document, the syntax of the operators is formally described by means of a meta-syntax which is not part of the VTL language, but has only presentation purposes.

The meta-syntax describes the syntax of the operators by means of *meta-expressions*, which define how the invocations of the operators must be written. The meta-expressions contain the symbol of the operator (e.g., “**join**”), the possible other keywords to denote special parameters (e.g., **using**), other symbols to be used (e.g., parentheses, commas), the named formal parameters (e.g., multiplicand and multiplier for the multiplication).

As for the typographic stile, in order to distinguish between the syntax symbols (which are used in the operator invocations) and meta-syntax symbols (used just for explanatory purposes, and not actually used in invocations), the syntax symbols are in **boldface** (i.e., the operator symbol, the special keywords, the possible parenthesis, commas an so on). The names of the generic operands (e.g., multiplicand, multiplier) are in Roman type, even if they are part of the syntax.

The meta-expression can be very simple, for example the meta-expression for the addition is:

op1 **+** op2

This means that the addition has two operands (op1, op2) and is invoked by specifying the name of the first addendum (op1), then the addition symbol (**+**) followed by the name of the second addendum (op2).

In this example, all the three parts of the meta-expression are fixed. In other cases, the meta-expression can be more complex and made of optional, alternative or repeated parts.

In the simple cases, the optional parts are denoted by using the *italic* face, for example:

substr (op **,** *start* **,** *length* **)**

The expression above implies that in the **substr** operator the start and length operands are optional. In the invocation, a non-specified optional operand is substituted by an underscore or, if it is in the end of the invocation, can be omitted. Hence the following syntaxes are all formally correct:

substr (op**,** start**,** length **)**

substr (op**,** start **)**

substr (op**, _ ,** length **)**

substr (op **)**

In more complex cases, a **regular expression style** is used to denote the parts (sub-expressions) of the meta-expression that are optional, alternative or repeated. In particular, braces denote a sub-expression; a vertical bar (or sometimes named “pipe”) within braces denotes possible alternatives; an optional trailing number, following the braces, specifies the number of possible repetitions.

- non-optional : non-optional sub-expression (text without braces)
- {optional} : optional sub-expression (zero or 1 occurrence)
- {non-optional}^{1} : non-optional sub-expression (just 1 occurrence)
- {one-or-more}+ : sub-expression repeatable from 1 to many occurrences

- {zero-or-more}* : sub-expression repeatable from 0 to many occurrences
- { part1 | part2 | part3 } : optional alternative sub-expressions (zero or 1 occurrence)
- { part1 | part2 | part3 }^{1} : alternative sub-expressions (just 1 occurrence)
- { part1 | part2 | part3 }+: alternative sub-expressions, from 1 to many occurrences
- { part1 | part2 | part3 }* : alternative sub-expressions, from 0 to many occurrences

Moreover, to improve the readability, some sub-expressions (the underlined ones) can be referenced by their names and separately defined, for example a meta-expression can take the following form:

sub-expr1-text sub-expr2-name … sub-exprN-1-name sub-exprN-text

sub-expr2-name ::= sub-expr2-text

… possible others …

sub-exprN-1-name ::= sub-exprN-1-text

In this representation of a meta-expression:

- The first line is the text of the meta-expression
- sub-expr$_1$-text, sub-expr$_N$-text are sub-expressions directly written in the meta-expression
- *sub-expr2-name*, … *sub-exprN-1-name* are identifiers of sub-expressions.
- sub-expr$_2$-text, … sub-expr$_{N-1}$-text are subexpression written separately from the meta-expression.
- The symbol **::=** means “is defined as” and denotes the assignment of a sub-expression-text to a sub-expression-name.

The following example shows the definition of the syntax of the operators for removing the leading and/or the trailing whitespaces from a string:

Meta-expression ::= { **trim** | **ltrim** | **rtrim** }^{1} **(** op **)**

The meta-expression above synthesizes that:

- **trim**, **ltrim**, **rtrim** are the operators’ symbols (reserved keywords);
- **(**, **)** are symbols of the operators syntax (reserved keywords);
- op is the only operand of the three operators;
- “{ }^{1}” and **“**|” are symbols of the meta-syntax; in particular “|” indicates that the three operators are alternative (a single invocation can contain only one of them) and “{ }^{1}” indicates that a single invocation contains just one of the shown alternatives;

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

```
ltrim ( DS_2 )
rtrim ( DS_3 )
```

In these invocations, **ltrim** and **rtrim** are the symbols of the invoked operator and DS_2 and DS_3 are the names of the specific Data Sets which are operands respectively of the former and the latter invocation.

Description of data types of operands and result

This section contains a brief legenda of the meaning of the symbols used for describing the possible types of operands and results of the VTL operators. For a complete description of the VTL data types, see the chapter “VLT Data Types” in the User Manual.

Symbol	Meaning	Example	Example meaning
parameter **::** type2	parameter is of the *type2*	param1 :: string	param1 is of type *string*
type1 **|** type2	alternative *types*	dataset | component | scalar	either *dataset* or *component* or *scalar*
type1 **<** type2 **>**	scalar *type2* restricts *type1*	measure <string>	Measure of *string* type
type1 _ (underscore)	*type1* can appear just once	measure <string> _	just one string Measure

type1 elementName	predetermined element of *type1*	measure <string> my_text	just one string Measure named “my_text”
type1 _ **+**	*type1* can appear one or more times	measure <string>_+	one or more string Measures
type1 _ *****	*type1* can appear zero, one or more times	measure <string>_*	zero, one or more string Measures
dataset **{** type_constraint **}**	*Type_constraint* restricts the *dataset* type	dataset { measure < string > _+ }	Dataset having one or more string Measures
t_1 ***** t_2 ***** … ***** t_n	Product of the types *t1* , *t2* , … , *tn*	string * integer * boolean	triple of scalar values made of a string, an integer and a boolean value
t_1 **->** t_2	Operator from t_1 to t_2	string -> number	Operator having input string and output number
ruleset **{** type_constraint **}**	*Type_constraint* restricts the *ruleset* type	hierarchical { geo_area }	hierarchical ruleset defined on geo_area
set **<** t **>**	Set of elements of type “t”	set < dataset >	set of datasets

Moreover, the word “name” in the data type description denotes the fact that the argument of the invocation can contain only the name of an artefact of such a type but not a sub-expression. For example:

comp :: name < component < string > >

Means that the argument passed for the input parameter comp can be only the name of a Component of the basic scalar type *string*. The argument passed for comp cannot be a component expression.

The word “name” added as a suffix to the parameter name means the same (for example if the parameter above is called comp_name).

VTL-ML Operators

Name	Symbol	Syntax	Description
Parentheses	( )	(op)	Override the default evaluation order of the operators
Persistent assignment	<-	re <- op	Assigns an Expression to a persistent model artefact
Non persistent assignment	:=	re := op	Assigns an Expression to a non persistent model artefact
Membership	#	ds#comp	Identifies a Component within a Data Set
User defined operator call		operator_name ({ argument {, argument }* })	Invokes a user defined operator passing the arguments
Evaluation of an external routine	eval	eval (externalRoutineName ({argument} {, argument }*), language, returns outputType)	Evaluates an external routine
Type conversion	cast	cast (op, scalarType {, mask })	converts to the specified data type

Join	inner_join, left_join, full_join, cross_join	joinOperator (ds { as alias } { , ds { as alias }}* { using usingComp } { filter filterCondition } { apply applyExpr | calc calcClause | aggr aggrClause { groupingClause } } { keep comp {, comp }* | drop comp {, comp }* } { rename compFrom to compTo { , compFrom to compTo }* }) joinOperator::= { inner_join | left_join| full_join | cross_join } [1] calcClause ::= { calcRole } calcComp := calcExpr { , { calcRole } calcComp := calcExpr }* calcRole :: { identifier | measure | attribute | viral attribute} [1] aggrClause ::= { aggrRole } aggrComp := aggrExpr { , { aggrRole } aggrComp := aggrExpr }* aggrRole ::= { measure | attribute | viral attribute } [1] groupingClause ::= { group by idList | group except idList | group all conversionExpr } [1] { having havingCondition }	Inner join, left outer join, full outer join, cross join
String concatenation	||	op1 || op2	Concatenates two strings
Whitespace removal	trim, rtrim, ltrim	{trim|ltrim|rtrim} [1] (op)	Removes trailing or/and leading whitespace from a string
Character case conversion	upper, lower	{upper | lower} [1] (op)	Converts the character case of a string in upper or lower case
Sub-string extraction	substr	substr (op, start, length)	Extracts the substring that starts in a specified position and has a specified length
String pattern replacement	replace	replace (op, pattern1, pattern2)	Replaces a specified string-pattern with another one
String pattern location	instr	instr(op, pattern, start, occurrence)	Returns the location of a specified string-pattern
String length	length	length (op)	Returns the length of a string
Unary plus	+	+ op	Replicates the operand with the sign unaltered
Unary minus	-	- op	Replicates the operand with the sign changed

Addition	+	op1 + op2	Sums two numbers
Subtraction	-	op1 - op2	Subtracts two numbers
Multiplication	*	op1 * op2	Multiplies two numbers
Division	/	op1 / op2	Divides two numbers
Modulo	mod	mod (op1, op2)	Calculates the remainder of a number divided by a certain divisor
Rounding	round	round (op, numDigit)	Rounds a number to a certain digit
Truncation	trunc	trunc (op, numDigit)	Truncates a number to a certain digit
Ceiling	ceil	ceil (op)	Returns the smallest integer which is greater or equal than a number
Floor	floor	floor (op)	Returns the greater integer which is smaller or equal than a number
Absolute value	abs	abs (op)	Calculates the absolute value of a number
Exponential	exp	exp (op)	Raises e (base of the natural logarithm) to a number
Natural logarithm	ln	ln (op)	Calculates the natural logarithm of a number
Power	power	power (base, exponent)	Raises a number to a certain exponent
Logarithm	log	log (op, num)	Calculates the logarithm of a number to a certain base
Square root	sqrt	sqrt (op)	Calculates the square root of a number
Equal to	=	left = right	Verifies if two values are equal
Not equal to	<>	left <> right	Verifies if two values are not equal
Greater than	> >=	left { > | >= } [1] right	Verifies if a first value is greater (or equal) than a second value
Less than	< <=	left { < | <= } [1] right	Verifies if a first value is less (or equal) than a second value
Between	between	between(op, from, to)	Verify if a value belongs to a range of values
Element of	in	op in collection collection ::= set | valueDomainName	Verifies if a value belongs to a set of values
Element of	not_in	op not_in collection collection ::= set | valueDomainName	Verifies if a value does not belong to a set of values

Match_characters	match_characters	match_characters (op, pattern)	Verifies if a value respects or not a pattern
Isnull	isnull	isnull (op)	Verifies if a value is NULL
Exists in	exists_in	exists_in (op1, op2, retain) retain ::= { true | false | all }	As for the common identifiers of op1 and op2, verifies if the combinations of values of op1 exist in op2.
Logical conjunction	and	op1 and op2	Calculates the logical AND
Logical disjunction	or	op1 or op2	Calculates the logical OR
Exclusive disjunction	xor	op1 xor op2	Calculates the logical XOR
Logical negation	not	not op	Calculates the logical NOT
Period indicator	period_indicator	period_indicator ({op})	Extracts the period indicator from a time_period value
Fill time series	fill_time_series	fill_time_series (op {, limitsMethod }) limitsMethod ::= single | all	Replaces each missing data point in the input Data Set
Flow to stock	flow_to_stock	flow_to_stock (op)	Transforms from a flow interpretation of a Data Set to stock
Stock to flow	stock_to_flow	stock_to_flow (op)	Transforms from stock to flow interpretation of a Data Set
Time shift	timeshift	timeshift (op, shiftNumber)	Shifts the time component of a specified range of time
Time aggregation	time_agg	time_agg (periodIndTo {, periodIndFrom } {,op }{, first | last })	Converts the time values from higher to lower frequency values
Actual time	current_date	current_date ( )	Returns the current date
Union	union	union (dsList) dsList ::= ds {, ds }*	Computes the union of N datasets
Intersection	intersect	intersect (dsList) dsList ::= ds {, ds }*	Computes the intersection of N datasets
Set difference	setdiff	setdiff (ds1, ds2)	Computes the differences of two datasets
Symmetric difference	symdiff	symdiff (ds1, ds2)	Computes the symmetric difference of two datasets

Hierarchical roll-up	hierarchy	hierarchy (op, hr { condition condComp {, condComp }* } { rule ruleComp } { mode } { input } { output }) condComp ::= component {, component }* mode ::= non_null | non_zero | partial_null | partial_zero | always_null | always_zero input ::= dataset | rule | rule_priority output ::= computed | all	Aggregates data using a hierarchical ruleset
Aggregate invocation		in a Data Set expression: aggregateOperator (firstOperand {, additionalOperand }* { groupingClause }) in a Component expression within an aggr clause: aggregateOperator (firstOperand {, additionalOperand }*) { groupingClause } aggregateOperator ::= avg | count | max | median | min | stddev_pop| stddev_samp | sum | var_pop | var_samp groupingClause ::= { group by groupingId {, groupingId}* | group except groupingId {, groupingId}* | group all conversionExpr } [1] { having havingCondition }	Set of statistical functions used to aggregate data

Analytic invocation		analyticOperator (firstOperand {, additionalOperand }* over (analyticClause)) analyticOperator ::= avg | count | max | median | min | stddev_pop| stddev_samp | sum | var_pop | var_samp | first_value | lag | last_value | lead | rank | ratio_to_report analyticClause ::= { partitionClause } { orderClause } { windowClause } partitionClause ::= partition by identifier {, identifier }* orderClause ::= order by component { asc | desc } {, component { asc | desc } }* windowClause ::= { data points | range } [1] between limitClause and limitClause limitClause ::= { num preceding | num following | current data point | unbounded preceding | unbounded following } [1]	Set of statistical functions used to aggregate data
Check datapoint	check_datapoint	check_datapoint (op, dpr { components listComp } { output output }) listComp ::= comp {, comp }* output ::= invalid | all | all_measures	Applies one datapoint ruleset on a Data Set
Check hierarchy	check_hierarchy	check_hierarchy (op, hr { condition condComp {, condComp }* } { rule ruleComp } { mode } { input } { output }) mode ::= non_null | non_zero | partial_null | partial_zero | always_null | always_zero input ::= dataset | dataset_priority output ::= invalid | all | all_measures	Applies a hierarchical ruleset to a Data Set
Check	check	check (op { errorcode errorcode } { errorlevel errorlevel } { imbalance imbalance } { output }) output ::= invalid | all	Checks if an expression verifies a condition

If then else	if… then… else…	if condition then thenOperand else elseOperand	Makes alternative calculations according to a condition
Nvl	nvl	nvl (op1, op2)	Replaces the null value with a value.
Filtering Data Points	filter	op [filter condition]	Filter data using a Boolean condition
Calculation of a Component	calc	op [calc { calcRole } calcComp := calcExpr {, { calcRole } calcComp := calcExpr }*]	Calculates the values of a Structure Component
Aggregation	aggr	op [aggr aggrClause { groupingClause }] aggrClause ::= { aggrRole } aggrComp := aggrExpr {, { aggrRole } aggrComp:= aggrExpr }* groupingClause ::= { group by groupingId {, groupingId }* | group except groupingId {, groupingId }* | group all conversionExpr } [1] { having havingCondition } aggrRole::= measure | attribute | viral attribute	Aggregates using an aggregate operator
Maintaining Components	keep	op [keep comp {, comp }*]	Keep list of components
Removal of Components	drop	op [drop comp {, comp }*]	Drop list of components
Change of Component name	rename	op [rename comp_from to comp_to {,comp_from to comp_to }*]	Rename components
Pivoting	pivot	op [pivot identifier, measure]	Transform identifier values to measures
Unpivoting	unpivot	op [unpivot identifier, measure]	Transform measures to identifier values
Subspace	sub	op [sub identifier = value {, identifier = value }*]	Remove the specified identifiers by fixing a value for them

VTL-ML - Evaluation order of the Operators

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

Evaluation order	Operator	Description	Default as sociativity rule
I	( )	Parentheses. To alter the default order.	None
II	VTL operators with functional syntax	VTL operators with functional syntax	Left-to-right
III	Clause Membership	Clause Membership	Left-to-right

IV	unary plus unary minus not	Unary minus Unary plus Logical negation	None
V	* /	Multiplication Division	Left-to-right
VI	+ - ||	Addition Subtraction String concatenation	Left-to-right
VII	> >= < <= = <> in not_in	Greater than Less than Equal-to Not-equal-to In a value list Not in a value list	Left-to-right
VIII	and	Logical AND	Left-to-right
IX	or xor	Logical OR Logical XOR	Left-to-right
X	if-then-else case	Conditional (if-then-else/case)	None

Description of VTL Operators

The structure used for the description of the VTL-DL Operators is made of the following parts:

- **Operator name**, which is also used to invoke the operator
- **Semantics:** a brief description of the purpose of the operator
- **Syntax:** the syntax of the Operator (this part follows the conventions described in the previous section “Conventions for describing the operators’ syntax”)
- **Syntax description:** detailed explanation of the meaning of the various parts of the syntax
- **Parameters:** list of the input parameters and their types
- **Constraints**: additional constraints that are not specified with the meta-syntax and need a textual explanation
- **Semantic specifications**: detailed description of the semantics of the operator
- **Examples**: examples of invocation of the operator

The structure used for the description of the VTL-ML Operators is made of the following parts:

- **Operator name**, followed by the **operator symbol** (keyword) which is used to invoke the operator
- **Syntax:** the syntax of the Operator (this part follows the conventions described in the previous section “Conventions for describing the operators’ syntax”)
- **Input parameters:** list of all input parameters and the subexpressions with their meaning and the indication if they are mandatory or optional
- **Examples of valid syntaxes:** examples of syntactically valid invocations of the Operator
- **Semantics for scalar operations**: the behaviour of the Operator on scalar operands, which is the basic behaviour of the Operator
- **Input parameters type**: the formal description of the type of the input parameters (this part follows the conventions described in the previous section “Description of the data types of operands and results”)
- **Result type:** the formal description of the type of the result (this part follows the conventions described in the previous section “Description of the data types of operands and results”)
- **Additional constraints**: additional constraints that are not specified with the meta-syntax and need a textual explanation, including both possible semantic constraints under which the operation is possible or impossible, and syntactical constraint for the invocation of the Operator

- **Behaviour**: description of the behaviour of the Operator for non-scalar operations (for example operations at Data Set or at Component level). When the Operator belongs to a class of Operators having a common behaviour, the common behaviour is described once for all in a section of the chapter “Typical behaviours of the ML Operators” and therefore this part describes only the specific aspect of the behaviour and contains a reference to the section where the common part of the behaviour is described.
- **Examples**: a series of examples of invocation and application of the operator in case of operations at Data Sets or at Component level.

VTL-DL - Rulesets

define datapoint ruleset

Semantics

The Data Point Ruleset contains Rules to be applied to each individual Data Point of a Data Set for validation purposes. These rulesets are also called “horizontal” 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 individual Data Points (rows), i.e., horizontally on the tabular representation.

Syntax

define datapoint ruleset rulesetName **(** dpRulesetSignature **) is** dpRule { **;** dpRule }* **end datapoint ruleset**

dpRulesetSignature ::= **valuedomain** listValueDomains | **variable** listVariables

listValueDomains ::= valueDomain { **as** vdAlias } { **,** valueDomain { **as** vdAlias } }*

listVariables ::= variable { **as** varAlias } { **,** variable { **as** varAlias } }*

dpRule ::= { ruleName **:** } { **when** antecedentCondition **then** } consequentCondition | { **errorcode** errorCode } | { **errorlevel** errorLevel }

Syntax description

rulesetName	the name of the Data Point Ruleset to be defined.
dpRulesetSignature	the Cartesian space of the Ruleset (signature of the Ruleset), which specifies either the Value Domains or the Represented Variables (see the information model) on which the Ruleset is defined. If **valuedomain** is specified then the Ruleset is applicable to the Data Sets having Components that take values on the specified Value Domains. If **variable** is specified then the Ruleset is applicable to Data Sets having the specified Variables as Components.
valueDomain	a Value Domain on which the Ruleset is defined.
vdAlias	an (optional) alias assigned to a Value Domain and valid only within the Ruleset, this can be used for the sake of compactness in writing the Rules. If an alias is not specified then the name of the Value Domain (parameter valueDomain) is used in the body of the rules.
variable	a Represented Variable on which the Ruleset is defined.

varAlias	an (optional) alias assigned to a Variable and valid only within the Ruleset, this can be used for the sake of compactness in writing the Rules. If an alias is not specified then the name of the Variable (parameter valueDomain) is used in the body of the Rules.
dpRule	a Data Point Rule, as defined in the following parameters.
ruleName	the name assigned to the specific Rule within the Ruleset. If the Ruleset is used for validation then the ruleName identifies the validation results of the various Rules of the Ruleset. The ruleName is optional and, if not specified, is assumed to be the progressive order number of the Rule in the Ruleset. However please note that, if ruleName is omitted, then the Rule names can change in case the Ruleset is modified, e.g., if new Rules are added or existing Rules are deleted, and therefore the users that interpret the validation results must be aware of these changes.
antecedentCondition	a *boolean* expression to be evaluated for each single Data Point of the input Data Set. It can contain Values of the Value Domains or Variables specified in the Ruleset signature and constants; all the VTL-ML component level operators are allowed. If omitted then antecedentCondition is assumed to be TRUE.
consequentCondition	a *boolean* expression to be evaluated for each single Data Point of the input Data Set when the antecedentCondition evaluates to TRUE (as mentioned, missing antecedent conditions are assumed to be TRUE). It contains Values of the Value Domains or Variables specified in the Ruleset signature and constants; all the VTL-ML component level operators are allowed. A consequent condition equal to FALSE is considered as a non-valid result.
errorCode	a literal denoting the error code associated to the rule, to be assigned to the possible non-valid results in case the Rule is used for validation. If omitted then no error code is assigned (NULL value). VTL assumes that a Value Domain errorcode_vd of error codes exists in the Information Model and contains all possible error codes: the errorCode literal must be one of the possible Values of such a Value Domain. VTL assumes also that a Variable errorcode for describing the error codes exists in the IM and is a dependent variable of the Data Sets which contain the results of the validation.
errorLevel	a literal denoting the error level (severity) associated to the rule, to be assigned to the possible non-valid results in case the Rule is used for validation. If omitted then no error level is assigned (NULL value). VTL assumes that a Value Domain errorlevel_vd of error levels exists in the Information Model and contains all possible error levels: the errorLevel literal must be one of the possible Values of such a Value Domain. VTL assumes also that a Variable errorlevel for describing the error levels exists in the IM and is a dependent variable of the Data Sets which contain the results of the validation.

Parameters

rulesetName:

`name <ruleset >`

valueDomain:

`name < valuedomain >`

vdAlias:

`name`

variable:

`name`

varAlias:

`name`

ruleName:

`name`

antecedentCondition:

`boolean`

consequentCondition:

`boolean`

errorCode:

`errorcode_vd`

errorLevel:

`errorlevel_vd`

Constraints

- antecedentCondition and consequentCondition can refer only to the Value Domains or Variables specified in the dpRulesetSignature.
- Either ruleName is specified for all the Rules of the Ruleset or for none.
- If specified, then ruleName must be unique within the Ruleset.

Semantic specification

This operator defines a persistent Data Point Ruleset named rulesetName that can be used for validation purposes.

A Data Point Ruleset is a persistent object that contains Rules to be applied to the Data Points of a Data Set [43]. The Data Point Rulesets can be invoked by the **check_datapoint** operator. The Rules are aimed at checking the combinations of values of the Data Set Components, assessing if these values fulfil the logical conditions expressed by the Rules themselves. The Rules are evaluated independently for each Data Point, returning a Boolean scalar value (i.e., TRUE for valid results and FALSE for non-valid results).

Each Rule contains an (optional) antecedentCondition *boolean* expression followed by a consequentCondition *boolean* expression and expresses a logical implication. Each Rule states that when the antecedentCondition evaluates to TRUE for a given Data Point, then the consequentCondition is expected to be TRUE as well. If this implication is fulfilled, the result is considered as valid (TRUE), otherwise as non-valid (FALSE). On the other side, if the antecedentCondition evaluates to FALSE, the consequentCondition does not applies and is not evaluated at all, and the result is considered as valid (TRUE). In case the antecedentCondition is absent then it is assumed to be always TRUE, therefore the consequentCondition is expected to evaluate to TRUE for all the Data Points. See an example below:

Rule	*Meaning*
On Value Domains: **when** flow_type **=** “CREDIT” or flow_type **=** “DEBIT” **then** numeric_value **>=** 0	When the Component of the Data Set which is defined on the Value Domain named flow_type takes the value “CREDIT” or the value “DEBIT”, then the other Component defined on the Value Domain named numeric_value is expected to have a zero or positive value.
On Variables: **when** flow **=** “CREDIT” or flow = “DEBIT” **then** obs_value **>= 0**	When the Component of the Data Set named flow has the value “CREDIT” or “DEBIT” then the Component named obs_value is expected to have a value greater than zero.

The definition of a Ruleset comprises a **signature** (dpRulesetSignature), which specifies the Value Domains or Variables on which the Ruleset is defined and a set of Rules, that are the Boolean expressions to be applied to each Data Point. The antecedentCondition and consequentCondition of the Rules can refer only to the Value Domains or Variables of the Ruleset signature.

The Value Domains or the Variables of the Ruleset signature identify the space in which the rules are defined while each Rule provides for a criterion that demarcates the Set of valid combinations of Values inside this space.

The Data Point Rulesets can be defined in terms of Value Domains in order to maximize their reusability, in fact this way a Ruleset can be applied on any Data Set which has Components which take values on the Value Domains of the Ruleset signature. The association between the Components of the Data Set and the Value Domains of the Ruleset signature is provided by the **check_datapoint** operator at the invocation of the Ruleset.

When the Ruleset is defined on Variables, their reusability is intentionally limited to the Data Sets which contains such Variables (and not to other possible Variables which take values from the same Value Domain). If at a later stage the Ruleset would need to be applied also to other Variables defined on the same Value Domain, a similar Ruleset should be defined also for the other Variable.

Rules are uniquely identified by ruleName. If omitted then ruleName is implicitly assumed to be the progressive order number of the Rule in the Ruleset. Please note however that, using this default mechanism, the Rule Name can change if the Ruleset is modified, e.g., if new Rules are added or existing Rules are deleted, and therefore the users that interpret the validation results must be aware of these changes. In addition, if the results of more than one Ruleset have to be combined in one Data Set, then the user should make the relevant rulesetNames different.

As said, each Rule is applied in a row-wise fashion to each individual Data Point of a Data Set. The references to the Value Domains defined in the antecedentCondition and consequentCondition are replaced with the values of the respective Components of the Data Point under evaluation.

Examples

```
define datapoint ruleset DPR_1 ( valuedomain flow_type A, numeric_value B ) is
   when A = “CREDIT” or A = “DEBIT” then B >= 0 errorcode “Bad value” errorlevel 10
end datapoint ruleset;

define datapoint ruleset DPR_2 ( variable flow F, obs_value O) is
  when F = “CREDIT” or F = “DEBIT” then O >= 0 errorcode “Bad value”
 end datapoint ruleset;
```

43 In order to apply the Ruleset to more Data Sets, these Data Sets must be composed together using the appropriate VTL operators in order to obtain a single Data Set.

define hierarchical ruleset

Semantics

This operator defines a persistent Hierarchical Ruleset that contains Rules to be applied to individual Components of a given Data Set 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 main purpose of the hierarchical Rules is to express some more aggregated Code Items (e.g. the continents) in terms of less aggregated ones (e.g., their countries) by using Code Item Relationships. 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). These rules can be used also for validation, 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), provided that the input Data Set contains all of them, i.e. the more and the less aggregated Code Items.

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 more detailed one (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. These relationships can be recursive, i.e., the aggregated Code Items can be in their turn component of even more aggregated ones, without limitations about the number of recursions.

As a first simple example, the following Hierarchical Ruleset named “BeneluxCountriesHierarchy” 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 rule Geo_Area ) is
   BENELUX = BELGIUM + LUXEMBOURG + NETHERLANDS
end hierarchical ruleset
```

Syntax

define hierarchical ruleset rulesetName **(** hrRulesetSignature **) is**

hrRule

{ **;** hrRule }*

end hierarchical ruleset

hrRulesetSignature ::= vdRulesetSignature | varRulesetSignature

vdRulesetSignature ::= **valuedomain** { **condition** vdConditioningSignature } **rule** ruleValueDomain

vdConditioningSignature ::= condValueDomain { **as** vdAlias } { **,** condValueDomain { **as** vdAlias } }*

varRulesetSignature ::= **variable** { **condition** varConditioningSignature} **rule** ruleVariable

varConditioningSignature ::= condVariable { **as** vdAlias } { **,** condVariable { **as** vdAlias } }*

hrRule ::= { ruleName **:**} codeItemRelation { **errorcode** errorCode } { **errorlevel** errorLevel }

codeItemRelation ::= { when leftCondition then }

leftCodeItem { **=** | **>** | **<** | **>=** | **<=**}:sup:*1*
{ **+** | **-** } rightCodeItem { **[** rightCondition **]** }
{ { **+** | **-** }[1] rightCodeItem { **[** rightCondition **]** } }*

Syntax description

rulesetName	the name of the Hierarchical Ruleset to be defined.
hrRulesetSignature	the signature of the Ruleset. It specifies the Value Domain or Variable on which the Ruleset is defined, and the Conditioning Signature.
vdRulesetSignature	the signature of a Ruleset defined on Value Domains
varRulesetSignature	the signature of a Ruleset defined on Variables
hrRule	a single hierarchical rule, as described below.
vdConditioningSignature	specifies the Value Domains on which the conditions are defined. The Ruleset is meant to be applicable to the Data Sets having Components that take values on the Value Domain on which the ruleset is defined (i.e., ruleValueDomain) and on the conditioning Value Domains (i.e., condValueDomain).
ruleValueDomain	the Value Domain on which the Ruleset is defined
condValueDomain	a conditioning Value Domain of the Ruleset
vdAlias	an (optional) alias assigned to a Value Domain and valid only within the Ruleset, this can be used for the sake of compactness in writing leftCondition and rightCondition. If an alias is not specified then the name of the Value Domain (i.e., condValueDomain) must be used.
varConditioningSignature	the signature of the (possible) conditions of the Ruleset defined on Variables. It specifies the Represented Variables (see the information model) on which these conditions are defined. The Ruleset is meant to be applicable to any Data Set having Components which are defined by the Variable on which the Ruleset is expressed (i.e., variable) and on the Conditioning Variables.
ruleVariable	the variable on which the Ruleset is defined
condVariable	a conditioning Variable of the Ruleset
varAlias	an (optional) alias assigned to a Variable and valid only within the Ruleset, this can be used for the sake of compactness in writing leftCondition and rightCondition. If an alias is not specified then the name of the Variableomain (parameter condVariable) must be used.
ruleName	the name assigned to the specific Rule within the Ruleset. If the Ruleset is used for validation then the ruleName identifies the validation results of the various Rules of the Ruleset. The ruleName is optional and, if not specified, is assumed to be the progressive order number of the Rule in the Ruleset. However please note that, if ruleName is omitted, then the Rule names can change in case the Ruleset is modified, e.g., if new Rules are added or existing Rules are deleted, and therefore the users that interpret the validation results must be aware of these changes. In addition, if the results of more than one Ruleset have to be combined in one Data Set, then the user should make the relevant rulesetNames different.

codeItemRelation	specifies a (possibly conditioned) Code Item Relation. It expresses a logical relation between Code Items belonging to the Value Domain of the hrRulesetSignature, possibly conditioned by the Values of the Value Domains or Variables of the Conditioning Signature. The relation is expressed by one of the symbols **=**, **>**, **>=**, **<**, **<=**, that in this context denote special logical relationships typical of Code Items. The first member of the relation is a single Code Item. The second member of the relationship is the composition of one or more Code Items combined using the symbols **+** or **-**, which in turn also denote special logical operators typical of Code Items. The meaning of these symbols is better explained below and in the User Manual.
errorCode	a literal denoting the error code associated to the rule, to be assigned to the possible non-valid results in case the Rule is used for validation. If omitted then no error code is assigned (NULL value). VTL assumes that a Value Domain errorcode_vd of the error codes exists in the Information Model and contains all the possible error codes: the errorCode literal must be one of the possible Values of such a Value Domain. VTL assumes also that a Variable errorcode for describing the error codes exists in the IM and is a dependent variable of the Data Sets which contain the results of the validation.
errorLevel	a literal denoting the error level (severity) associated to the rule, to be assigned to the possible non-valid results in case the Rule is used for validation. If omitted then no error level is assigned (NULL value). VTL assumes that a Value Domain errorlevel_vd of the error levels exists in the Information Model and contains all the possible error levels: the errorLevel literal must be one of the possible Values of such a Value Domain. VTL assumes also that a Variable errorlevel for describing the error levels exists in the IM and is a dependent variable of the Data Sets which contain the results of the validation.
leftCondition	a *boolean* expression which defines the pre-condition for evaluating the left member Code Item (i.e., it is evaluated only when the leftCondition is TRUE); It can contain references to the Value domains or the Variables of the conditioningSignature of the Ruleset and Constants; all the VTL-ML component level operators are allowed. The leftCondition is optional, if missing it is assumed to be TRUE and the Rule is always evaluated.
leftCodeItem	a Code Item of the Value Domain specified in the hrRulesetSignature.
rightCodeItem	a Code Item of the Value Domain specified in the hrRulesetSignature.

rightCondition	a *boolean* scalar expression which defines the condition for a right member Code Item to contribute to the evaluation of the Rule (i.e., the right member Code Item is taken into account only when the relevant rightCondition is TRUE). It can contain references to the Value Domains or Variables of the vdConditioningSignature or varConditioningSignature of the Ruleset and Constants; all the VTL-ML component level operators are allowed. The rightCondition is optional, if omitted then it is assumed to be TRUE and the right member Code Item is always taken into account.

Input parameters type

rulesetName

```
name < ruleset >
```

ruleValueDomain

```
name <valuedomain >
```

condValueDomain

```
name <valuedomain >
```

vdAlias

```
name
```

ruleVariable

```
name
```

condVariable

```
name
```

varAlias

```
name
```

ruleName

```
name
```

errorCode

```
errorcode_vd
```

errorLevel

```
errorlevel_vd
```

leftCondition

```
boolean
```

leftCodeItem

```
name
```

rightCodeItem

```
name
```

rightCondition

```
boolean
```

Constraints

- leftCondition and rightCondition can refer only to Value Domains or Variables specified in vdConditioningSignature or varConditioningSignature.
- Either the ruleName is specified for all the Rules of the Ruleset or for none.
- If specified, the ruleName must be unique within the Ruleset.

Semantic specification

This operator defines a Hierarchical Ruleset named rulesetName that can be used both for validation and calculation purposes (see **check_hierarchy** and **hierarchy**). A Hierarchical Ruleset is a set of Rules expressing logical relationships between the Values (Code Items) of a Value Domain or a Represented Variable.

Each rule contains a Code Item Relation, possibly conditioned, which expresses the **relation between Code Items** to be enforced. In the relation, the left member Code Item is put in relation to a combination of one or more right member Code Items. The kinds of relations are described below.

The left member Code Item can be optionally conditioned through a leftCondition, a *boolean* expression which defines the cases in which the Rule has to be applied (if not declared the Rule is applied ever). The participation of each right member Code Item in the Relation can be optionally conditioned through a rightCondition, a *boolean* expression which defines the cases in which the Code Item participates in the relation (if not declared the Code Item participates to the relation ever).

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

The envisaged types of relations are: “coincides” (**=**), “implies” (**<**), “implies or coincides” (**<=**), “is implied by” (**>**), “is implied by or coincides” (**>=**) [44]. For example:

UnitedKingdom < *Europe*

means that UnitedKingdom implies Europe (if a point belongs to United Kingdom it also belongs to Europe).

January2000 < *year2000*

means that January of the year 2000 implies the year 2000 (if a time instant belongs to “January 2000” it also belongs to the “year 2000”)

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

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

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

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

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

The Ruleset Signature specifies the space on which the Ruleset is defined, i.e., the ValueDomain or Variable on which the Code Item Relations are defined (the Ruleset is meant to be applicable to Data Sets having a Component which takes values on such a Value Domain or are defined by such a Variable). The optional vdConditioningSignature specifies the conditioning Value Domains (the conditions can refer only to those Value Domains), as well as the optional varConditioningSignature specifies the conditioning Variables (the conditions can refer only to those Variables).

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

Within the Hierarchical Rulesets there can be dependencies between Rules, because the inputs of some Rules can be the output of other Rules, so the former can be evaluated only after the latter. For example, the data relevant to the Continents can be calculated only after the calculation of the data relevant to the Countries. As a consequence, the order of calculation of the Rules is determined by their mutual dependencies and can be different from the order in which the Rules are written in the Ruleset. The dependencies between the Rules form a directed acyclic graph.

The Hierarchical ruleset can be used for calculations to calculate the upper levels of the hierarchy if the data relevant to the leaves (or some other intermediate level) are available in the operand Data Set of the **hierarchy** operator (for more information see also the “Hierarchy” operator). For example, having additive Measures broken by region, it would be possible to calculate these Measures broken by countries, continents and the world. Besides, having additive Measures broken by country, it would be possible to calculate the same Measures broken by continents and the world.

When a Hierarchical Ruleset is used for calculation, only the Relations expressing coincidence (**=**) are evaluated (provided that the leftCondition is TRUE, and taking into account only right-side Code Items whose rightCondition is TRUE). The result Data Set will contain the compound Code Items (the left members of those relations) calculated from the component Code Items (the right member of those Relations), which are taken from the input Data Set (for more details about the evaluation options see the **hierarchy** operator). Moreover, the clauses typical of the validation are ignored (e.g., ErrorCode, ErrorLevel).

The Hierarchical Ruleset can be also used to filter the input Data Points. In fact if some Code Items are defined equal to themselves, the relevant Data Points are brought in the result unchanged. For example, the following Ruleset will maintain in the result the Data Points of the input Data Set relevant to Belgium, Luxembourg and Netherland and will add new Data Points containing the calculated value for Benelux:

```
define hierarchical ruleset BeneluxRuleset ( valuedomain rule GeoArea) is

      Belgium = Belgium

      ; Luxembourg = Luxembourg

      ; Netherlands = Netherlands

      ; Benelux = Belgium + Luxembourg + Netherlands

end hierarchical ruleset
```

The Hierarchical Rulesets can be used for validation in case various levels of detail are contained in the Data Set to be validated (see also the **check_hierarchy** operator for more details). The Hierarchical Rulesets express the coherency Rules between the different levels of detail. Because in the validation the various Rules can be evaluated independently, their order is not significant.

If a Hierarchical Ruleset is used for validation, all the possible Relations (**=**, **>**, **>=**, **<**, **<=**) are evaluated (provided that the leftCondition is TRUE and taking into account only right-side Code Items whose rightCondition is TRUE). The Rules are evaluated independently. Both the Code Items of the left and right members of the Relations are expected to belong to and taken from the input Data Set (for more details about the evaluation options see the **check_hierarchy** operator). The Antecedent Condition is evaluated and, if TRUE, the operations specified in the right member of the Relation are performed and the result is compared to the first member, according to the specified type of Relation. The possible relations in which Code Items are defined as equal to themselves are ignored. Further details are described in the **check_hierarchy** operator.

If the data to be validated are in different Data Sets, either they can be joined in advance using the proper VTL operators or the validation can be done by comparing those Data Sets directly, without using a Hierarchical Ruleset (see also the **check** operator).

Through the right and left Conditions, the Hierarchical Rulesets allow to declare the time validity of Rules and Relations. In fact leftCondition and RightCondition can be defined in term of the time Value Domain, expressing respectively when the left member Code Item has to be evaluated (i.e., when it is considered valid) and when a right member Code Item participates in the relation.

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

Example 1 (for simplicity the time literals are written without the needed “cast” operation)

```
define hierarchical ruleset EuropeanUnionAreaCountries1
   ( valuedomain condition ReferenceTime as Time rule GeoArea ) is

     when between (Time, “1.1.1958”, “31.12.1972”)
             then EU = BE + FR + DE + IT + LU + NL

     ; when between (Time, “1.1.1973”, “31.12.1980”)
             then EU = *… same as above …* + DK + IE + GB

     ; when between (Time, “1.1.1981”, “02.10.1985”)
         then EU = *… same as above …* + GR

     ; when between (Time, “1.1.1986”, “31.12.1994”)
         then EU = *… same as above …* + ES + PT

     ; when between (Time, “1.1.1995”, “30.04.2004”)
         then EU = *… same as above …* + AT + FI + SE

     ; when between (Time, “1.5.2004”, “31.12.2006”)
         then EU = *… same as above …* +CY+CZ+EE+HU+LT+LV+MT+PL+SI+SK

     ; when between (Time, “1.1.2007”, “30.06.2013”)
         then EU = *… same as above …* + BG + RO

     ; when >= “1.7.2013”
         then EU = *… same as above …* + HR

end hierarchical ruleset
```

Example 2 (for simplicity the time literals are written without the needed “cast” operation)

```
define hierarchical ruleset EuropeanUnionAreaCountries2
   (valuedomain condition ReferenceTime as Time rule GeoArea ) is

EU = AT [ Time >= “0101.1995” ]
   + BE [ Time >= “01.01.1958” ]
   + BG [ Time >= “01.01.2007” ]
   + …
   + SE [ Time >= “01.01.1995” ]
   + SI [ Time >= “01.05.2004” ]
   + SK [ Time >= “01.05.2004” ]

end hierarchical ruleset
```

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

```
define hierarchical ruleset GeoHierarchy ( valuedomain rule Geo_Area) is

     World = Africa + America + Asia + Europe + Oceania

     ; Africa = Algeria + … + Zimbabwe
```

```
      ; America = Argentina + … + Venezuela

      ; Asia = Afghanistan + … + Yemen

      ; Europe = Albania + … + VaticanCity

      ; Oceania = Australia + … + Vanuatu

      ; Afghanistan = AF_reg_01 + … + AF_reg_N

      … … … … … …

      ; Zimbabwe = ZW_reg_01 + … + ZW_reg_M

      ; EuropeanUnion = … + … + … + …

      ; CentralAmericaCommonMarket = … + … + … + …

      ; OECD_Area = … + … + … + …

end hierarchical ruleset
```

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

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

```
define hierarchical ruleset TransportBreakdown ( variable rule BoPItem ) is

      transport_method1 : Transport = AirTransport + SeaTransport +
      LandTransport

      ; transport_method2 : Transport = PassengersTransport +
      FreightsTransport

end hierarchical ruleset
```

Multiple relations can be useful even for calculation. For example, imagine that the input Data Set contains data about resident units broken down by region and data about non-residents units broken down by country. In order to calculate a homogeneous level of aggregation (e.g., by country), a possible Ruleset is the following:

```
define hierarchical ruleset CalcCountryLevel ( valuedomain condition
   Residence rule GeoArea) is

      when Residence = “resident” then Country1 = Country1

      ; when Residence = “non-resident” then Country1 = Region11+ … +Region1M
      …

      ; when Residence = “resident” then CountryN = CountryN

      ; when Residence = “non-resident” then CountryN = Region N1+ …+ RegionNM

end hierarchical ruleset
```

In the calculation, basically, for each Rule, for all the input Data Points and provided that the conditions are TRUE, the right Code Items are changed into the corresponding left Code Item, obtaining Data Points referred only to the left Code Items. Then the outcomes of all the Rules of the Ruleset are aggregated together to obtain the Data Points of the result Data Set.

As far as each left Code Item is calculated by means of a single Rule (i.e., a single calculation method), this process cannot generate inconsistencies.

Instead if a left Code Item is calculated by means of more Rules (e.g., through more than one calculation method), there is the risk of producing erroneous results (e.g., duplicated data), because the outcome of the multiple Rules producing the same Code Item are aggregated together. Proper definition of the left or right conditions can avoid this risk, ensuring that for each input Data Point just one Rule is applied.

If the Ruleset is aimed only at validation, there is no risk of producing erroneous results because in the validation the rules are applied independently.

Examples

1) The Hierarchical Ruleset is defined on the Value Domain “sex”: Total is defined as Male + Female. No conditions are defined.

::

define hierarchical ruleset sex_hr (valuedomain rule sex) is

TOTAL = MALE + FEMALE

end hierarchical ruleset

2) BENELUX is the aggregation of the Code Items BELGIUM, LUXEMBOURG and NETHERLANDS. No conditions are defined.

```
define hierarchical ruleset BeneluxCountriesHierarchy (valuedomain rule GeoArea) is

    BENELUX = BELGIUM + LUXEMBOURG + NETHERLANDS errorcode “Bad value for
    Benelux”

end hierarchical ruleset
```

3) American economic partners. The first rule states that the value for North America should be greater than the value reported for US. This type of validation is useful when the data communicated by the data provider do not cover the whole composition of the aggregate but only some elements. No conditions are defined.

```
define hierarchical ruleset american_partners_hr (variable rule PartnerArea) is

    NORTH_AMERICA > US

    ; SOUTH_AMERICA = BR + UY + AR + CL

end hierarchical ruleset
```

4) Example of an aggregate Code Item having multiple definitions to be used for validation only. The Balance of Payments item “Transport” can be broken down by type of carrier (Air transport, Sea transport, Land transport) and by type of objects transported (Passengers and Freights) and both breakdowns must sum up to the total “Transport” figure.

```
define hierarchical ruleset validationruleset_bop (variable rule BoPItem ) is

    transport_method1 : Transport = AirTransport + SeaTransport +
    LandTransport

    ; transport_method2 : Transport = PassengersTransport +
    FreightsTransport

end hierarchical ruleset
```

44 “Coincides” means “implies and is implied”

VTL-DL – User Defined Operators

define operator

Syntax

define operator operator_name **(** { parameter { **,** parameter }* } **) | **{**returns** outputType } **is** operatorBody | **end define operator**

parameter::= parameterName parameterType { **default** parameterDefaultValue }

Syntax description

operator_name	the name of the operator
parameter	the names of parameters, their data types and defaultvalues
outputType	the data type of the artefact returned by the operator
operatorBody	the expression which defines the operation
parameterName	the name of the parameter
parameterType	the data type of the parameter
parameterDefaultValue	the default value for the parameter (optional)

Input parameters type

operator_name:

```
name
```

outputType:

```
a VTL data type (see the Data Type Syntax below)
```

operatorBody:

```
a VTL expression having the parameters (i.e., parameterName) as the operands
```

parameterName:

```
name
```

parameterType:

```
a VTL data type (see the Data Type Syntax below)
```

parameterDefaultValue:

```
a Value of the same type as the parameter
```

Constraints

- Each parameterName must be unique within the list of parameters
- parameterDefaultValue must be of the same data type as the corresponding parameter
- if outputType is specified then the type of operatorBody must be compatible with outputType
- If outputType is omitted then the type returned by the operatorBody expression is assumed
- If parameterDefaultValue is specified then the parameter is optional

Semantic specification

This operator defines a user-defined Operator by means of a VTL expression, specifying also the parameters, their data types, whether they are mandatory or optional and their (possible) default values.

Examples

Example 1:

```
define operator max1 (x integer, y integer)
   returns boolean is

      if x > y then x else y

end operator
```

Example 2:

```
define operator add (x integer default 0, y integer default 0)
   returns number is

      x+y
end operator
```

Data type syntax

The VTL data types are described in the VTL User Manual. Types are used throughout this Reference Manual as both meta-syntax and syntax.

They are used as meta-syntax in order to define the types of input and output parameters in the descriptions of VTL operators; they are used in the syntax, and thus are proper part of the VTL, in order to allow other operators to refer to specific data types. For example, when defining a custom operator (see the **define operator** above), one will need to declare the type of the input/output parameters.

The syntax of the data types is described below (as for the meaning of these definitions, see the section VTL Data Types in the User Manual). See also the section “Conventions for describing the operators’ syntax” in the chapter “Overview of the language and conventions” above.

dataType ::= scalarType | scalarSetType | componentType | datasetType | operatorType | rulesetType

scalarType ::= { basicScalarType_| valueDomainName | setName }[1] { scalarTypeConstraint } **{ { not } null }**

basicScalarType ::= **scalar** | **number** | **integer** | **string** | **boolean** | **time** | **date** | **time_period** | **duration**

scalarTypeConstraint ::= **[** valueBooleanCondition **]**| **{** scalarLiteral { , scalarLiteral }* **}**

scalarSetType ::= **set** { **<** scalarType **>** }

componentType ::= componentRole { **<** scalarType **>** }

componentRole ::= **component** | **identifier** | **measure** | **attribute** | **viral attribute**

datasetType ::= **dataset** { **{** componentConstraint { **,** componentConstraint }* **}** }

componentConstraint ::= componentType { componentName | multiplicityModifier }[1]

multiplicityModifier ::= _ { **+** | ***** }

operatorType ::= inputParameterType { * inputParameterType }* } -> outputParameterType

inputParameterType ::= scalarType | scalarSetType | componentType | datasetType | rulesetType

outputParameterType ::= scalarType | componentType | datasetType

rulesetType ::= { **ruleset** | dpRuleset | hrRuleset}[1]

dpRuleset ::= datapoint |

datapoint_on_valuedomains { (name { ***** name }* **) }** |
datapoint_on_variables { (name { ***** name }* **) }**

hrRuleset ::= hierarchical |

hierarchical_on_valuedomains { valueDomainName **{ (** condValueDomainName { * condValueDomainName **}*) } } }** |
hierarchical_on_variables { variableName **{ (** condValueDomainName { * condValueDomainName **}*) } } }**

Note that the valueBooleanCondition in scalarTypeConstraint is expressed with reference to the fictitious variable "value" (see also the User Manual, section "Conventions for describing the Scalar Types"), which represents the generic value of the scalar type, for example:

integer { 0, 1 } means an integer number whose value is 0 or 1

number [value >= 0] means a number greater or equal than 0

string { "A", "B", "C" } means a string whose value is A, B or C

string [length (value) <= 6] means a string whose length is lower or equal than 6

General examples of the syntax for defining types can be found in the User Manual, section VTL Data Types and in the declaration of the data types of the VTL operators (sub-sections "input parameters type" and "result type").

VTL-ML - Typical behaviours of the ML Operators

In this section, the common behaviours of some class of VTL-ML operators are described, both for a better understanding of the characteristics of such classes and to factor out and not repeat the explanation for each operator of the class.

Typical behaviour of most ML Operators

Unless differently specified in the Operator description, the Operators can be applied to Scalar Values, to Data Sets and to Data Set Components.

The operations on Scalar Values are primitive and are part of the core of the language. The other kind of operations can be typically be obtained by means of the scalar operations in conjunction with the Join operator, which is part of the core too.

In the operations on Data Set, the Operators are meant to be applied by default only to the values of the Measures of the input Data Sets, leaving the Identifiers unchanged. The Attributes follow by default their specific propagation rules, which are described in the User Manual.

In the operations on Components, the Operators are meant to be applied on the specified components of one input Data Set, in order to calculate a new component which becomes part of the resulting Data Set. In this case, the Attributes can be operated like the Measures.

Operators applicable on one Scalar Value or Data Set or Data Set Component

Operations on Scalar values

The operator is applied on a scalar value and returns a scalar value.

Operations on Data Sets

The operator is applied on a Data Set and returns a Data Set.

For example, using a functional style and denoting the operator with **f** (…), this can written as:

DS_r **:= f (** DS_1 **)**

The same operation, using an infix style and denoting the operator as **op**, can be also written as

DS_r **:= op** DS_1

This means that the operator is applied to the values of all the Measures of DS_1 in order to produce homonymous Measures in DS_r.

The application of the operator is allowed only if all the Measures of the operand Data Set are of a data type compatible with the operator (for example, a numeric operator is applicable only if all the Measures of the operand Data Sets are numeric). If the Measures of the operand Data Set are of different types, not all compatible with the operator to be applied, the membership or the keep clauses can be used to select only the proper Measures. No applicability constraints exist on Identifiers and Attributes, which can be any.

As for the data content, for each Data Point (DP_1) of the operand Data Set, a result Data Point (DP_r) is returned, having for the Identifiers the same values as DP_1.

For each Data Point DP_1 and for each Measure, the operator is applied on the Measure value of DP_1 and returns the corresponding Measure value of DP_r.

For each Data Point DP_1 and for each viral Attribute, the value of the Attribute propagates unchanged in DP_r.

As for the data structure, the result Data Set (DS_r) has the Identifiers and the Measures of the operand Data Set (DS_1), and has the Attributes resulting from the application of the attribute propagation rules on the Attributes of the operand Data Set (DS_r maintains the Attributes declared as “viral” in DS_1; these Attributes are considered as “viral” also in DS_r, the “non-viral” Attributes of DS_1 are not kept in DS_r).

Operations on Data Set Components

The operator is applied on a Component (COMP_1) of a Data Set (DS_1) and returns another Component (COMP_r) which alters the structure of DS_1 in order to produce the result Data Set (DS_r).

For example, using a functional style and denoting the operator with f (…), this can be written as:

DS_r := DS_1 **[calc** COMP_r **:= f (** COMP_1**)]**

The same operation, using an infix style and denoting the operator as **op**, can be written as:

DS_r := DS_1 **[calc** COMP_r **:= op** COMP_1 **]**

This means that the operator is applied on COMP_1 in order to calculate COMP_r.

- If COMP_r is a new Component which originally did not exist in DS_1, it is added to the original Components of DS_1, by default as a Measure (unless otherwise specified), in order to produce DS_r.
- If COMP_r is one of the original Measures or Attributes of DS_1, the values obtained from the application of the operator f (…) replace the DS_1 original values for such a Measure or Attribute in order to produce DS_r.
- If COMP_r is one of the original Identifiers of DS_1, the operation is not allowed, because the result can become inconsistent.

In any case, an operation on the Components of a Data Set produces a new Data Set, as in the example above.

The application of the operator is allowed only if the input Component belongs to a data type compatible with the operator (for example, a numeric operator is applicable only on numeric Components). As already said, COMP_r cannot have the same name of an Identifier of DS_1.

As for the data content, for each Data Point DP_1 of DS_1, the operator is applied on the values of COMP_1 so returning the value of COMP_r.

As for the data structure, like for the operations on Data Sets above, the result Data Set (DS_r) has the Identifiers and the Measures of the operand Data Set (DS_1), and has the Attributes resulting from the application of the attribute propagation rules on the Attributes of the operand Data Set (DS_r maintains the Attributes declared as “viral” in DS_1; these Attributes are considered as “viral” also in DS_r, the “non-viral” Attributes of DS_1 are not kept in DS_r). If an Attribute is explicitly calculated, the attribute propagation rule is overridden.

Moreover, in the case of the operations on Data Set Components, the (possible) new Component DS_r can be added to the original structure, the role of a (possible) existing DS_1 Component can be altered, the virality of a (possibly) existing DS_r Attribute can be altered, a (possible) COMP_r non-viral Attribute can be kept in the result. For the alteration of role and virality see also the **calc** clause.

Operators applicable on two Scalar Values or Data Sets or Data Set Components

Operation on Scalar values

The operator is applied on two Scalar values and returns a Scalar value.

Operation on Data Sets

The operator is applied either on two Data Sets or on one Data Set and one Scalar value and returns a Data Set. The composition of a Data Set and a Component is not allowed (it makes no sense).

For example, using a functional style and denoting the operator with **f** (…), this can be written as:

DS_r **:= f (** DS_1**,** DS_2 **)**

The same kind of operation, using an infix stile and denoting the operator as **op**, can be also written as

DS_r **:=** DS_1 **op** DS_2

This means that the operator is applied to the values of all the couples of Measures of DS_1 and DS_2 having the same names in order to produce homonymous Measures in DS_r. DS_1 or DS_2 may be replaced by a Scalar value.

The composition of two Data Sets (DS_1, DS_2) is allowed if the two operand Data Sets have exactly the same Measures and if all these Measures belong to a data type compatible with the operator (for example, a numeric operator is applicable only if all the Measures of the operand Data Sets are numeric). If the Measures of the operand Data Sets are different or of different types not all compatible with the operator to be applied, the membership or the **keep** clauses can be used to select only the proper Measures. The composition is allowed if these operand Data Sets have the same Identifiers or if one of them has at least all the Identifiers of the other one (in other words, the Identifiers of one of the Data Sets must be a superset of the Identifiers of the other one). No applicability constraints exist on the Attributes, which can be any.

As for the data content, the operand Data Sets (DS_1, DS_2) are joined to find the couples of Data Points (DP_1, DP_2), where DP_1 is from the first operand (DS_1) and DP_2 from the second operand (DS_2), which have the same values as for the common Identifiers. Data Points that are not coupled are left out (the inner join is used). An operand Scalar value is treated as a Data Point that couples with all the Data Points of the other operand. For each couple (DP_1, DP_2) a result Data Point (DP_r) is returned, having for the Identifiers the same values as DP_1 and DP_2.

For each Measure and for each couple (DP_1, DP_2), the Measure values of DP_1 and DP_2 are composed through the operator so returning the Measure value of DP_r. An operand Scalar value is composed with all the Measures of the other operand.

For each couple (DP_1, DP_2) and for each Attribute that propagates in DP_r, the Attribute value is calculated by applying the proper Attribute propagation algorithm on the values of the Attributes of DP_1 and DP_2 .

As for the data structure, the result Data Set (DS_r) has all the Identifiers (with no repetition of common Identifiers) and the Measures of both the operand Data Sets, and has the Attributes resulting from the application of the attribute propagation rules on the Attributes of the operands (DS_r maintains the Attributes declared as “viral” for the operand Data Sets; these Attributes are considered as “viral” also in DS_r, the “non-viral” Attributes of the operand Data Sets are not kept in DS_r).

Operation on Data Set Components

The operator is applied either on two Data Set Components (COMP_1, COMP_2) belonging to the same Data Set (DS_1) or on a Component and a Scalar value, and returns another Component (COMP_r) which alters the structure of DS_1 in order to produce the result Data Set (DS_r). The composition of a Data Set and a Component is not allowed (it makes no sense).

For example, using a functional style and denoting the operator with **f** (…), this can be written as:

DS_r := DS_1 **[calc** COMP_r **:= f (** COMP_1**,** COMP_2 **)]**

The same operation, using an infix style and denoting the operator as **op**, can be written as:

DS_r := DS_1 **[calc** COMP_r **:=** COMP_1 **op** COMP_2 **]**

This means that the operator is applied on COMP_1 and COMP_2 in order to calculate COMP_r.

- If COMP_r is a new Component which originally did not exist in DS_1, it is added to the original Components of DS_1, by default as a Measure (unless otherwise specified), in order to produce DS_r.
- If COMP_r is one of the original Measures or Attributes of DS_1, the values obtained from the application of the operator f (…) replace the DS_1 original values for such a Measure or Attribute in order to produce DS_r.
- If COMP_r is one of the original Identifiers of DS_1, the operation is not allowed, because the result can become inconsistent.

In any case, an operation on the Components of a Data Set produces a new Data Set, like in the example above.

The composition of two Data Set Components is allowed provided that they belong to the same Data Set [45]. Moreover, the input Components must belong to data types compatible with the operator (for example, a numeric operator is applicable only on numeric Components). As already said, COMP_r cannot have the same name of an Identifier of DS_1.

As for the data content, for each Data Point of DS_1, the values of COMP_1 and COMP_2 are composed through the operator so returning the value of COMP_r.

As for the data structure, the result Data Set (DS_r) has the Identifiers and the Measures of the operand Data Set (DS_1), and has the Attributes resulting from the application of the attribute propagation rules on the Attributes of the operand Data Set (DS_r maintains the Attributes declared as “viral” in DS_1; these Attributes are considered as “viral” also in DS_r, the “non-viral” Attributes of DS_1 are not kept in DS_r). If an Attribute is explicitly calculated, the attribute propagation rule is overridden.

Moreover, in the case of the operations on Data Set Components, a (possible) new Component DS_r can be added to the original structure of DS_1, the role of a (possibly) existing DS_1 Component can be altered, the virality of a (possibly) existing DS_r Attributes can be altered, a (possible) COMP_r non-viral Attribute can be kept in the result. For the alteration of role and virality see also the **calc** clause.

Operators applicable on more than two Scalar Values or Data Set Components

The cases in which an operator can be applied on more than two Data Sets (like the Join operators) are described in the relevant sections.

Operation on Scalar values

The operator is applied on more Scalar values and returns a Scalar value according to its semantics.

Operation on Data Set Components

The operator is applied either on a combination of more than two Data Set Components (COMP_1, COMP_2) belonging to the same Data Set (DS_1) or Scalar values, and returns another Component (COMP_r) which alters the structure of DS_1 in order to produce the result Data Set (DS_r). The composition of a Data Set and a Component is not allowed (it makes no sense).

For example, using a functional style and denoting the operator with **f** (…), this can be written as:

DS_r := DS_1 **[substr** COMP_r **:= f (** COMP_1**,** COMP_2, COMP_3 **)]**

This means that the operator is applied on COMP_1, COMP_2 and COMP_3 in order to calculate COMP_r.

- If COMP_r is a new Component which originally did not exist in DS_1, it is added to the original Components of DS_1, by default as a Measure (unless otherwise specified), in order to produce DS_r.
- If COMP_r is one of the original Measures or Attributes of DS_1, the values obtained from the application of the operator f (…) replace the DS_1 original values for such a Measure or Attribute in order to produce DS_r.
- If COMP_r is one of the original Identifiers of DS_1, the operation is not allowed, because the result can become inconsistent.

In any case, an operation on the Components of a Data Set produces a new Data Set, like in the example above.

The composition of more Data Set Components is allowed provided that they belong to the same Data Set [46]. Moreover, the input Components must belong to data types compatible with the operator (for example, a numeric operator is applicable only on numeric Components). As already said, COMP_r cannot have the same name of an Identifier of DS_1.

As for the data content, for each Data Point of DS_1, the values of COMP_1, COMP_2 and COMP_3 are composed through the operator so returning the value of COMP_r.

As for the data structure, the result Data Set (DS_r) has the Identifiers and the Measures of the operand Data Set (DS_1), and has the Attributes resulting from the application of the attribute propagation rules on the Attributes of the operand Data Set (DS_r maintains the Attributes declared as “viral” in DS_1; these Attributes are considered as “viral” also in DS_r, the “non-viral” Attributes of DS_1 are not kept in DS_r). If an Attribute is explicitly calculated, the attribute propagation rule is overridden.

Moreover, in the case of the operations on Data Set Components, a (possible) new Component DS_r can be added to the original structure of DS_1, the role of a (possibly) existing DS_1 Component can be altered, the virality of a (possibly) existing DS_r Attributes can be altered, a (possible) COMP_r non-viral Attribute can be kept in the result. For the alteration of role and virality see also the **calc** clause.

Behaviour of Boolean operators

The Boolean operators are allowed only on operand Data Sets that have a single measure of type *boolean*. As for the other aspects, the behaviour is the same as the operators applicable on one or two Data Sets described above.

Behaviour of Set operators

These operators apply the classical set operations (union, intersection, difference, symmetric differences) to the Data Sets, considering them as sets of Data Points. These operations are possible only if the Data Sets to be operated have the same data structure, and therefore the same Identifiers, Measures and Attributes [47].

Behaviour of Time operators

The *time* operators are the operators dealing with *time*, *date* and *time_period* basic scalar types. These types are described in the User Manual in the sections “Basic Scalar Types” and “External representations and literals used in the VTL Manuals”.

The time-related formats used for explaining the time operators are the following (they are described also in the User Manual).

For the *time* values:

YYYY-MM-DD/YYYY-MM-DD

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

2000-01-01/2000-12-31 the whole year 2000

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

For the *date* values:

YYYY-MM-DD

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

2000-12-31 the 31st December of the year 2000

2010-01-01 the first of January of the year 2010

For the *time_period* values:

YYYY{P}{NNN}

Where *YYYY* are 4 digits for the year, *P* is one character for the period indicator of the regular period (it refers to the *duration* data type and can assume one of the possible values listed below), *NNN* are from zero to three digits which contain the progressive number of the period in the year. For annual data the A and the three digits NNN can be omitted. For example:

2000M12 the month of December of the year 2000 (duration: M)

2010Q1 the first quarter of the year 2010 (duration: Q)

2010A the whole year 2010 (duration: A)

2010 the whole year 2010 (duration: A)

For the *duration* values, which are the possible values of the period indicator of the regular periods above, it is used for simplicity just one character whose possible values are the following:

Code Duration

D Day

W Week

M Month

Q Quarter

S Semester

A Year

As mentioned in the User Manual, these are only examples of possible time-related representations, each VTL system is free of adopting different ones. In fact no predefined representations are prescribed, VTL systems are free to using they preferred or already existing ones.

Several time operators deal with the specific case of Data Sets of time series, having an Identifier component that acts as the reference time and can be of one of the scalar types *time*, *date* or *time_period*; moreover this Identifier must be periodical, i.e. its possible values are regularly spaced and therefore have constant duration (frequency).

It is worthwhile to recall here that, in the case of Data Sets of time series, VTL assumes that the information about which is the Identifier Components that acts as the reference time and which is the period (frequency) of the time series exists and is available in some way in the VTL system. The VTL Operators are aware of which is the reference time and the period (frequency) of the time series and use these information to perform correct operations. VTL also assumes that a Value Domain representing the possible periods (e.g. the period indicator Value Domain shown above) exists and refers to the *duration* scalar type. For the assumptions above, the users do not need to specify which is the Identifier Component having the role of reference time.

The operators for time series can be applied only on Data Sets of time series and returns a Data Set of time series. The result Data Set has the same Identifier, Measure and Attribute Components as the operand Data Set and contains the same time series as the operand. The Attribute propagation rule is not applied.

Operators changing the data type

These Operators change the Scalar data type of the operands they are applied to (i.e. the type of the result is different from the type of the operand). For example, the **length** operator is applied to a value of *string* type and returns a value of *integer* type. Another example is the **cast** operator.

Operation on Scalar values

The operator is applied on (one or more) Scalar values and returns one Scalar value of a different data type.

Operation on Data Sets

If an Operator change the data type of the Variable it is applied to (e.g., from *string* to *number*), the result Data Set cannot maintain this Variable as it happens in the previous cases, because a Variable cannot have different data types in different Data Sets [48].

As a consequence, the converted variable cannot follow the same rules described in the sections above and must be replaced, in the result Data Set, by another Variable of the proper data type.

For sake of simplicity, the operators changing the data type are allowed only on mono-measure operand Data Sets, so that the conversion happens on just one Measure. A default generic Measure is assigned by default to the result Data Set, depending on the data type of the result (the default Measure Variables are reported in the table below).

Therefore, if the operands are originally multi-measure, just one Measure must be pre-emptively selected (for example through the membership operator) in order to apply the changing-type operator. Moreover, if in the result Data Set a different Measure Variable name is desired than the one assigned by default, it is possible to change the Variable name (see the **rename** operator).

As for the Identifiers and the Attributes, the behaviour of these operators is the same as the typical behaviour of the unary or binary operators.

Operation on Data Set Components

For the same reasons above, the result Component cannot be the same as one of the operand Components and must be of the appropriate Scalar data type.

Default Names for Variables and Value Domains used in this manual

The following table shows the default Variable names and the relevant default Value Domain. These are only the names used in this manual for explanatory purposes and can be personalised in the implementations. If VTL rules are exchanged, the personalised names need to be shared with the partners of the exchange.

Scalar data type	Default Variable	Default Value Domain
string	string_var	string_vd
number	num_var	num_vd
integer	int_var	int_vd

time	time_var	time_vd
time_period	time_period_var	time_period_vd
date	date_var	date_vd
duration	duration_var	duration_vd
boolean	bool_var	bool_vd

Type Conversion and Formatting Mask

The conversions between *scalar* types is provided by the operator **cast**, described in the section of the general purpose operators. Some particular types of conversion require the specification of a formatting mask, which specifies which format the source or the destination of the conversion should assume. The formatting masks for the various scalar types are explained here.

If needed, the formatting Masks can be personalized in the VTL implementations. If VTL rules are exchanged, the personalised masks need to be shared with the partners of the exchange.

The Numbers Formatting Mask

The **number formatting mask** can be defined as a combination of characters whose meaning is the following:

- “D” one numeric digit (for the mantissa of the scientific notation)
- “E” one numeric digit (for the exponent of the scientific notation)
- “*” an arbitrary number of digits
- “+” at least one digit
- “.” (dot) can be used as a separator between the integer and the decimal parts.
- “,” (comma) can be used as a separator between the integer and the decimal parts.

Examples of valid masks are:

DD.DDDDD, DD.D, D, D.DDDD, D*.D*, D+.D+ , DD.DDDEEEE

The Time Formatting Mask

The format of the values of the types *time*, *date* and *time_period* can be specified through specific formatting masks. A mask related to *time*, *date* and *time_period* is formed by a sequence of symbols which denote:

- the time units that are used, for example years, months, days
- the format in which they are represented, for example 4 digits for the year (2018), 2 digits for the month within the year (04 for April) and 2 digits for the day within the year and the month (05 for the 5th)
- the order of these parts; for example, first the 4 digits for the year, then the 2 digits for the month and finally the 2 digits for the day
- other (possible) typographical characters used in the representation; for example, a line between the year and the month and between the month and the day (e.g., 2018-04-05).

The time formatting masks follows the general rules below.

For a numerical representations of the time units:

- A digit is denoted through the use of a **special character** which depends on the time unit. for example Y is for “year”, M is for “month” and D is for “day”
- The special character is lowercase for the time units shorter than the day (for example h for “hour”, m for “minute”, s for “second”) and uppercase for time units equal to “day” or longer (for example W for “week”, Q for “quarter”, S for “semester”)
- The number of letters matches the number of digits, for example YYYY means that the year is represented with four digits and MM that the month is of 2 digits
- The numerical representation is assumed to be padded by leading 0 by default, for example MM means that April is represented as 04 and the year 33 AD as 0033

- If the numerical representation is not padded, the optional digits that can be omitted (if equal to zero) are enclosed within braces; for example {M}M means that April is represented by 4 and December by 12, while {YYY}Y means that the 33 AD is represented by 33

For textual representations of the time units:

- **Special words** denote a textual localized representation of a certain unit, for example DAY means a textual representation of the day (MONDAY, TUESDAY …)
- An optional number following the special word denote the maximum length, for example DAY3 is a textual representation that uses three characters (MON, TUE …)
- The case of the special word correspond to the case of the value; for example day3 (lowercase) denotes the values mon, tue …
- The case of the initial character of the special word correspond to the case of the initial character of the time format; for example Day3 denotes the values Mon, Tue …
- The letter P denotes the period indicator, (i.e., day, week, month …) and the letter p denotes the number of periods

Representation of more time units:

- If more time units are used in the same mask (for example years, months, days), it is assumed that the more detailed units (e.g., the day) are expressed through the order number that they assume within the less detailed ones (e.g., the month and the year). For example, if years, weeks and days are used, the weeks are within the year (from 1 to 53) and the days are within the year and the week (from 1 to 7).
- The position of the digits in the mask denotes the position of the corresponding values; for example, YYYMMDD means four digits for the year followed by two digits for the month and then two digits for the day (e.g., 20180405 means the year 2018, month April, day 5^{th})
- Any other character can be used in the mask, meaning simply that it appears in the same position; for example, YYYY-MM-DD means that the values of year, month and day are separated by a line (e.g., 2018-04-05 means the year 2018, month April, day 5^{th}) and \PMM denotes the letter “P” followed by two characters for the month.
- The special characters and the special words, if prefixed by the reverse slash (\) in the mask, appear in the same position in the time format; for example \PMM\M means the letter “P” followed by two characters for the month and then the letter “M”; for example, P03M means a period of three months (this is an ISO 8601 standard representation for a period of MM months). The reverse slash can appear in the format if needed by prefixing it with another reverse slash; for example YYYY\MM means for digits for the year, a reverse slash and two digits for the month.

The **special characters** and the corresponding time units are the following:

C century

Y year

S semester

Q quarter

M month

W week

D day

h hour digit (by default on 24 hours)

m minute

s second

d decimal of second

P period indicator (see the “duration” codes below)

p number of periods

The **special words** for textual representations are the following:

AM/PM indicator of AM / PM (e.g. am/pm for “am” or “pm”)

MONTH textual representation of the month (e.g., JANUARY for January)

DAY textual representation of the day (e.g., MONDAY for Monday)

Examples of formatting masks for the time scalar type:

A Scalar Value of type *time* denotes time intervals of any duration and expressed with any precision, which are the intervening time between two time points.

These examples are about three possible ISO 8601 formats for expressing time intervals:

- Start and end time points, such as “2015-03-03T09:30:45Z/2018-04-05T12:30:15Z”
  VTL Mask: YYYY-MM-DDThh:mm:ssZ/YYYY-MM-DDThh:mm:ssZ
- Start and duration, such as “2015-03-03T09:30:45-01/P1Y2M10DT2H30M”
  VTL Mask: YYYY-MM-DDThh:mm:ss-01/PY\YM\MDD\DT{h}h\Hmm\M
- Duration and end, such as “P1Y2M10DT2H30M/2018-04-05T12:30:00+02”
  VTL Mask: PY\YM\MDD\DT{h}h\Hmm\M/YYYY-MM-DDThh:mm:ssZ

Example of other possible ISO formats having accuracy reduced to the day

- Start and end, such as “20150303/20180405”
  VTL Mask: YYYY-MM-DD/YYYY-MM-DD
- Start and duration, such as “2015-03-03/P1Y2M10D”
  VTL Mask: YYYY-MM-DD/PY\YM\MDD\D
- Duration and end, such as “P1Y2M10D/2018-04-05”
  VTL Mask: PY\YM\MDD\DT/YYYY-MM-DD

Examples of formatting masks for the date scalar type:

A *date* scalar type is a point in time, equivalent to an interval of time having coincident start and end duration equal to zero.

These examples about possible ISO 8601 formats for expressing dates:

- Date and day time with separators: “2015-03-03T09:30:45Z”
  VTL Mask: YYYY-MM-DDThh:mm:ssZ
- Date and day time without separators “20150303T093045-01 “
  VTL Mask: YYYYMMDDThhmmss-01

Example of other possible ISO formats having accuracy reduced to the day

- Date and day-time with separators “2015-03-03/2018-04-05”
  VTL Mask: YYYY-MM-DD/YYYY-MM-DD
- Start and duration, such as “2015-03-03/P1Y2M10D”
  VTL Mask: YYYY-MM-DD/PY\YM\MDD\D

Examples of formatting masks for the time_period scalar type:

A *time_period* denotes non-overlapping time intervals having a regular duration (for example the years, the quarters of years, the months, the weeks and so on). The time_period values include the representation of the duration of the period.

These examples are about possible formats for expressing time-periods:

- Generic time period within the year such as: “2015Q4”, “2015M12””2015D365”
  VTL Mask: YYYYP{ppp} where P is the period indicator and ppp three digits for the number of periods, in the values, the period indicator may assume one of the values of the duration scalar type listed below.
- Monthly period: “2015M03”
  VTL Mask: YYYY\MMM

Examples of formatting masks for the duration scalar type:

A Scalar Value of type *duration* denotes the length of a time interval expressed with any precision and without connection to any particular time point (for example one year, half month, one hour and fifteen minutes).

These examples are about possible formats for expressing durations (period / frequency)

- Non ISO representation of the *duration* in one character, whose possible codes are:
  Code Duration

  D Day

  W Week

  M Month

  Q Quarter

  S Semester

  A Year

VTL Mask: P (period indicator)

- ISO 8601 composite duration: “P10Y2M12DT02H30M15S” (P stands for “period”)
  VTL Mask: \PYY\YM\MDD\DThh\Hmm\Mss\S
- ISO 8601 duration in weeks: “P018W” (P stands for “period”)
  VTL Mask: \PWWW\W
- ISO 4 characters representation: P10M (ten months), P02Q (two quarters) …
  VTL Mask: \PppP

Examples of fixed characters used in the ISO 8601 standard which can appear as fixed characters in the relevant masks:

P designator of duration

T designator of time

Z designator of UTC zone

“+” designator of offset from UTC zone

”-“ designator of offset form UTC zone

/ time interval separator

Attribute propagation

The VTL has different default behaviours for Attributes and for Measures, to comply as much as possible with the relevant manipulation needs. At the Data Set level, the VTL Operators manipulate by default only the Measures and not the Attributes. At the Component level, instead, Attributes are calculated like Measures, therefore the algorithms for calculating Attributes, if any, can be specified explicitly in the invocation of the Operators. This is the behaviour of clauses like **calc**, **keep**, **drop**, **rename** and so on, either inside or outside the join (see the detailed description of these operators in the Reference Manual).

The users which want to automatize the propagation of the Attributes’ Values can optionally enforce a mechanism, called Attribute Propagation rule, whose behaviour is explained in the User Manual (see the section “Behaviour for Attribute Components”). The adoption of this mechanism is optional, users are free to allow the attribute propagation rule or not. The users that do not want to allow Attribute propagation rules simply will not implement what follows.

In short, the automatic propagation of an Attribute depends on a Boolean characteristic, called “virality”, which can be assigned to any Attribute of a Data Set (a viral Attribute has virality = TRUE, a non-viral Attribute has virality=FALSE, if the virality is not defined, the Attribute is considered as non-viral).

By default, an Attribute propagates from the operand Data Sets (DS_i) to the result Data Set (DS_r) if it is “viral” at least in one of the operand Data Sets. By default, an Attribute which is viral in one of the operands DS_i is considered as viral also in the result DS_r.

The Attribute propagation rule does not apply for the time series operators.

The Attribute propagation rule does not apply if the operations on the Attributes to be propagated are explicitly specified in the expression (for example through the **keep** and **calc** operators). This way it is possible to keep in the result also Attribute which are non-viral in all the operands, to drop viral Attributes, to override the (possible) default calculation algorithm of the Attribute, to change the virality of the resulting Attributes.

45 As obvious, the input Data Set can be the result of a previous composition of more other Data Sets, even within the same expression

46 As obvious, the input Data Set can be the result of a previous composition of more other Data Sets, even within the same expression

47 According to the VTL IM, the Variables that have the same name have also the same data type

48 This according both to the mathematical meaning of a Variable and the VTL Information Model; in fact a Represented Variable is defined on just one Value Domain, which has just one data type, independently of the Data Structures and the Data Sets in which the Variable is used.

Operators

VTL-ML - General Purpose Operators

Parentheses: *()*

Syntax

(op **)**

Input parameters

op	the operand to be evaluated before performing other operations written outside the parentheses. According to the general VTL rule, operators can be nested, therefore any Data Set, Component or scalar op can be obtained through an expression as complex as needed (for example op can be written as the expression 2 + 3).

Examples of valid syntaxes

```
( DS_1 + DS_2 )
( CMP_1 - CMP_2 )
( 2 + DS_1 )
( DS_2 - 3 * DS_3 )
```

Semantics for scalar operations

Parentheses override the default evaluation order of the operators that are described in the section “VTL-ML - Evaluation order of the Operators”. The operations enclosed in the parentheses are evaluated first. For example `(2+3)*4` returns `20`, instead `2+3*4` returns `14` because the multiplication has higher precedence than the addition.

Input parameters type

op

```
dataset
| component
| scalar
```

Result type

result

```
dataset
| component
| scalar
```

Additional Constraints

None.

Behaviour

As mentioned, the *op* of the parentheses can be obtained through an expression as complex as needed (for example *op* can be written as `DS_1 - DS_2`). The part of the expression inside the parentheses is evaluated before the part outside of the parentheses. If more parentheses are nested, the inner parentheses are evaluated first, for example `( 20 - 10 / (2 + 3) ) * 3` would give `54`.

Examples

Given the operand datasets DS_1 and DS_2:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1	Me_2
10	A	5	5.0
10	B	2	10.5
11	A	3	12.2
11	B	4	20.3

Input **DS_2** (see structure)

Id_1	Id_2	Me_1	Me_2
10	A	10	3.0
10	C	11	6.2
11	B	6	7.0

Example 1

```
DS_r := (DS_1 + DS_2) * DS_2;
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
10	A	150	24.0
11	B	60	191.1

Persistent assignment: <-

Syntax

re **<-** op

Input parameters

re	the result
right	the operand. According to the general VTL rule allowing the indentation of the operators, op can be obtained through an expression as complex as needed (for example op can be the expression DS_1 - DS_2).

Examples of valid syntaxes

```
DS_r <- DS_1
DS_r <- DS_1 - DS_2
```

Semantics for scalar operations

Empty

Input parameters type

op

`dataset`

Result type

result

`dataset`

Additional Constraints

The assignment cannot be used at Component level because the result of a Transformation cannot be a Data Set Component. When operations at Component level are invoked, the result is the Data Set which the output Components belongs to.

Behaviour

The input operand *op* is assigned to the **persistent** result *re*, which assumes the same value as *op*. As mentioned, the operand *op* can be obtained through an expression as complex as needed (for example *op* can be the expression `DS_1 - DS_2`).

The result *re* is a persistent Data Set that has the same data structure as the Operand. For example in `DS_r <- DS_1` the data structure of `DS_r` is the same as the one of `DS_1`.

If the Operand *op* is a scalar value, the result Data Set has no Components and contains only such a scalar value. For example, `income <- 3` assigns the value 3 to the persistent Data Set named income.

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1	Me_2
2013	Belgium	5	5
2013	Denmark	2	10
2013	France	3	12
2013	Spain	4	20

Example 1

```
DS_r <- DS_1;
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
2013	Belgium	5	5
2013	Denmark	2	10
2013	France	3	12
2013	Spain	4	20

Non-persistent assignment: :=`

Syntax

re **:=** op

Input parameters

re	the result
right	the operand. According to the general VTL rule allowing the indentation of the operators, *op* can be obtained through an expression as complex as needed (for example *op* can be the expression `DS_1 - DS_2`).

Examples of valid syntaxes

```
DS_r := DS_1
DS_r := 3
DS_r := DS_1 - DS_2
DS_r := 3 + 2
```

Semantics for scalar operations

Empty

Input parameters type

op

```
dataset
| scalar
```

Result type

re

```
dataset
```

Additional Constraints

The assignment cannot be used at Component level because the result of a Transformation cannot be a Data Set Component. When operations at Component level are invoked, the result is the Data Set which the output Components belongs to.

The same symbol denoting the non-persistent assignment Operator (**:=**) is also used inside other operations at Component level (for example in **calc** and **aggr**) in order to assign the result of the operation to the output Component: please note that in these cases the symbol **:=** does not denote the non-persistent assignment (i.e., this Operator), which cannot operate at Component level, but a special keyword of the syntax of the other Operator in which it is used.

Behaviour

The value of the operand *op* is assigned to the result *re*, which is non-persistent and therefore is not stored. As mentioned, the operand *op* can be obtained through an expression as complex as needed (for example op can be the expression `DS_1 - DS_2`).

The result re is a non-persistent Data Set that has the same data structure as the Operand. For example in `DS_r := DS_1` the data structure of `DS_r` is the same as the one of `DS_1`.

If the Operand *op* is a scalar value, the result Data Set has no Components and contains only such a scalar value. For example, `income := 3` assigns the value 3 to the non-persistent Data Set named *income*.

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1	Me_2
2013	Belgium	5	5
2013	Denmark	2	10
2013	France	3	12
2013	Spain	4	20

Example 1

```
DS_r := DS_1;
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
2013	Belgium	5	5
2013	Denmark	2	10
2013	France	3	12
2013	Spain	4	20

Membership:

Syntax

ds # comp

Input parameters

ds	the Data Set
cop	the Data Set Component

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

ds

`dataset`

comp

`name<component>`

Result type

result

`dataset`

Additional Constraints

comp must be a Data Set Component of the Data Set *ds*.

Behaviour

The membership operator returns a Data Set having the same Identifier Components of *ds* and a single Measure. If *comp* is a Measure in *ds*, then *comp* is maintained in the result while all other Measures are dropped. If *comp* is an Identifier or an Attribute Component in *ds*, then all the existing Measures of *ds* are dropped in the result and a new Measure is added. The Data Points' values for the new Measure are the same as the values of *comp* in *ds*. A default conventional name is assigned to the new Measure depending on its type: for example *num_var* if the Measure is *numeric*, *string_var* if it is *string* and so on (the default name can be renamed through the **rename** operator if needed). The Attributes follow the Attribute propagation rule as usual (viral Attributes of *ds* are maintained in the result as viral, non-viral ones are dropped). If *comp* is an Attribute, it follows the Attribute propagation rule too. The same symbol denoting the membership operator (#) is also used inside other operations at Component level (for example in **join**, **calc**, **aggr**) in order to identify the Components to be operated: please note that in these cases the symbol # does not denote the membership operator (i.e., this operator, which does not operate at Component level), but a special keyword of the syntax of the other operator in which it is used.

Examples

Given the operand datasets DS_1 and DS_2, where the attribute component At_1 is viral for DS_2 and non-viral for DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1	Me_2	At_1
1	A	1	5	
1	B	2	10	P
2	A	3	12	

Input **DS_2** (see structure)

Id_1	Id_2	Me_1	Me_2	At_1
1	A	1	5	
1	B	2	10	P

2	A	3	12	

Example 1

```
DS_r := DS_1#Me_1;
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1
1	A	1
1	B	2
2	A	3

Example 2

```
DS_r := DS_1#Id_1;
```

results in (see structure):

DS_r

Id_1	Id_2	int_var
1	A	1
1	B	1
2	A	2

Example 3

```
DS_r := DS_1#At_1;
```

results in (see structure):

DS_r

Id_1	Id_2	string_var
1	A	
1	B	P
2	A	

Example 4

```
DS_r := DS_2#Me_1;
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	At_1
1	A	1	

1	B	2	P
2	A	3	

Example 5

```
DS_r := DS_2#Id_1;
```

results in (see structure):

DS_r

Id_1	Id_2	int_var	At_1
1	A	1	
1	B	1	P
2	A	2	

Example 6

```
DS_r := DS_2#At_1;
```

results in (see structure):

DS_r

Id_1	Id_2	string_var	At_1
1	A		
1	B	P	P
2	A		

User-defined operator call

Syntax

operatorName **(** { argument { , argument }* } **)**

Input parameters

operatorName	the name of an existing user-defined operator
argument	argument passed to the operator

Examples of valid syntaxes

```
max1 ( 2, 3 )
```

Semantics for scalar operations

It depends on the specific user-defined operator that is invoked.

Input parameters type

operatorName

```
name
```

argument

```
A data type compatible with the type of the parameter of the user-defined operator
that is invoked (see also the “Type syntax” section).
```

Result type

result

```
The data type of the result of the user-defined operator that is invoked
(see also the “Type syntax” section).
```

Additional Constraints

- *operatorName* must refer to an operator created with the **define operator** statement.
- The type of each argument value must be compliant with the type of the corresponding parameter of the user defined operator (the correspondence is in the positional order).

Behaviour

The invoked user-defined operator is evaluated. The arguments passed to the operator in the invocation are associated to the corresponding parameters in positional order, the first argument as the value of the first parameter, the second argument as the value of the second parameter, and so on. An underscore (“_”) can be used to denote that the value for an optional operand is omitted. One or more optional operands in the last positions can be simply omitted.

Examples

Example 1

Definition of the *max1* operator (see also “define operator” in the VTL-DL):

```
define operator max1 (x integer, y integer)
returns boolean
is if x > y then x else y
end define operator
```

User-defined operator call of the max1 operator:

```
max1 ( 2, 3 )
```

Evaluation of an external routine: *eval*

Syntax

eval (externalRoutineName **(** { argument } { , argument }* **)** language languageName **returns** outputType **)**

Input parameters

externalRoutineName	the name of an external routine
argument	the arguments passed to the external routine
language	the implementation language of the routine
outputType	the data type of the object returned by eval (see outputParameterType in Data type syntax)

Examples of valid syntaxes

```
eval(routine1(DS_1) language “PL/SQL” returns string)
```

Semantics for scalar operations

This is not a scalar operation.

Input parameters type

externalRoutineName

`name`

argument

`any data type`

language

`string`

outputType

`any data type restricting Data Set or scalar`

Result type

result

`dataset`

Additional Constraints

- The **eval** is the only VTL Operator that does not allow nesting and therefore a Transformation can contain just one invocation of **eval** and no other invocations. In other words, **eval** cannot be nested as the operand of another operation as well as another operator cannot be nested as an operand of **eval**
- The result of an expression containing **eval** must be persistent
- *externalRoutineName* is the conventional name of a non-VTL routine
- The invoked external routine must be consistent with the VTL principles, first of all its behaviour must be functional, so having in input and providing in output first-order functions
- argument is an argument passed to the external routine, it can be a name or a value of a VTL artefacts or some other parameter required by the routine
- The arguments passed to the routine correspond to the parameters of the invoked external routine in positional order; as usual the optional parameters are substituted by the underscore if missing. The conversion of the VTL input/output data types from and to the external routine processor is left to the implementation.

Behaviour

The **eval** operator invokes an external, non-VTL routine, and returns its result as a Data Set or a scalar. The specific data type can be given in the invocation. The routine specified in the **eval** operator can perform any internal logic.

Examples

Assuming that SQL3 is an SQL statement which produces DS_r starting from DS_1:

```
DS_r := eval( SQL3( DS_1 ) language “PL/SQL”
        returns dataset { identifier<geo_area> ref_area,
                          identifier<date> time_,
                          measure<number> obs_value,
                          attribute<string> obs_status } );
```

Assuming that f is an externally defined Java method:

```
DS_r := DS_1 [calc Me := eval ( f (Me ) language “Java” returns integer) ];
```

Type conversion: *cast*

Syntax

cast (op , scalarType { , mask} **)**

Input parameters

op	the operand to be cast
scalarType	the name of the scalar type into which *op* has to be converted
mask	a character literal that specifies the format of *op*

Semantics for scalar operations

This operator converts the scalar type of *op* to the scalar type specified by *scalarType*. It returns a copy of *op* converted to the specified *scalarType*.

Input parameters type

op

```
dataset{ measure<scalar> _ }
| component<scalar>
| scalar
```

scalarType

```
scalar type (see the section: Data type syntax)
```

mask

```
string
```

Result type

result

```
dataset{ measure<scalar> _ }
| component<scalar>
| scalar
```

Additional Constraints

- Not all the conversions are possible, the specified casting operation is allowed only according to the semantics described below.
- The *mask* must adhere to one of the formats specified below.

Behaviour

Conversions between basic scalar types

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

The external representations are those of the Value Domains which refers to such a basic scalar types (more Value Domains can refer to the same basic scalar type, see the VTL Data Types in the User Manual). For example, there can exist a *boolean* Value Domain which uses the values TRUE and FALSE and another *boolean* Value Domain which uses the values 1 and 0. The external representations are the ones of the Data Point Values and are obviously known by users.

The unique internal representation of a basic scalar type, instead, is used by the **cast** operator as a technical expedient to make the conversion between external representations easier: users are not necessarily aware of it. In a conversion, the **cast** converts the source external representation into the internal representation (of the corresponding scalar type), then this last one is converted into the target external representation (of the target type). As mentioned in the User Manual, VTL does not prescribe any specific internal representation for the various scalar types, leaving different organisations free of using their preferred or already existing ones.

In some cases, depending on the type of *op*, the output *scalarType* and the invoked operator, an automatic conversion is made, that is, even without the explicit invocation of the **cast** operator: this kind of conversion is called **implicit casting**.

In other cases, more than all when the implicit casting is not possible, the type conversion must be specified explicitly through the invocation of the **cast operator**: this kind of conversion is called **explicit casting**. If an explicit casting is specified, the (possible) implicit casting is overridden; the explicit conversion requires a formatting mask that specifies how the actual casting is performed.

The table below summarises the possible castings between the basic scalar types. In particular, the input type is specified in the first column (row headings) and the output type in the first row (column headings).

Provided (down) / expected (right)	integer	number	boolean	time	date	time_period	string	duration
integer	•	Implicit	Implicit	Not feasible	Not feasible	Not feasible	Implicit	Not feasible
number	Implicit	•	Implicit	Not feasible	Not feasible	Not feasible	Implicit	Not feasible
boolean	Implicit	Implicit	•	Not feasible	Not feasible	Not feasible	Implicit	Not feasible
time	Not feasible	Not feasible	Not feasible	•	Not feasible	Not feasible	Explicit with mask	Not feasible
date	Not feasible	Not feasible	Not feasible	Implicit	•	Explicit w/o mask	Explicit with mask	Not feasible
time_period	Not feasible	Not feasible	Not feasible	Implicit	Explicit with mask	•	Implicit	Not feasible
string	Implicit	Explicit with mask	Not feasible	Explicit with mask	Explicit with mask	Explicit with mask	•	Explicit with mask
duration	Not feasible	Not feasible	Not feasible	Not feasible	Not feasible	Not feasible	Explicit with mask	•

The type of casting can be personalised in specific environments, provided that the personalisation is explicitly documented with reference to the table above. For example, assuming that an explicit **cast** with *mask* is required and that in a specific environment a definite *mask* is used for such a kind of conversions, the **cast** can also become implicit provided that the mask that will be applied is specified.

The **implicit casting** is performed when a value of a certain type is provided when another type is expected. Its behaviour is described here:

- From **integer** to **number**: an *integer* is provided when a *number* is expected (for example, an *integer* and a *number* are passed as inputs of a n-ary numeric operator); it returns a *number* having the integer part equal to the *integer* and the decimal part equal to zero;
- From **integer** to **string**: an *integer* is provided when a *string* is expected (for example, an *integer* is passed as an input of a *string* operator); it returns a *string* having the literal value of the *integer*;
- From **number** to **string**: a *number* is provided when a *string* is expected; it returns the *string* having the literal value of the *number*; the decimal separator is converted into the character “.” (dot).
- From **boolean** to **string**: a *boolean* is provided when a *string* is expected; the *boolean* value TRUE is converted into the *string* “TRUE” and FALSE into the *string* “FALSE”;

- From **date** to **time**: a *date* (point in time) is provided when a *time* is expected (interval of time): the conversion results in an interval having the same start and end, both equal to the original *date*;
- From **time_period** to **time**: a *time_period* (a regular interval of *time*, like a month, a quarter, a year…) is provided when a *time* (any interval of time) is expected; it returns a *time* value having the same start and end as the *time_period* value.
- From **integer** to **boolean**: if the *integer* is different from 0, then TRUE is returned, FALSE otherwise.
- From **number** to **integer**: converts a *number* with no decimal part into an *integer*; if the decimal part is present, a runtime error is raised.
- From **number** to **boolean**: if the *number* is different from 0.0, then TRUE is returned, FALSE otherwise.
- From **boolean** to **integer**: TRUE is converted into 1; FALSE into 0.
- From **boolean** to **number**: TRUE is converted into 1.0; FALSE into 0.0.
- From **time_period** to **string**: it is applied the *time_period* formatting mask.
- From **string** to **integer**: the *integer* having the literal value of the *string* is returned; if the *string* contains a literal that cannot be matched to an *integer*, a runtime error is raised.

An implicit cast is also performed from a **value domain type** or a **set type** to a **basic scalar type**: when a *scalar* value belonging to a Value Domains or a Set is involved in an operation (i.e., provided as input to an operator), the value is implicitly cast into the basic scalar type which the Value Domain refers to (for this relationship, see the description of Type System in the User Manual). For example, assuming that the Component *birth_country* is defined on the Value Domain *country*, which contains the ISO 3166-1 numeric codes and therefore refers to the basic scalar type *integer*, the (possible) invocation `length(birth_country)`, which calculates the length of the input string, automatically casts the values of *birth_countr* into the corresponding string. If the basic scalar type of the Value Domain is not compatible with the expression where it is used, an error is raised. This VTL feature is particularly important as it provides a general behaviour for the Value Domains and relevant Sets, preventing from the need of defining specific behaviours (or methods or operations) for each one of them. In other words, all the Values inherit the operations that can be performed on them from the basic scalar types of the respective Value Domains.

The **cast** operator can be invoked explicitly even for the conversions which allow an implicit cast and in this case the same behaviour as the implicit cast is applied.

When an **explicit casting with mask** is required, the conversion is made by applying the formatting mask which specifies the meaning of the characters in the output *string*. The formatting Masks are described in the section “VTL-ML – Typical Behaviour of the ML Operators”, sub-section “Type Conversion and Formatting Mask.

The behaviour of the **cast** operator for such conversions is the following:

- From **time** to **string**: it is applied the *time* formatting mask.
- From **date** to **time_period**: it converts a *date* into the corresponding daily value of *time_period*.
- From **date** to **string**: it is applied the *time_period* formatting mask.
- From **time_period** to **date**: it is applied a formatting mask which accepts two possible values (“START”, “END”). If “START” is specified, then the *date* is set to the beginning of the *time_period*; if *END* is specified, then the *date* is set to the end of the *time_period*.
- From **string** to **number**: the *number* having the literal value of the *string* is returned; if the *string* contains a literal that cannot be matched to a *number*, a runtime error is raised. The *number* is generated by using a *number* formatting mask.
- From **string** to **time**: the *time* having the literal value of the *string* is returned; if the *string* contains a literal that cannot be matched to a *date*, a runtime error is raised. The *time* value is generated by using a *time* formatting mask.
- From **string** to **date**: it converts a *string* value to a *date* value.
- From **string** to **time_period**: it converts a *string value* to a *time_period* value.
- From **string** to **duration**: the *duration* having the literal value of the *string* is returned; if the *string* contains a literal that cannot be matched to a *duration*, a runtime error is raised. The *duration* value is generated by using a *time* formatting mask.
- From **duration** to **string**: a *duration* (an absolute time interval) is provided when a *string* is expected; it returns the *string* having the default *string* representation for the *duration*.

Conversions between basic scalar types and Value Domains or Set types

A value of a basic *scalar* type can be converted into a value belonging to a Value Domain which refers to such a *scalar* type. The resulting *scalar* value must be one of the allowed values of the Value Domain or Set; otherwise, a runtime error is raised. This specific use of **cast** operators does not really correspond to a type conversion; in more formal terms, we would say that it acts as a constructor, i.e., it builds an instance of the output type. Yet, towards a homogeneous and possibly simple definition of VTL syntax, we blur the distinction between constructors and type conversions and opt for a unique formalism. An example is given below.

Conversions between different Value Domain types

As a result of the above definitions, conversions between values of different Value Domains are also possible. Since an element of a Value Domain is implicitly cast into its corresponding basic scalar type, we can build on it to turn the so obtained scalar type into another Value Domain type. Of course, this latter Value Domain type must use as a base type this scalar type.

Examples

Example 1

From string to number:

```
ds2 := ds1[calc m2 := cast(m1, number, “DD.DDD”) + 2 ];
```

In this case we use explicit cast from *string* to *numbers*. The mask is used to specify how the *string* must be interpreted in the conversion.

Example 2

From string to date:

```
ds2 := ds1[calc m2 := cast(m1, date, “YYYY-MM-DD”) ];
```

In this case we use explicit cast from *string* to *date*. The mask is used to specify how the *string* must be interpreted in the conversion.

Example 3

From number to integer:

```
ds2 := ds1[calc m2 := cast(m1, integer) + 3 ];
```

In this case we cast a *number* into an *integer*, no mask is required.

Example 4

From number to string:

```
ds2 := ds1[calc m2 := length(cast(m1, string)) ];
```

In this case we cast a *number* into a *string*, no mask is required.

Example 5

From date to string:

```
ds2 := ds1[calc m2 := cast(m1, string, “YY-MON-DAY hh:mm:ss”) ];
```

In this example a *date* instant is turned into a *string*. The mask is used to specify the *string* layout.

Example 6

From string to GEO_AREA:

```
ds2 := ds1[calc m2 := cast(GEO_STRING, GEO_AREA)];
```

In this example we suppose we have elements of Value Domain Subset for GEO_AREA. Let GEO_STRING be a string Component of Data Set *ds1* with string values compatible with the GEO_AREA Value Domain Subset. Thus, the following expression moves *ds1* data into *ds2*, explicitly casting strings to geographical areas.

Example 7

From GEO_AREA to string:

```
ds2 := ds1[calc m2 := length(GEO_AREA)];
```

In this example we use a Component GEO_AREA in a *string* expression, which calculates the length of the corresponding *string*; this triggers the automatic cast.

Example 8

From GEO_AREA2 to GEO_AREA1:

```
ds2 := ds1 [ calc m2 := cast (GEO, GEO_AREA1) ];
```

In this example we suppose we have to compare elements two Value Domain Subsets. They are both defined on top of Strings. The following cast expressions performs the conversion.

Now, Component GEO is of type GEO_AREA2, then we specify it has to be cast into GEO_AREA1. As both work on *strings* (and the values are compatible), the conversion is feasible. In other words, the cast of an operand into GEO_AREA1 would expect a *string*. Then, as GEO is of type GEO_AREA2, defined on top of *strings*, it is implicitly cast to the respective *string*; this is compatible with what cast expects and it is then able to build a value of type GEO_AREA1.

Example 9

From string to time_period:

In the following examples we convert from strings to time_periods, by using appropriate masks.

The first quarter of year 2000 can be expressed as follows (other examples are possible):

```
cast ( “2000Q1”, time_period, “YYYY\QQ” )
cast ( “2000-Q1”, time_period, “YYYY-\QQ” )
cast ( “2000-1”, time_period, “YYYY-Q” )
cast ( “Q1-2000”, time_period, “\QQ-YYYY” )
cast ( “2000Q01”, time_period, “YYYY\QQQ” )
```

Examples of daily data:

```
cast ( “2000M01D01”, time_period, “YYYY\MMM\DDD” )
cast ( “2000.01.01”, time_period, “YYYY\.MM\.DD” )
```

VTL-ML - Join operators

The Join operators are fundamental VTL operators. They are part of the core of the language and allow to obtain the behaviour of the majority of the other non-core operators, plus many additional behaviours that cannot be obtained through the other operators.

The Join operators are four, namely the inner_join, the left_join, the full_join and the cross_join. Because their syntax is similar, they are described together.

Join: *inner_join*, *left_join*, *full_join*, *cross_join*

Syntax

joinOperator (ds1 { as alias1 }, ds2 { as alias2 } { , dsN { as aliasN } }*

{ **using** usingComp { , usingComp }* }
{ **filter** filterCondition }

{ **apply** applyExpr
| **calc** calcClause
| **aggr** aggrClause { groupingClause } }
{ **keep** comp {, comp }* | **drop** comp {, comp }* }
{ **rename** compFrom **to** compTo { , compFrom **to** compTo }* } **)**

joinOperator: { **inner_join | left_join | full_join | cross_join** }[1]

calcClause ::= { calcRole } calcComp **:=** calcExpr { , { calcRole } calcComp **:=** calcExpr }*

calcRole ::= { **identifier | measure | attribute | viral attribute** }[1]

aggrClause ::= { aggrRole } aggrComp **:=** aggrExpr { , { aggrRole } aggrComp **:=** aggrExpr }*

aggrRole ::= { **measure | attribute | viral attribute** }[1]

groupingClause ::= { **group by** groupingId { , groupingId }* | **group except** groupingId { , groupingId }* | **group all** conversionExpr }[1] { **having** havingCondition }

Input parameters

joinOperator	the Join operator to be applied
ds1, …, dsN	the Data Set operands (at least two must be present) [49]
alias	optional aliases for the input Data Sets, valid only within the “join” operation to make it easier to refer to them. If omitted, the Data Set name must be used.
usingComp	component of the input Data Sets whose values have to match in the join (the **using** clause is allowed for the **left_join** only under certain constraints described below and is not allowed at all for the **full_join** and **cross_join**)
filterCondition	a condition (*boolean* expression) at component level, having only Components of the input Data Sets as operands, which is evaluated for each joined Data Point and filters them (when TRUE the joined Data Point is kept, otherwise it is not kept)
applyExpr	an expression, having the input Data Sets as operands, which is pairwise applied to all their homonym Measure Components and produces homonym Measure Components in the result; for example if both the Data Sets *ds1* and *ds2* have the numeric measures *m1* and *m2*, the clause *apply ds1 + ds2* would result in calculating *m1 := ds1#m1 + ds2#m1 and m2 := ds1#m2 + ds2#m2*
calcClause	clause that specifies the Components to be calculated, their roles and their calculation algorithms, to be applied on the joined and filtered Data Points.
calcRole	the role of the Component to be calculated
calcComp	the name of the Component to be calculated

calcExpr	expression at component level, having only Components of the input Data Sets as operands, used to calculate a Component
aggrClause	clause that specifies the required aggregations, i.e., the aggregated Components to be calculated, their roles and their calculation algorithm, to be applied on the joined and filtered Data Points
aggrRole	the role of the aggregated Component to be calculated; if omitted, the Measure role is assumed
aggrComp	the name of the aggregated Component to be calculated; this is a dependent Component of the result (Measure or Attribute, not Identifier)
aggrExpr	expression at component level, having only Components of the input Data Sets as operands, which invokes an aggregate operator (e.g. **avg, count, max**…, see also the corresponding sections) to perform the desired aggregation. Note that the **count** operator is used in an aggrClause without parameters, e.g.: *DS_1 [aggr Me_1 := count ( ) group by Id_1)]*
groupingClause	the following alternative grouping options: · **group by**: the Data Points are grouped by the values of the specified Identifiers (*groupingId*). The Identifiers not specified are dropped in the result. · **group except**: the Data Points are grouped by the values of the Identifiers not specified as *groupingId*. The specified Identifiers are dropped in the result. · **group all**: converts the values of an Identifier Component using *conversionExpr* and keeps all the resulting Identifiers.
groupingId	Identifier Component to be kept (in the **group by** clause) or dropped (in the **group except** clause).
conversionExpr	specifies a conversion operator (e.g. **time_agg**) to convert an Identifier from finer to coarser granularity. The conversion operator is applied on an Identifier of the operand Data Set.

havingCondition	a condition (*boolean* expression) at component level, having only Components of the input Data Sets as operands (and possibly constants), to be fulfilled by the groups of Data Points: only groups for which *havingCondition* evaluates to TRUE appear in the result. The *havingCondition* refers to the groups specified through the *groupingClause*, therefore it must invoke aggregate operators (e.g. *avg, count, max*…, see also the section Aggregate invocation). A correct example of *havingCondition* is *max(obs_value) < 1000*, while the condition *obs_value < 1000* is not a right *havingCondition*, because it refers to the values of single Data Points and not to the groups. The count operator is used in a *havingCondition* without parameters, e.g.: *sum (ds group by id1 having count ( ) >= 10)*
comp	dependent Component (Measure or Attribute, not Identifier) to be kept (in the **keep** clause) or dropped (in the **drop** clause)
compFrom	the original name of the Component to be renamed
compTo	the new name of the Component after the renaming

49 In the previous versions for inner_join only one Data Set operand had to be specified; to be more consistent and similarly to SQL syntax, at least to operands are required.

Examples of valid syntaxes

```
inner_join  ( ds1 as d1,  ds2 as d2 using Id1, Id2
            filter d1#Me1 + d2#Me1 <10
            apply  d1 / d2
            keep Me1, Me2, Me3
            rename Id1 to Id10, id2 to id20
            )
left_join ( ds1 as d1,  ds2 as d2
            filter d1#Me1 + d2#Me1 <10,
            calc Me1 := d1#Me1 + d2#Me3,
            keep  Me1
            rename Id1 to Ident1, Me1 to Meas1
            )
full_join ( ds1 as d1,  ds2 as d2
            filter d1#Me1 + d2#Me1 <10,
            aggr Me1 := sum(Me1), attribute At20 := avg(Me2)
            group by  Id1, Id2
            having sum(Me3) > 0
            )
```

Semantics for scalar operations

The join operator does not perform scalar operations.

Input parameters type

ds1, …, dsN

`dataset`

alias1, …, aliasN

`name`

usingId

`name<component>`

filterCondition

`component<boolean>`

applyExpr

`dataset`

calcComp

`name<component>`

calcExpr

`component<scalar>`

aggrComp

`name<component>`

aggrExpr

`component<scalar>`

groupingId

`name<identifier>`

conversionExpr

`component<scalar>`

havingCondition

`component<boolean>`

comp

`name<component>`

compFrom

`component<scalar>`

compTo

`component<scalar>`

Result type

result

`dataset`

Additional Constraints

The aliases must be all distinct and different from the Data Set names. Aliases are mandatory for Data Sets which appear more than once in the Join (self-join) and for non-named Data Set obtained as result of a sub-expression. The *using* clause is not allowed for the **full_join** and for the **cross_join**, because otherwise a non-functional result could be obtained.

If the *using* clause is not specified (we will label this case as “Case A”), calling *Id(ds■)* the set of Identifier Components of operand *ds*■, the following group of constraints must hold:

- For **inner_join**, for each pair *ds*■, *ds*■, either *d(ds■) ⊆ Id(ds■)* or *Id(ds) ⊆ Id(ds■)*. In simpler words, the Identifiers of one of the joined Data Sets must be a superset of the identifiers of all the other ones.
- For **left_join** and **full_join**, for each pair *ds*■, *ds*■, *Id(ds■) = Id(ds■)*. In simpler words, the joined Data Sets must have the same Identifiers.
- For **cross-join** (Cartesian product), no constraints are needed.

If the *using* clause is specified (we will label this case as “Case B”, allowed only for the **inner_join** and the **left_join**), all the join keys must appear as Components in all the input Data Sets. Moreover two sub-cases are allowed:

- Sub-case B1: the constraints of the Case A are respected and the join keys are a subset of the common Identifiers of the joined Data Sets;
- Sub-case B2:
  - In case of **inner_join**, one Data Set acts as the reference Data Set which the others are joined to; in case of **left_join**, this is the left-most Data Set (i.e., ds■);
  - All the input Data Sets, except the reference Data Set, have the same Identifiers [Id■,…, Id■];
  - The *using* clause specifies all and only the common Identifiers of the non-reference Data Sets[Id■,…, Id■].

The join operators must fulfil also other constraints:

- **apply**, **calc** and **aggr** clauses are mutually exclusive
- **keep** and **drop** clauses are mutually exclusive
- *comp* can be only dependent Components (Measures and Attributes, not Identifiers)
- An Identifier not included in the **group by** clause (if any) cannot be included in the **rename** clause
- An Identifier included in the **group except** clause (if any) cannot be included in the **rename** clause. If the **aggr** clause is invoked and the grouping clause is omitted, no Identifier can be included in the **rename** clause
- A dependent Component not included in the **keep** clause (if any) cannot be renamed
- A dependent Component included in the **drop** clause (if any) cannot be renamed

Behaviour

The **semantics of the join operators** can be procedurally described as follows.

1. A *relational join* of the input operands is performed, according to SQL inner (**inner_join**), left-outer (**left_join**), full-outer (**full_join**) and Cartesian product (**cross_join**) semantics (these semantics will be explained below), producing an intermediate internal result, that is a Data Set that we will call “virtual” (VDS■).
2. The *filterCondition*, if present, is applied on VDS■, producing the Virtual Data Set VDS■.
3. The specified calculation algorithms (**apply**, **calc** or **aggr**), if present, are applied on VDS■. For the Attributes that have not been explicitly calculated in these clauses, the Attribute propagation rule is applied (see the User Manual), so producing the Virtual Data Set VDS■.
4. The **keep** or **drop** clause, if present, is applied on VDS■, producing the Virtual Data Set VDS■.
5. The **rename** clause, if present, is applied on VDS■, producing the Virtual Data Set VDS■.
6. The final automatic alias removal is performed in order to obtain the output Data Set.

An alias can be optionally declared for each input Data Set. The aliases are valid only within the “join” operation, in particular to allow joining a dataset with itself (self join). If omitted, the input Data Sets are referenced only through their Data Set names. If the aliases are ambiguous (for example duplicated or equal to the name of another Data Set), an error is raised.

The **structure of the virtual Data Set** VDS■ which is the output of the relational join is the following.

For the **inner_join**, the **left_join** and the **full_join**, the virtual Data Set contains the following Components:

- The Components used as join keys, which appear once and maintain their original names and roles. In The cases A and B1, all of them are Identifiers. In the sub-case B2, the result takes the roles from the reference Data Set.
- In the sub-case B2: the Identifiers of the reference Data Set, which appear once and maintain their original name and role.
- The other Components coming from exactly one input Data Set, which appear once and maintain their original name
- The other Components coming from more than one input Data Set, which appears as many times as the Data Set they come from; to distinguish them, their names are prefixed with the alias (or the name) of the Data Set they come from, separated by the "#" symbol (e.g., *ds■#cmp■*). For example, if the Component "*population*" appears in two input Data Sets "*ds1*" and "*ds2*" that have the aliases "*a*" and "*b*" respectively, the Components "*a#population*" and "*b#population*" will appear in the virtual Data Set. If the aliases are not defined, the two Components are prefixed with the Data Set name (i.e., "*ds1#population*" and "*ds2#population*"). In this context, the symbol "#" does not denote the membership operator but acts just as a separator between the the Data Set and the Component names.
- If the same Data Set appears more times as operand of the join (self-join) and the aliases are not defined, an exception is raised because it is not allowed that two or more Components in the virtual Data Set have the same name. In the self-join the aliases are mandatory to disambiguate the Component names.
- If a Data Set in the join list is the result of a sub-expression, then an alias is mandatory all the same because this Data Set has no name. If the alias is omitted, an exception is raised.

As for the **cross_join**, the virtual Data Set contains all the Components from all the operands, possibly prefixed with the aliases to avoid ambiguities.

The **semantics of the relational join** is the following.

The join is performed on some join keys, which are the Components of the input Data Sets whose values are used to match the input Data Points and produce the joined output Data Points.

By default (only for the **full_join** and the **cross_join**), the join is performed on the subset of homonym Identifier Components of the input Data Sets.

The parameter **using** allows to specify different join keys than the default ones, and can be used only for the **inner_join** and the **left_join** in order to preserve the functional behaviour of the operations.

The different kinds of relational joins behave as follows.

- **inner_join**: the Data Points of *ds1, …, dsN* are joined if they have the same values for the common Identifier Components or, if the **using** clause is present, for the specified Components. A (joined) virtual Data Point is generated in the virtual Data Set VDS■ when a matching Data Point is found for each one of the input Data Sets. In this case, the Values of the Components of a virtual Data Point are taken from the corresponding Components of the matching Data Points. If there is no match for one or more input Data Sets, no virtual Data Point is generated.
- **left_join**: the join is ideally performed stepwise, between consecutive pairs of input Data Sets, starting from the left side and proceeding towards the right side. The Data Points are matched like in the **inner_join**, but a virtual Data Point is generated even if no Data Point of the right Data Set matches (in this case, the Measures and Attributes coming from the right Data Set take the NULL value in the virtual Data Set). Therefore, for each Data Points of the left Data Set a virtual Data Point is always generated. These stepwise operations are associative. More formally, consider the generic pair <*ds■, ds■■■*>, where *ds■* is the result of the *left_join* of the first "i" operands and *ds■■■* is the i+1th operand. For each pair <*ds■, ds■■■*>, the joined Data Set is fed with all the Data Points that match in *ds■* and *ds■■■* or are only in *ds■*. The constraints described above guarantee the absence of null values for the Identifier Components of the joined Data Set, whose values are always taken from the left Data Set. If the join succeeds for a Data Point in *ds■*, the values for the Measures and the Attributes are carried from *ds■* and *ds■■■* as explained above. Otherwise, i.e., if no Data Point in *ds■■■* matches the Data Point in *ds■*, null values are given to Measures and Attributes coming only from *ds■■■*.
- **full_join**: the join is ideally performed stepwise, between consecutive pairs of input Data Sets, starting from the left side and proceeding toward the right side. The Data Points are matched like in the **inner_join** and **left_join**, but the **using** clause is not allowed and a virtual Data Point is generated either if no Data Point of the right Data Set matches with the left Data Point or if no Data Point of the left Data Set matches with the right Data Point (in this case, Measures and Attributes coming from the non matching Data Set take the NULL value

in the virtual Data Set). Therefore, for each Data Points of the left and the right Data Set, a virtual Data Point is always generated. These stepwise operations are associative. More formally, consider the generic pair <*ds*■*, ds*■■■>, where *ds*■ is the result of the **full_join** of the first "i" operands and *ds*■■■ is the i+1th operand. For each pair <*ds*■*, ds*■■■>, the resulting Data Set is fed with the Data Points that match in *ds*■ and *ds*■■■ or that are only in *ds*■ or in *ds*■■■. If for a Data Point in ds■ the join succeeds, the values for the Measures and the Attributes are carried from *ds*■ and *ds*■■■ as explained. Otherwise, i.e., if no Data Point in *ds*■■■ matches the Data Point in *ds*■, NULL values are given to Measures and Attributes coming only from *ds*■■■. Symmetrically, if no Data Point in *ds*■ matches the Data Point in *ds*■■■, NULL values are given to Measures and Attributes coming only from *ds*■. The constraints described above guarantee the absence of NULL values on the Identifier Components. As mentioned, the **using** clause is not allowed in this case.

- **cross_join**: the join is performed stepwise, between consecutive pairs of input Data Sets, starting from the left side and proceeding toward the right side. No match is performed but the Cartesian product of the input Data Points is generated in output. These stepwise operations are associative. More formally, consider the ordered pair <*ds*■*, ds*■■■>, where *ds*■ is the result of the *cross_join* of the first "i" operands and *ds*■■■ is the i+1-th operand. For each pair <*ds*■*, ds*■■■>, the resulting Data Set is fed with the Data Points obtained as the Cartesian product between the Data Points of *ds*■ and *ds*■■■. The resulting Data Set will have all the Components from *ds*■ and *ds*■■■. For the Data Sets which have at least one Component in common, the alias parameter is mandatory. As mentioned, the **using** parameter is not allowed in this case.

The **semantics of the clauses** is the following.

- **filter** takes as input a Boolean Component expression (having type *component*<*boolean*>). This clause filters in or out the input Data Points; when the expression is TRUE the Data Point is kept, otherwise it is not kept in the result. Only one **filter** clause is allowed.
- **apply** combines the homonym Measures in the source operands whose type is compatible with the operators used in *applyExpr*, generating homonym Measures in the output. The expression *applyExpr* can use as input the names or aliases of the operand Data Sets. It applies the expression to all the n-uples of homonym Measures in the input Data Sets producing in the target a single homonym Measure for each n-uple. It can be thought of as the multi-measure version of the **calc**. For example, if the following aliases have been declared: *d1*, *d2*, *d3*, then the following expression *d1+d2+d3*, sums all the homonym Measures in the three input Data Sets, say *M1* and *M2*, so as to obtain in the result: *M1 := d1#M1 + d2#M1 + d3#M1 and M2 := d1#M2 + d2#M2 + d3#M2*. It is not only a compact version of a multiple **calc**, but also essential when the number of Measures in the input operands is not known beforehand. Only one **apply** clause is allowed.
- **calc** calculates new Identifier, Measure or Attribute Components on the basis of sub-expressions at Component level. Each Component is calculated through an independent sub-expression. It is possible to specify the role of the calculated Component among **measure**, **identifier**, **attribute** or **viral attribute**, therefore the **calc** clause can be used also to change the role of a Component when possible. The keyword **viral** allows controlling the virality of Attributes (for the Attribute propagation rule see the User Manual). The following rule is used when the role is omitted: if the component exists in the operand Data Set then it maintains that role; if the component does not exist in the operand Data Set then the role is **measure**. The *calcExpr* are independent one another, they can only reference Components of the input Virtual Data Set and cannot use Components generated, for example, by other *calcExpr*. If the calculated Component is a new Component, it is added to the output virtual Data Set. If the Calculated component is a Measure or an Attribute that already exists in the input virtual Data Set, the calculated values overwrite the original values. If the Calculated component is an Identifier that already exists in the input virtual Data Set, an exception is raised because overwriting an Identifier Component is forbidden for preserving the functional behaviour. Analytic operators can be used in the **calc** clause.
- **aggr** calculates aggregations of dependent Components (Measures or Attributes) on the basis of sub-expressions at Component level. Each Component is calculated through an independent sub-expression. It is possible to specify the role of the calculated Component among **measure**, **identifier**, **attribute**, or **viral attribute**. The substring **viral** allows to control the virality of Attributes, if the Attribute propagation rule is adopted (see the User Manual). The **aggr** sub-expressions are independent of one another, they can only reference Components of the input Virtual Data Set and cannot use Components generated, for example, by other **aggr** sub-expressions. The **aggr** computed Measures and Attributes are the only Measures and Attributes returned in the output virtual Data Set (plus the possible viral Attributes, see below **Attribute propagation**). The sub-expressions must contain only Aggregate operators, which are able to compute an aggregated Value relevant to a group of Data Points. The groups of Data Points to be aggregated are specified through the *groupingClause*, which allows the following alternative options. | **group by**: the Data Points are grouped by the values of the specified Identifier. The Identifiers not specified are dropped in the result. | **group except**: the Data Points are grouped by the values of the Identifiers not specified in the clause. The specified

Identifiers are dropped in the result. | **group all**: converts an Identifier Component using *conversionExpr* and keeps all the resulting Identifiers. | The **having** clause is used to filter groups in the result by means of an aggregate condition evaluated on the single groups, for example the minimum number of rows in the group. If no grouping clause is specified, then all the input Data Points are aggregated in a single group and the clause returns a Data Set that contains a single Data Point and has no Identifier Components.

- **keep** maintains in the output only the specified dependent Components (Measures and Attributes) of the input virtual Data Set and drops the non-specified ones. It has the role of a projection in the usual relational semantics (specifying which columns have to be projected in). Only one **keep** clause is allowed. If **keep** is used, **drop** must be omitted.
- **drop** maintains in the output only the non-specified dependent Components (Measures and Attributes) of the input virtual Data Set (component<scalar>) and drops the specified ones. It has the role of a projection in the usual relational join semantics (specifying which columns will be projected out). Only one **drop** clause is allowed. If **drop** is used, **keep** must be omitted.
- **rename** assigns new names to one or more Components (Identifier, Measure or Attribute Components). The resulting Data Set, after renaming all the specified Components, must have unique names of all its Components (otherwise a runtime error is raised). Only the Component name is changed and not the Component Values, therefore the new Component must be defined on the same Value Domain and Value Domain Subset as the original Component (see also the IM in the User Manual). If the name of a Component defined on a different Value Domain or Set is assigned, an error is raised. In other words, rename is a transformation of the variable without any change in its values.

The semantics of the **Attribute propagation** in the join is the following. The Attributes calculated through the **calc** or **aggr** clauses are maintained unchanged. For all the other Attributes that are defined as **viral**, the Attribute propagation rule is applied (for the semantics, see the Attribute Propagation Rule section in the User the Manual). This is done before the application of the **drop**, **keep** and **rename** clauses, which acts also on the Attributes resulting from the propagation.

The semantics of the **final automatic aliases** removal is the following. After the application of all the clauses, the structure of the final virtual Data Set is further modified. All the Components of the form “alias#component_name” (or “dataset_name#component_name”) are implicitly renamed into “component_name”. This means that the prefixes in the Component names are automatically removed. It is responsibility of the user to guarantee the absence of duplicated Component names once the prefixes are removed. In other words, the user must ensure that there are no pairs of Components whose names are of the form “alias1#c1” and “alias2#c1” in the structure of the virtual Data Point, since the removal of “alias1” and “alias2” would cause the clash. If, after the aliases removal two Components have the same name, an error is raised. In particular, name conflicts may derive if the using clause is present and some homonym Identifier Components do not appear in it; these components should be properly renamed because cannot be removed; the input Data Set have homonym Measures and there is no apply clause which unifies them; these Measures can be renamed or removed.

Examples

Given the operand datasets DS_1 and DS_2, DS_3 and that || is the string concatenation:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1	Me_2
1	A	A	B
1	B	C	D
2	A	E	F

Input **DS_2** (see structure)

Id_1	Id_2	Me_1A	Me_2
1	A	B	Q
1	B	S	T
3	A	Z	M

Input **DS_3** (see structure)

Id_1	Id_2	Me_1	Me_2
1	A	B	Q
1	B	S	T
3	A	Z	M

Example 1

```
DS_r := inner_join (DS_1 as d1, DS_2 as d2 keep Me_1, d2#Me_2, Me_1A );
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2	Me_1A
1	A	A	Q	B
1	B	C	T	S

Example 2

```
DS_r := left_join ( DS_1 as d1, DS_2 as d2 keep Me_1, d2#Me_2, Me_1A );
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2	Me_1A
1	A	A	Q	B
1	B	C	T	S
2	A	E		

Example 3

```
DS_r := full_join ( DS_1 as d1, DS_2 as d2 keep Me_1, d2#Me_2, Me_1A );
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2	Me_1A
1	A	A	Q	B
1	B	C	T	S
2	A	E		
3	A		M	Z

Example 4

```
DS_r := cross_join (DS_1 as d1, DS_2 as d2 rename d1#Id_1 to Id11, d1#Id_2 to Id12, d2#Id_1
```

results in (see structure):

DS_r

Id11	Id12	Id21	Id22	Me_1	Me12	Me_1A	Me_2
1	A	1	A	A	B	B	Q
1	A	1	B	A	B	S	T
1	A	3	A	A	B	Z	M
1	B	1	A	C	D	B	Q
1	B	1	B	C	D	S	T
1	B	3	A	C	D	Z	M
2	A	1	A	E	F	B	Q
2	A	1	B	E	F	S	T
2	A	3	A	E	F	Z	M

Example 5

```
DS_r := inner_join (DS_1 as d1, DS_2 as d2 filter Me_1 = "A" calc Me_4 := Me_1 || Me_1A drop
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_1A	Me_2	Me_4
1	A	A	B	Q	AB

Example 6

```
DS_r := inner_join ( DS_1  filter Id_2 ="B" calc Me_2 := Me_2 || "_NEW" keep Me_1, Me_2);
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
1	B	C	D_NEW

Example 7

```
DS_r := inner_join (DS_1 as d1, DS_3 as d2 apply d1 || d2);
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
1	A	AB	BQ
1	B	CS	DT

VTL-ML - String Operators

String concatenation: \|

Syntax

op1 **||** op2

Input parameters

op1, op2	the operands

Semantics for scalar operations

Concatenates two strings. For example, `"Hello" || ", World"` gives `"Hello, World"`.

Input parameters type

op1, op2

```
dataset { measure<string> _+ }
| component<string>
| string
```

Result type

result

```
dataset { measure<string> _+ }
| component<string>
| string
```

Additional Constraints

None.

Behavior

The operator has the behaviour of the “Operators applicable on two Scalar Values or Data Sets or Data Set Components” (see the section “Typical behaviours of the ML Operators”).

Examples

Given the operand datasets DS_1 and DS_2:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1
1	A	hello
2	B	hi

Input **DS_2** (see structure)

Id_1	Id_2	Me_1
1	A	world
2	B	there

Example 1

```
DS_r := DS_1 || DS_2;
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1
1	A	helloworld
2	B	hithere

Example 2

```
DS_r := DS_1[calc Me_2 := Me_1 || " world"];
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
1	A	hello	hello world
2	B	hi	hi world

Whitespace removal: *trim*, *rtrim*, *ltrim*

Syntax

{ **trim** | **ltrim** | **rtrim** }[1] **(** op **)**

Input parameters

op	the operand

Semantics for scalar operations

Removes trailing or/and leading whitespace from a string. For example, `trim("Hello ")` gives `"Hello"`.

Input parameters type

op1

```
dataset { measure<string> _+ }
| component<string>
| string
```

Result type

result

```
dataset { measure<string> _+ }
| component<string>
| string
```

Additional Constraints

None.

Behavior

The operator has the behaviour of the “Operators applicable on one Scalar Value or Data Set or Data Set Component” (see the section “Typical behaviours of the ML Operators”).

Examples

Given the operand dataset DS_1 (note that the input data have a whitesapce at the end of the string, which may not be visualised):

Input **DS_1** (see structure)

Id_1	Id_2	Me_1
1	A	hello
2	B	hi

Example 1

```
DS_r := rtrim(DS_1);
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1
1	A	hello
2	B	hi

Example 2

```
DS_r := DS_1[ calc Me_2:= rtrim(Me_1)];
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
1	A	hello	hello
2	B	hi	hi

Character case conversion: *upper/lower*

Syntax

{ **upper** | **lower** }[1] **(** op **)**

Input parameters

op	the operand

Examples of valid syntaxes

```
upper("Hello")
lower(ds_1)
```

Semantics for scalar operations

Converts the character case of a string in upper or lower case. For example, `upper("Hello")` gives `"HELLO"`.

Input parameters type

op

```
dataset { measure<string> _+ }
| component<string>
| string
```

Result type

result

```
dataset { measure<string> _+ }
| component<string>
| string
```

Additional Constraints

None.

Behavior

The operator has the behaviour of the “Operators applicable on one Scalar Value or Data Set or Data Set Component” (see the section “Typical behaviours of the ML Operators”).

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1
1	A	hello
2	B	hi

Example 1

```
DS_r := upper(DS_1);
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1
1	A	HELLO
2	B	HI

Example 2

```
DS_r := DS_1[calc Me_2:= upper(Me_1)];
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2

1	A	hello	HELLO
2	B	hi	HI

Sub-string extraction: *substr*

Syntax

substr (op, *start, length* **)**

Input parameters

op	the operand
start	the starting digit (first character) of the string to be extracted
length	the length (number of characters) of the string to be extracted

Examples of valid syntaxes

```
substr  ( DS_1,  2 ,  3 )
substr  ( DS_1, 2 )
substr  ( DS_1, _ , 3 )
substr  ( DS_1 )
```

Semantics for scalar operations

The operator extracts a substring from op, which must be *string type*. The substring starts from the startth character of the input string and has a number of characters equal to the **length** parameter.

- If *start* is omitted, the substring starts from the 1st position.
- If *length* is omitted or overcomes the length of the input string, the substring ends at the end of the input string.
- If *start* is greater than the length of the input string, an empty string is extracted.

For example:

`substr (“abcdefghijklmnopqrstuvwxyz”, 5 , 10 )` gives: “efghijklmn”
`substr (“abcdefghijklmnopqrstuvwxyz”, 25 , 10 )` gives: “yz”
`substr (“abcdefghijklmnopqrstuvwxyz”, 30 , 10 )` gives: “”

Input parameters type

op

```
dataset { measure<string> _+ }
| component<string>
| string
```

start

```
component < integer [ value >= 1 ] >
| integer [ value >= 1 ]
```

length

```
component < integer [ value >= 0 ] >
| integer [ value >= 0 ]
```

Result type

result

```
dataset { measure<string> _+ }
| component<string>
| string
```

Additional Constraints

None.

Behavior

As for the invocations at Data Set level, the operator has the behaviour of the “Operators applicable on one Scalar Value or Data Set or Data Set Component”. As for the invocations at Component or Scalar level, the operator has the behaviour of the “Operators applicable on more than two Scalar Values or Data Set Components” (see the section “Typical behaviours of the ML Operators”).

Examples

Given the operand datasets DS_1 and DS_2:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1	Me_2
1	A	hello world	medium size text
1	B	abcdefghilmno	short text
2	A	pqrstuvwxyz	this is a long description

Example 1

```
DS_r:= substr ( DS_1 , 7 );
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
1	A	world	size text
1	B	ghilmno	text
2	A	vwxyz	s a long description

Example 2

```
DS_r:= substr ( DS_1 , 1 , 5 );
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
1	A	hello	mediu
1	B	abcde	short
2	A	pqrst	this

Example 3

```
DS_r:= DS_1 [ calc Me_2:= substr ( Me_2 , 1 , 5 ) ];
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
1	A	hello world	mediu
1	B	abcdefghilmno	short
2	A	pqrstuvwxyz	this

String pattern replacement: *replace*

Syntax

replace (op , pattern1, *pattern2* **)**

Input parameters

op	the operand
pattern1	the pattern to be replaced
pattern2	the replacing pattern

Examples of valid syntaxes

```
replace(DS_1, "Hello", "Hi")
replace(DS_1, "Hello")
```

Semantics for scalar operations

Replaces all the occurrences of a specified string-pattern (*pattern1*) with another one (*pattern2*). If *pattern2* is omitted then all occurrences of *pattern1* are removed. For example:

`replace("Hello world", "Hello", "Hi")` gives `"Hi world"`
`replace("Hello world", "Hello")` gives `" world"`
`replace ("Hello", "ello", "i")` gives `"Hi"`

Input parameters type

op

```
dataset { measure<string> _+ }
| component<string>
| string
```

pattern1, pattern2

```
component<string>
| string
```

Result type

result

```
dataset { measure<string> _+ }
| component<string>
| string
```

Additional Constraints

The second parameter (*pattern1*) cannot be omitted.

Behaviour

As for the invocations at Data Set level, the operator has the behaviour of the “Operators applicable on one Scalar Value or Data Set or Data Set Component”. As for the invocations at Component or Scalar level, the operator has the behaviour of the “Operators applicable on more than two Scalar Values or Data Set Components” (see the section “Typical behaviours of the ML Operators”).

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1
1	A	hello world
2	A	say hello
3	A	he
4	A	hello!

Example 1

```
DS_r := replace (DS_1,"ello","i");
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1
1	A	hi world
2	A	say hi
3	A	he
4	A	hi!

Example 2

```
DS_r := DS_1[ calc Me_2:= replace (Me_1,"ello","i")];
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
1	A	hello world	hi world
2	A	say hello	say hi
3	A	he	he

4	A	hello!	hi!

String pattern location: *instr*

Syntax

instr (op, pattern, *start, occurrence* **)**

Input parameters

op	the operand
pattern	the string-pattern to be searched
start	the position in the input string of the character from which the search starts
occurrence	the occurrence of the pattern to search

Examplesof valid syntaxes

```
instr ( DS_1,  “ab”, 2 , 3 )
instr ( DS_1,  “ab”, 2 )
instr ( DS_1,  “ab”, _ , 2 )
instr ( DS_1,  “ab” )
```

Semantics for scalar operations

The operator returns the position in the input string of a specified string (*pattern*). The search starts from the *start*th character of the input string and finds the nth occurrence of the pattern, returning the position of its first character. If *start* is omitted, the search starts from the 1st position. If nth *occurrence* is omitted, the value is 1. If the nth *occurrence* of the string-pattern after the *start* th character is not found in the input string, the returned value is 0. For example:

`instr ("abcde", "c" )` gives `3`
`instr ("abcdecfrxcwsd", "c", _ , 3 )` gives `10`
`instr ("abcdecfrxcwsd", "c", 5 , 3 )` gives `0`

Input parameters type

op

```
dataset { measure<string> _+ }
| component<string>
| string
```

pattern

```
component<string>
| string
```

start

```
component < integer [ value >= 1 ] >
| integer [ value >= 1 ]
```

occurrence

```
component < integer [ value >= 1 ] >
| integer [ value >= 1 ]
```

Result type

result

```
dataset { measure<integer[value >= 0]> int_var }
| component<integer[value >= 0]>
| integer[value >= 0]
```

Additional Constraints

The second parameter (pattern) cannot be omitted. For operations at Data Set level, the input Data Set must have exactly one *string* type Measure.

Behaviour

As for the invocations at Data Set level, the operator has the behaviour of the “Operators applicable on one Scalar Value or Data Set or Data Set Component”, as for the invocations at Component or Scalar level, the operator has the behaviour of the “Operators applicable on more than two Scalar Values or Data Set Components”, (see the section “Typical behaviours of the ML Operators”). If *op* is a Data Set then **instr** returns a dataset with a single measure *int_var* of type *integer*.

Examples

Given the operand datasets DS_1 and DS_2:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1
1	A	hello world
2	A	say hello
3	A	he
4	A	hi, hello!

Input **DS_2** (see structure)

Id_1	Id_2	Me_1	Me_2
1	A	hello	world
2	B		hi

Example 1

```
DS_r:= instr(DS_1,"hello");
```

results in (see structure):

DS_r

Id_1	Id_2	int_var
1	A	1
2	A	5
3	A	0
4	A	5

Example 2

```
DS_r := DS_1[calc Me_2:=instr(Me_1,"hello")];
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
1	A	hello world	1
2	A	say hello	5
3	A	he	0
4	A	hi, hello!	5

Example 3

```
DS_r := DS_1 [calc Me_10:= instr(Me_1, "o" ), Me_20:=instr(Me_2, "o")];
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2	Me_10	Me_20
1	A	hello	world	5	2
2	B		hi		0

Example 4

Applying the instr operator at Data Set level to a multi Measure Data Set:

DS_r := instr(DS_2, “o”) would give error because DS_2 has more than one Measure.

String length: *length*

Syntax

length (op **)**

Input parameters

op	the operand

Examples of valid syntaxes

```
length("Hello, World!")
length(DS_1)
```

Semantics for scalar operations

Returns the length of a string. For example:

`length("Hello, World!")` gives `13`
`length("")` gives `0`

Input parameters type

op

```
dataset { measure<string> _+ }
| component<string>
| string
```

Result type

result

```
dataset { measure<integer[value >= 0]> int_var }
| component<integer[value >= 0]>
| integer[value >= 0]
```

Additional Constraints

For operations at Data Set level, the input Data Set must have exactly one *string* type Measure.

Behaviour

The operator has the behaviour of the “Operators changing the data type” (see the section “Typical behaviours of the ML Operators”). If *op* is a Data Set then **length** returns a dataset with a single measure int_var of type *integer*.

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1
1	A	hello
2	B	

Input **DS_2** (see structure)

Id_1	Id_2	Me_1	Me_2
1	A	hello	world
2	B		hi

Example 1

```
DS_r := length(DS_1);
```

results in (see structure):

DS_r

Id_1	Id_2	int_var
1	A	5
2	B	0

Example 2

```
DS_r:= DS_1[calc Me_2:=length(Me_1)];
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2

1	A	hello	5
2	B		0

Example 3

```
DS_r := DS_2 [calc Me_10:= length(Me_1), Me_20:=length(Me_2)];
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2	Me_10	Me_20
1	A	hello	world	5	5
2	B		hi	0	2

Example 4

length operator applied at Data Set level to a multi Measure Data Set:

```
DS_r := length(DS_2)
```

would give error because DS_2 has more than one Measure.

VTL-ML - Numeric Operators

Unary Plus: +

Syntax

+ op

Input parameters

op	the operand

Examples of valid syntaxes

```
+ DS_1
+ 3
```

Semantics for scalar operations

The operator **+** returns the operand unchanged. For example:

+3 gives 3
`+(-5)` gives `-5`

Input parameters type

op:

```
dataset { measure<number> _+ }
| component<number>
| number
```

Result type

result:

```
dataset { measure<number> _+ }
| component<number>
| number
```

Additional Constraints

None.

Behaviour

The operator has the behaviour of the “Operators applicable on one Scalar Value or Data Set or Data Set Component” (see the section “Typical behaviours of the ML Operators”). According to the general rules about data types, the operator can be applied also on sub-types of *number*, that is the type *integer*. If the type of the operand is *integer* then the result has type *integer*. If the type of the operand is *number* then the result has type *number*.

Examples

Given the operand Data Set DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1	Me_2
10	A	1.0	5
10	B	2.3	10
11	A	3.2	12

Example 1

```
DS_r := + DS_1;
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
10	A	1.0	5
10	B	2.3	10
11	A	3.2	12

Example 2

```
DS_r := DS_1 [ calc Me_3 := + Me_1 ];
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2	Me_3
10	A	1.0	5	1.0
10	B	2.3	10	2.3
11	A	3.2	12	3.2

Unary Minus: -

Syntax

- op

Input parameters

op	the operand

Examples of valid syntaxes

```
-DS_1
-3
```

Semantics for scalar operations

The operator **-** inverts the sign of *op*. For example:

`-3` gives `-3`
`-(-5)` gives `5`

Input parameters type

op:

```
dataset { measure<number> _+ }
| component<number>
| number
```

Result type

result:

```
dataset { measure<number> _+ }
| component<number>
| number
```

Additional Constraints

None.

Behaviour

The operator has the behaviour of the “Operators applicable on one Scalar Value or Data Set or Data Set Component” (see the section “Typical behaviours of the ML Operators”). According to the general rules about data types, the operator can be applied also on sub-types of *number*, that is the type *integer*. If the type of the operand is *integer* then the result has type *integer*. If the type of the operand is *number* then the result has type *number*.

Examples

Given the operand Data Set DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1	Me_2
10	A	1	5.0
10	B	2	10.0
11	A	3	12.0

Example 1

```
DS_r := - DS_1;
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
10	A	-1	-5.0
10	B	-2	-10.0
11	A	-3	-12.0

Example 2

```
DS_r := DS_1 [ calc Me_3 := - Me_1 ];
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2	Me_3
10	A	1	-5.0	-1
10	B	2	-10.0	-2
11	A	3	-12.0	-3

Addition: +

Syntax

op1 **+** op2

Input parameters

op1	the first addendum
op2	the second addendum

examples of valid syntaxes

```
DS_1 + DS_2
3 + 5
```

Semantics for scalar operations

The operator addition returns the sum of two numbers. For example:

`3 + 5` gives 8

Input parameters type

op1, op2

```
dataset { measure<number> _+ }
| component<number>
| number
```

Result type

result

```
dataset { measure<number> _+ }
| component<number>
| number
```

Additional Constraints

None.

Behaviour

The operator has the behaviour of the “Operators applicable on two Scalar Values or Data Sets or Data Set Components” (see the section “Typical behaviours of the ML Operators”). According to the general rules about data types, the operator can be applied also on sub-types of *number*, that is the type *integer*. If the type of both operands is *integer* then the result has type *integer*. If one of the operands is of type *number*, then the other operand is implicitly cast to *number* and therefore the result has type *number*.

Examples

Given the operand datasets DS_1 and DS_2:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1	Me_2
10	A	5	5.0
10	B	2	10.5
11	A	3	12.2
11	B	4	20.3

Input **DS_2** (see structure)

Id_1	Id_2	Me_1	Me_2
10	A	10	3.0
10	C	11	6.2
11	B	6	7.0

Example 1

```
DS_r := DS_1 + DS_2;
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
10	A	15	8.0
11	B	10	27.3

Example 2

```
DS_r := DS_1 + 3;
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
10	A	8	8.0
10	B	5	13.5
11	A	6	15.2
11	B	7	23.3

Example 3

```
DS_r := DS_1 [ calc Me_3 := Me_1 + 3.0 ];
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2	Me_3
10	A	5	5.0	8.0
10	B	2	10.5	5.0
11	A	3	12.2	6.0
11	B	4	20.3	7.0

Subtraction: -

Syntax

op1 **-** op2

Input parameters

op1	the minuend
op2	the subtrahend

Examples of valid syntaxes

```
DS_1 - DS_2
3 - 5
```

Semantics for scalar operations

The operator subtraction returns the difference of two numbers. For example:

`3 - 5` gives `-2`

Input parameters type

op1, op2

```
dataset { measure<number> _+ }
| component<number>
| number
```

Result type

result

```
dataset { measure<number> _+ }
| component<number>
| number
```

Additional Constraints

None.

Behaviour

The operator has the behaviour of the “Operators applicable on two Scalar Values or Data Sets or Data Set Components” (see the section “Typical behaviours of the ML Operators”). According to the general rules about data types, the operator can be applied also on sub-types of *number*, that is the type *integer*. If the type of both operands is *integer* then the result has type *integer*. If one of the operands is of type *number*, then the other operand is implicitly cast to *number* and therefore the result has type *number*.

Examples

Given the operand datasets DS_1 and DS_2:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1	Me_2
10	A	5	5.0
10	B	2	10.5
11	A	3	12.2
11	B	4	20.3

Input **DS_2** (see structure)

Id_1	Id_2	Me_1	Me_2
10	A	10	3.0
10	C	11	6.2
11	B	6	7.0

Example 1

```
DS_r := DS_1 - DS_2;
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
10	A	-5	2.0
11	B	-2	13.3

Example 2

```
DS_r := DS_1 - 3;
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
10	A	2	2.0
10	B	-1	7.5
11	A	0	9.2
11	B	1	17.3

Example 3

```
DS_r := DS_1 [ calc Me_3 := Me_1 - 3 ];
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2	Me_3
10	A	5	5.0	2
10	B	2	10.5	-1
11	A	3	12.2	0
11	B	4	20.3	1

Multiplication: *

Syntax

op1 ***** op2

Input parameters

op1	the multiplicand
op2	the multiplier

Examples of valid syntaxes

```
DS_1 * DS_2
3 * 5
```

Semantics for scalar operations

The operator addition returns the product of two numbers. For example:

`3 * 5` gives 15

Input parameters type

op1, op2

```
dataset { measure<number> _+ }
| component<number>
| number
```

Result type

result

```
dataset { measure<number> _+ }
| component<number>
| number
```

Additional Constraints

None.

Behavior

The operator has the behaviour of the “Operators applicable on two Scalar Values or Data Sets or Data Set Components” (see the section “Typical behaviours of the ML Operators”). According to the general rules about data types, the operator can be applied also on sub-types of *number*, that is the type *integer*. If the type of both operands is *integer* then the result has type *integer*. If one of the operands is of type *number*, then the other operand is implicitly cast to *number* and therefore the result has type *number*.

Examples

Given the operand datasets DS_1 and DS_2:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1	Me_2
10	A	100	7.6
10	B	10	12.3
11	A	20	25.0
11	B	2	20.0

Input **DS_2** (see structure)

Id_1	Id_2	Me_1	Me_2
10	A	1	2.0
10	C	5	3.0
11	B	2	1.0

Example 1

```
DS_r := DS_1 * DS_2;
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
10	A	100	15.2
11	B	4	20.0

Example 2

```
DS_r := DS_1 * -3;
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
10	A	-300	-22.8
10	B	-30	-36.9
11	A	-60	-75.0
11	B	-6	-60.0

Example 3

```
DS_r := DS_1 [ calc Me_3 := Me_1 * Me_2 ];
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2	Me_3
10	A	100	7.6	760.0
10	B	10	12.3	123.0
11	A	20	25.0	500.0
11	B	2	20.0	40.0

Division: /

Syntax

op1 **/** op2

Input parameters

op1	the dividend
op2	the divisor

Examples of valid syntaxes

```
DS_1 / DS_2
3 / 5
```

Semantics for scalar operations

The operator addition divides two numbers. For example: | `3 / 5` gives `0.6`

Input parameters type

op1, op2

```
dataset { measure<number> _+ }
| component<number>
| number
```

Result type

result

```
dataset { measure<number> _+ }
| component<number>
| number
```

Additional Constraints

None.

Behavior

The operator has the behaviour of the “Operators applicable on two Scalar Values or Data Sets or Data Set Components” (see the section “Typical behaviours of the ML Operators”). According to the general rules about data types, the operator can be applied also on sub-types of *number*, that is the type *integer*. The result has type *number*.

If *op2* is 0 then the operation generates a run-time error.

Examples

Given the operand datasets DS_1, DS_2 and DS_3:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1
10	A	7.6
10	B	12.3
11	A	25.0
11	B	20.0

Input **DS_2** (see structure)

Id_1	Id_2	Me_1
10	A	2.0
10	C	3.0
11	B	1.0

Input **DS_3** (see structure)

Id_1	Id_2	Me_1	Me_2
10	A	100	7.6
10	B	10	12.3
11	A	20	25.0
11	B	2	20.0

Example 1

```
DS_r := DS_1 / DS_2;
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1
10	A	3.8
11	B	20.0

Example 2

```
DS_r := DS_1 / 10;
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1
10	A	0.76
10	B	1.23
11	A	2.5
11	B	2.0

Example 3

```
DS_r := DS_3 [ calc Me_3 := Me_2 / Me_1 ];
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2	Me_3
10	A	100	7.6	0.076
10	B	10	12.3	1.23
11	A	20	25.0	1.25
11	B	2	20.0	10.0

Modulo: *mod*

Syntax

mod (op1 , op2 **)**

Input parameters

op1	the dividend
op2	the divisor

Examples of valid syntaxes

```
mod( DS_1, DS_2 )
mod ( DS_1, 5 )
mod ( 5, DS_2 )
mod ( 5, 2 )
```

Semantics for scalar operations

The operator **mod** returns the remainder of *op1* divided by *op2*. It returns *op1* if divisor *op2* is 0. For example: | `mod ( 5, 2 )` gives `1` | `mod ( 5, -2 )` gives `-1` | `mod ( 8, 2 )` gives `0` | `mod ( 9, 0 )` gives `9`

Input parameters type

op1, op2

```
dataset { measure<number> _+ }
| component<number>
| number
```

Result type

result

```
dataset { measure<number> _+ }
| component<number>
| number
```

Additional Constraints

None.

Behavior

The operator has the behaviour of the “Operators applicable on two Scalar Values or Data Sets or Data Set Components” (see the section “Typical behaviours of the ML Operators”).

According to the general rules about data types, the operator can be applied also on sub-types of *number*, that is the type *integer*. If the type of both operands is *integer* then the result has type *integer*. If one of the operands is of type *number*, then the other operand is implicitly cast to *number* and therefore the result has type *number*.

Examples

Given the operand datasets DS_1 and DS_2:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1	Me_2
10	A	100	0.7545
10	B	10	18.45
11	A	20	1.87
11	B	9	20.3

Input **DS_2** (see structure)

Id_1	Id_2	Me_1	Me_2
10	A	1	0.25
10	C	5	3.0
11	B	2	2.0

Example 1

```
DS_r := mod ( DS_1, DS_2 );
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
10	A	0	0.0045
11	B	1	0.3

Example 2

```
DS_r := mod ( DS_1, 15 );
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
10	A	10	0.7545
10	B	10	3.45
11	A	5	1.87
11	B	9	12.3

Example 3

```
DS_r := DS_1[ calc Me_3 := mod( Me_1, 3.0 ) ];
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2	Me_3
10	A	100	0.7545	1.0
10	B	10	18.45	1.0
11	A	20	1.87	2.0
11	B	9	12.3	0.0

Rounding: *round*

Syntax

round (op , *numDigit* **)**

Input parameters

op	the operand
numDigit	the number of positions to round to

Examples of valid syntaxes

```
round ( DS_1 ,  2 )
round ( DS_2 )
round ( 3.14159 ,  2 )
round ( 3.14159 ,  _ )
```

Semantics for scalar operations

The operator **round** rounds the operand to a number of positions at the right of the decimal point equal to the *numDigit* parameter. The decimal point is assumed to be at position 0. If *numDigit* is negative, the rounding happens at the left of the decimal point. The rounding operation leaves the *numDigit* position unchanged if the *numDigit*+1 position is between 0 and 4, otherwise it adds 1 to the number that is in the *numDigit* position. All the positions greater than *numDigit* are set to 0. The basic scalar type of the result is *integer* if *numDigit* is omitted, *number* otherwise. For example:

`round ( 3.14159, 2 )` gives `3.14`
`round ( 3.14159, 4 )` gives `3.1416`
`round ( 12345.6, 0 )` gives `12346.0`
`round ( 12345.6 )` gives `12346`
`round ( 12345.6, _ )` gives `12346`
`round ( 12345.6, -1 )` gives `12350.0`

Input parameters type

op

```
dataset { measure<number> _+ }
| component<number>
| number
```

numDigit

```
component<integer>
| integer
```

Result type

result

```
dataset { measure<number> _+ }
| component<number>
| number
```

Additional Constraints

None.

Behavior

As for the invocations at Data Set level, the operator has the behaviour of the “Operators applicable on one Scalar Value or Data Set or Data Set Component”. As for the invocations at Component or Scalar level, the operator has the behaviour of the “Operators applicable on two Scalar Values or Data Sets or Data Set Components”, (see the section “Typical behaviours of the ML Operators”).

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1	Me_2
10	A	7.5	5.9
10	B	7.1	5.5
11	A	36.2	17.7
11	B	44.5	24.3

Example 1

```
DS_r := round(DS_1, 0);
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
10	A	8.0	6.0
10	B	7.0	6.0
11	A	36.0	18.0
11	B	45.0	24.0

Example 2

```
DS_r := DS_1 [ calc Me_10:= round( Me_1 ) ];
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2	Me_10
10	A	7.5	5.9	8
10	B	7.1	5.5	7
11	A	36.2	17.7	36
11	B	44.5	24.3	45

Example 3

```
DS_r := DS_1 [ calc Me_20:= round( Me_1 , -1 ) ];
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2	Me_20
10	A	7.5	5.9	10
10	B	7.1	5.5	10
11	A	36.2	17.7	40
11	B	44.5	24.3	40

Truncation: *trunc*

Syntax

trunc (op , *numDigit* **)**

Input parameters

op	the operand
numDigit	the number of position from which to trunc

Examples of valid syntaxes

```
trunc ( DS_1 ,  2 )
trunc ( DS_2 )
trunc ( 3.14159 ,  2 )
trunc ( 3.14159 ,  _ )
```

Semantics for scalar operations

The operator **trunc** truncates the operand to a number of positions at the right of the decimal point equal to the *numDigit* parameter. The decimal point is assumed to be at position 0. If *numDigit* is negative, the truncation happens at the left of the decimal point. The truncation operation leaves the *numDigit* position unchanged. All the positions greater than *numDigit* are eliminated. The basic scalar type of the result is *integer* if *numDigit* is omitted, *number* otherwise. For example:

`trunc ( 3.14159, 2 )` gives `3.14`
`trunc ( 3.14159, 4 )` gives `3.1415`
`trunc ( 12345.6, 0 )` gives `12345.0`
`trunc ( 12345.6 )` gives `12345`
`trunc ( 12345.6, _ )` gives `12345`
`trunc( 12345.6, -1 )` gives `12340.0`

Input parameters type

op

```
dataset { measure<number> _+ }
| component<number>
| number
```

numDigit

```
component<integer>
| integer
```

Result type

result

```
dataset { measure<number> _+ }
| component<number>
| number
```

Additional Constraints

None.

Behavior

As for the invocations at Data Set level, the operator has the behaviour of the “Operators applicable on one Scalar Value or Data Set or Data Set Component”, as for the invocations at Component or Scalar level, the operator has the behaviour of the “Operators applicable on two Scalar Values or Data Sets or Data Set Components”, (see the section “Typical behaviours of the ML Operators”).

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1	Me_2
10	A	7.5	5.9
10	B	7.1	5.5
11	A	36.2	17.7
11	B	44.5	24.3

Example 1

```
DS_r := trunc(DS_1, 0);
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
10	A	7.0	5.0
10	B	7.0	5.0
11	A	36.0	17.0
11	B	44.0	24.0

Example 2

```
DS_r := DS_1[ calc Me_10:= trunc( Me_1 ) ];
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2	Me_10
10	A	7.5	5.9	7
10	B	7.1	5.5	7
11	A	36.2	17.7	36
11	B	44.5	24.3	44

Example 3

```
DS_r := DS_1[ calc Me_20:= trunc( Me_1 , -1 ) ];
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2	Me_20
10	A	7.5	5.9	0.0
10	B	7.1	5.5	0.0
11	A	36.2	17.7	30.0
11	B	44.5	24.3	40.0

Ceiling: *ceil*

Syntax

ceil (op **)**

Input parameters

op	the operand

Examples of valid syntaxes

```
ceil ( DS_1 )
ceil ( 3.14159 )
```

Semantics for scalar operations

The operator **ceil** returns the smallest integer greater than or equal to *op*. For example:

`ceil( 3.14159 )` gives `4`
`ceil( 15 )` gives `15`
`ceil( -3.1415 )` gives `-3`
`ceil( -0.1415 )` gives `0`

Input parameters type

op

```
dataset { measure<number> _+ }
| component<number>
| number
```

Result type

result

```
dataset { measure<integer> _+ }
| component<integer>
| integer
```

Additional Constraints

None.

Behaviour

The operator has the behaviour of the “Operators applicable on one Scalar Value or Data Set or Data Set Component” (see the section “Typical behaviours of the ML Operators”).

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1	Me_2
10	A	7.0	5.9
10	B	0.1	-5.0
11	A	-32.2	17.7
11	B	44.5	-0.3

Example 1

```
DS_r := ceil (DS_1);
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
10	A	7	6
10	B	1	-5
11	A	-32	18
11	B	45	0

Example 2

```
DS_r := DS_1 [ calc Me_10 := ceil (Me_1) ];
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2	Me_10
10	A	7.0	5.9	7
10	B	0.1	-5.0	1
11	A	-32.2	17.7	-32
11	B	44.5	-0.3	45

Floor: *floor*

Syntax

floor (op **)**

Input parameters

op	the operand

Examples of valid syntaxes

```
floor ( DS_1 )
floor ( 3.14159 )
```

Semantics for scalar operations

The operator **floor** returns the greatest integer which is smaller than or equal to *op*. For example:

`floor( 3.1415 )` gives 3
`floor( 15 )` gives 15
`floor( -3.1415 )` gives -4
`floor( -0.1415 )` gives -1

Input parameters type

op

```
dataset { measure<number> _+ }
| component<number>
| number
```

Result type

result

```
dataset { measure<integer> _+ }
| component<integer>
| integer
```

Additional Constraints

None.

Behaviour

The operator has the behaviour of the “Operators applicable on one Scalar Value or Data Set or Data Set Component” (see the section “Typical behaviours of the ML Operators”).

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1	Me_2
10	A	7.0	5.9
10	B	0.1	-5.0
11	A	-32.2	17.7
11	B	44.5	-0.3

Example 1

```
DS_r := floor ( DS_1 );
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
10	A	7	5
10	B	0	-5
11	A	-33	17
11	B	44	-1

Example 2

```
DS_r := DS_1 [ calc Me_10 := floor (Me_1) ];
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2	Me_10
10	A	7.0	5.9	7
10	B	0.1	-5.0	0
11	A	-32.2	17.7	-33
11	B	44.5	-0.3	44

Absolute value: *abs*

Syntax

abs (op **)**

Input parameters

op	the operand

Examples of valid syntaxes

```
abs ( DS_1 )
abs ( -5 )
```

Semantics for scalar operations

The operator **abs** calculates the absolute value of a number. For example:

`abs ( -5.49 )` gives `5.49`
`abs ( 5.49 )` gives `5.49`

Input parameters type

op

```
dataset { measure<number> _+ }
| component<number>
| number
```

Result type

result

```
dataset { measure<number [ value >= 0 ]> _+ }
| component<number [ value >= 0 ]>
| number [ value >= 0 ]
```

Additional Constraints

None.

Behavior

The operator has the behaviour of the “Operators applicable on one Scalar Value or Data Set or Data Set Component” (see the section “Typical behaviours of the ML Operators”).

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1	Me_2
10	A	0.484183	0.7545
10	B	-0.515817	-13.45
11	A	-1.000000	187.0

Example 1

```
DS_r := abs ( DS_1 );
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
10	A	0.484183	0.7545
10	B	0.515817	13.45
11	A	1.000000	187

Example 2

```
DS_r := DS_1 [ calc Me_10 := abs(Me_1) ];
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2	Me_10
10	A	0.484183	0.7545	0.484183
10	B	-0.515817	-13.45	0.515817
11	A	-1.000000	187.0	1.000000

Exponential: *exp*

Syntax

exp (op **)**

Input parameters

op	the operand

Examples of valid syntaxes

```
exp ( DS_1 )
exp ( 5 )
```

Semantics for scalar operations

The operator **exp** returns *e* (base of the natural logarithm) raised to the *op*-th power. For example:

`exp ( 5 )` gives `148.41315...`

`exp ( 1 )` gives `2.71828...` (the number *e*)
`exp ( 0 )` gives `1.0`
`exp ( -1 )` gives `0.36787...` (the number *1/e*)

Input parameters type

op

```
dataset { measure<number> _+ }
| component<number>
| number
```

Result type

result

```
dataset { measure<number[value > 0]> _+ }
| component<number [value > 0]>
| number[value > 0]
```

Additional Constraints

None.

Behavior

The operator has the behaviour of the “Operators applicable on one Scalar Value or Data Set or Data Set Component” (see the section “Typical behaviours of the ML Operators”).

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1	Me_2
10	A	5	0.7545
10	B	8	13.45
11	A	2	1.87

Example 1

```
DS_r := exp(DS_1);
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
10	A	148.413	2.126547
10	B	2980.95	693842.3
11	A	7.38905	6.488296

Example 2

```
DS_r := DS_1 [ calc Me_1 := exp ( Me_1 ) ];
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
10	A	148.4131591025766	0.7545
10	B	2980.9579870417283	13.45
11	A	7.38905609893065	1.87

Natural logarithm: *ln*

Syntax

ln (op **)**

Input parameters

op	the operand

Examples of valid syntaxes

```
ln ( DS_1 )
ln ( 148 )
```

Semantics for scalar operations

The operator **ln** calculates the natural logarithm of a number. For example:

`ln ( 148 )` gives `4.997...`
`ln ( e )` gives `1.0`
`ln ( 1 )` gives `0.0`
`ln ( 0.5 )` gives `-0.693...`

Input parameters type

op

```
dataset { measure<number [value > 0] > _+ }
| component<number [value > 0] >
| number [value > 0]
```

Result type

result

```
dataset { measure<number> _+ }
| component<number>
| number
```

Additional Constraints

None.

Behaviour

The operator has the behaviour of the “Operators applicable on one Scalar Value or Data Set or Data Set Component” (see the section “Typical behaviours of the ML Operators”).

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1	Me_2
10	A	148.413	0.7545
10	B	2980.95	13.45
11	A	7.38905	1.87

Example 1

```
DS_r := ln(DS_1);
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
10	A	5.0	-0.2817
10	B	8.0	2.598979
11	A	2.0	0.625938

Example 2

```
DS_r := DS_1 [ calc Me_2 := ln ( Me_1 ) ];
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
10	A	148.413	5.0
10	B	2980.95	8.0
11	A	7.38905	2.0

Power: *power*

Syntax

power (base **,** exponent **)**

Input parameters

base	the operand
exponent	the exponent of the power

Examples of valid syntaxes

```
power ( DS_1, 2 )
power ( 5, 2 )
```

Semantics for scalar operations

The operator **power** raises a number (the *base*) to another one (the *exponent*). For example:

```
power ( 5, 2 ) gives 25
power ( 5, 1 ) gives 5
power ( 5, 0 ) gives 1
power ( 5, -1 ) gives 0.2
power ( -5, 3 ) gives -125
```

Input parameters type

base

```
dataset { measure<number> _+ }
| component<number>
| number
```

exponent

```
component<number>
| number
```

Result type

result

```
dataset { measure<number> _+ }
| component<number>
| number
```

Additional Constraints

None.

Behaviour

As for the invocations at Data Set level, the operator has the behaviour of the “Operators applicable on one Scalar Value or Data Set or Data Set Component”. As for the invocations at Component or Scalar level, the operator has the behaviour of the “Operators applicable on two Scalar Values or Data Sets or Data Set Components”, (see the section “Typical behaviours of the ML Operators”).

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1	Me_2
10	A	3	0.7545
10	B	4	13.45
11	A	5	1.87

Example 1

```
DS_r := power(DS_1, 2);
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
10	A	9.0	0.56927
10	B	16.0	180.9025

11	A	25.0	3.4969

Example 2

```
DS_r := DS_1[ calc Me_1 := power(Me_1, 2) ];
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
10	A	9.0	0.7545
10	B	16.0	13.45
11	A	25.0	1.87

Logarithm: *log*

Syntax

log (op **,** num **)**

Input parameters

op	the base of the logarithm
num	the number to which the logarithm is applied

Examples of valid syntaxes

```
log ( DS_1, 2 )
log ( 1024, 2 )
```

Semantics for scalar operations

The operator **log** calculates the logarithm of *num* base *op*. For example:

`log ( 1024, 2 )` gives `10`
`log ( 1024, 10 )` gives `3.01`

Input parameters type

op

```
dataset { measure<number [value > 1] > _+ }
| component<number [value > 1] >
| number [value > 1]
```

num

```
component<integer [value > 0]>
| integer [value > 0]
```

Result type

result

```
dataset { measure<number> _+ }
| component<number>
| number
```

Additional Constraints

None.

Behavior

As for the invocations at Data Set level, the operator has the behaviour of the “Operators applicable on one Scalar Value or Data Set or Data Set Component”. As for the invocations at Component or Scalar level, the operator has the behaviour of the “Operators applicable on two Scalar Values or Data Sets or Data Set Components”, (see the section “Typical behaviours of the ML Operators”).

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1	Me_2
10	A	1024	0.7545
10	B	64	13.45
11	A	32	1.87

Example 1

```
DS_r := log ( DS_1, 2 );
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
10	A	10.0	-0.4064071941354039
10	B	6.0	3.749534267669262
11	A	5.0	0.9030382701129122

Example 2

```
DS_r := DS_1 [ calc Me_1 := log (Me_1, 2) ];
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
10	A	10.0	0.7545
10	B	6.0	13.45
11	A	5.0	1.87

Square root: *sqrt*

Syntax

sqrt (op **)**

Input parameters

op	the operand

Examples of valid syntaxes

```
sqrt ( DS_1 )
sqrt ( 5 )
```

Semantics for scalar operations

The operator **sqrt** calculates the square root of a number. For example:

`sqrt ( 25 )` gives 5

Input parameters type

op

```
dataset { measure<number [value >= 0] > _+ }
| component<number [value >= 0] >
| number [value >= 0]
```

Result type

result

```
dataset { measure<number [value >= 0] > _+ }
| component<number [value >= 0] >
| number [value >= 0]
```

Additional Constraints

None.

Behaviour

The operator has the behaviour of the “Operators applicable on one Scalar Value or Data Set or Data Set Component” (see the section “Typical behaviours of the ML Operators”). Some valid examples could be: **sqrt (DS_1)**, **sqrt (5)**.

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1	Me_2
10	A	16	0.7545
10	B	81	13.45
11	A	64	1.87

Example 1

```
DS_r := sqrt(DS_1);
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
10	A	4.0	0.86862
10	B	9.0	3.667424
11	A	8.0	1.367479

Example 2

```
DS_r := DS_1 [ calc Me_1 := sqrt ( Me_1 ) ];
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
10	A	4.0	0.7545
10	B	9.0	13.45
11	A	8.0	1.87

Random: *random*

Syntax

random (seed **,** index **)**

Input parameters

seed	the seed
index	the index

Examples of valid syntaxes

```
random(15,12)
ds [calc r := random(col_1, 12)]
random(ds, 12);
```

Semantics for scalar operations

The operator generates a sequence number >= 0 and <1, based on seed parameter and returns the number value corresponding to index.

Input parameters type

seed

```
dataset { measure<number> _+ }
| component<number>
| number
```

index

```
integer
```

Result type

result

```
dataset { measure<number> _+ }
| component<number[0-1] >
| number[0-1]
```

Additional Constraints

None.

Behaviour

The operator returns a random decimal number >= 0 and <1.

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1
10	A	16.0
10	B	4.0
11	A	7.2

Example 1

```
DS_r := random(DS_1,5);
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1
10	A	0.3582791
10	B	0.428819
11	A	0.715488

Example 2

```
DS_r  := DS_1 [ calc Me_2 := random( Me_1, 8 ) ];
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
10	A	16.0	0.7545341
10	B	4.0	0.3457166
11	A	7.2	0.5183224

VTL-ML - Comparison Operators

Equal to: =

Syntax

left **=** right

Input parameters

left	the left operand
right	the right operand

Examples of valid syntaxes

```
DS_1 = DS_2
```

Semantics for scalar operations

The operator returns TRUE if the left is equal to right, FALSE otherwise.

For example: | `5 = 9` gives: FALSE | `5 = 5` gives: TRUE | `“hello” = “hi”` gives: FALSE

Input parameters type

left, right

```
dataset {measure<scalar>}
| component<scalar>
| scalar
```

Result type

result

```
dataset { measure<boolean> bool_var }
| component<boolean>
| boolean
```

Additional Constraints

Operands *left* and *right* must be of the same scalar type

Behaviour

The operator has the typical behaviour of the “Operators changing the data type” (see the section “Typical behaviours of the ML Operators”).

Examples

Given the operand datasets DS_1:

Example 1

```
DS_r := DS_1 = 0.08;
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Id_4	bool_var
2012	B	Total	Total	
2012	G	Total	Total	False
2012	S	Total	Total	False
2012	M	Total	Total	False
2012	F	Total	Total	True
2012	W	Total	Total	True

Example 2

```
DS_r := DS_1 [ calc Me_2 := Me_1 = 0.08 ];
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Id_4	Me_1	Me_2
2012	B	Total	Total		
2012	G	Total	Total	0.286	False
2012	S	Total	Total	0.064	False
2012	M	Total	Total	0.043	False
2012	F	Total	Total	0.08	True
2012	W	Total	Total	0.08	True

Not equal to: <>

Syntax

left **<>** right

Input parameters

left	the left operand
right	the right operand

Examples of valid syntaxes

```
DS_1 <> DS_2
```

Semantics for scalar operations

The operator returns FALSE if the left is equal to right, TRUE otherwise. For example:

5 <> 9 gives TRUE
5 <> 5 gives FALSE
“hello” <> “hi” gives TRUE

Input parameters type

left, right

```
dataset {measure<scalar> _}
| component<scalar>
| scalar
```

Result type

result

```
dataset { measure<boolean> bool_var }
| component<boolean>
| boolean
```

Additional Constraints

Operands *left* and *right* must be of the same scalar type.

Behaviour

The operator has the typical behaviour of the “Operators changing the data type” (see the section “Typical behaviours of the ML Operators”).

Examples

Given the operand datasets DS_1 and DS_2:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Id_4	Me_1
G	Total	Percentage	Total	7.1
R	Total	Percentage	Total	

Input **DS_2** (see structure)

Id_1	Id_2	Id_3	Id_4	Me_1
G	Total	Percentage	Total	7.5
R	Total	Percentage	Total	3

Example 1

```
DS_r := DS_1 <> DS_2;
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Id_4	bool_var
G	Total	Percentage	Total	True
R	Total	Percentage	Total	

Example 2

```
DS_r := DS_1 [ calc Me_2 := Me_1<>7.5 ];
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Id_4	Me_1	Me_2

G	Total	Percentage	Total	7.1	True
R	Total	Percentage	Total		

Regarding example 1, note that due to the behaviour for NULL values, if the value for *G`* in the second operand had also been NULL, then the result would still be NULL for Greece.

Greater than: > >=

Syntax

left { **>** | **>=** }[1] right

Input parameters

left	the left operand part of the comparison
right	the right operand part of the comparison

Examples of valid syntaxes

::

DS_1 > DS_2 DS_1 >= DS_2

Semantics for scalar operations

The operator **>** returns TRUE if *left* is greater than *right*, FALSE otherwise. The operator **>=** returns TRUE if *left* is greater than or equal to *right*, FALSE otherwise. For example:

`5 > 9` gives FALSE
`5 >= 5` gives TRUE
`"hello" > "hi"` gives FALSE

Input parameters type

left, right

```
dataset {measure<scalar> _}
| component<scalar>
| scalar
```

Result type

result

```
dataset { measure<boolean> bool_var }
| component<boolean>
| boolean
```

Additional Constraints

Operands *left* and *right* must be of the same scalar type.

Behaviour

The operator has the typical behaviour of the “Operators changing the data type” (see the section “Typical behaviours of the ML Operators”).

Examples

Given the operand datasets DS_1, DS_2 and DS_3:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Id_4	Id_5	Me_1
2	G	2011	Total	Percentage	
2	R	2011	Total	Percentage	12.2
2	F	2011	Total	Percentage	29.5

Input **DS_2** (see structure)

Id_1	Id_2	Id_3	Id_4	Me_1
G	Total	Percentage	Total	7.1
R	Total	Percentage	Total	42.5

Input **DS_3** (see structure)

Id_1	Id_2	Id_3	Id_4	Me_1
G	Total	Percentage	Total	7.5
R	Total	Percentage	Total	33.7

Example 1

```
DS_r := DS_1 > 20;
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Id_4	Id_5	bool_var
2	G	2011	Total	Percentage	
2	R	2011	Total	Percentage	False
2	F	2011	Total	Percentage	True

Example 2

```
DS_r := DS_1 [ calc Me_2 := Me_1 > 20 ];
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Id_4	Id_5	Me_1	Me_2
2	G	2011	Total	Percentage		
2	R	2011	Total	Percentage	12.2	False
2	F	2011	Total	Percentage	29.5	True

Example 3

```
DS_r:= DS_2 > DS_3;
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Id_4	bool_var

G	Total	Percentage	Total	False
R	Total	Percentage	Total	True

Note that, for example, if the Me_1 column for Germany in the DS_2 or DS_3 Data Set had a NULL value the result would be NULL for Germany (G) and TRUE for Greece (R).

Less than < <=

Syntax

left { **<** | **<=** }[1] right

Input parameters

left	the left operand
right	the right operand

Examples of valid syntaxes

```
DS_1 < DS_2
DS_1 <= DS_2
```

Semantics for scalar operations

The operator **<** returns TRUE if left is smaller than right, FALSE otherwise. The operator **<=** returns TRUE if left is smaller than or equal to right, FALSE otherwise. For example:

`5 < 4` gives FALSE
`5 <= 5` gives TRUE
`"hello" < "hi"` gives TRUE

Input parameters type

left, right

```
dataset {measure<scalar> _}
| component<scalar>
| scalar
```

Result type

result

```
dataset { measure<boolean> bool_var }
| component<boolean>
| boolean
```

Additional Constraints

Operands *left* and *right* must be of the same scalar type.

Behaviour

The operator has the typical behaviour of the “Operators changing the data type” (see the section “Typical behaviours of the ML Operators”). Some valid examples could be: **DS_1 < DS_2**, **DS_1 <= DS_2**.

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Id_4	Me_1
2012	B	Total	Total	11094850
2012	G	Total	Total	11123034
2012	S	Total	Total	46818219
2012	M	Total	Total	
2012	F	Total	Total	5401267
2012	W	Total	Total	7954662

Example 1

```
DS_r := DS_1 < 15000000;
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Id_4	bool_var
2012	B	Total	Total	True
2012	G	Total	Total	True
2012	S	Total	Total	False
2012	M	Total	Total	
2012	F	Total	Total	True
2012	W	Total	Total	True

Between *between*

Syntax

between (op, from, to **)**

Input parameters

op	the Data Set to be checked
from	the left delimiter
to	the right delimiter

Examples of valid syntaxes

```
ds2 := between(ds1, 5, 10)
ds2 := ds1 [ calc m1 := between(me2, 5, 10) ]
```

Semantics for scalar operations

The operator returns TRUE if *op* is greater than or equal to *from* and lower than or equal to *to*. In other terms, it is a shortcut for the following:

```
op >= from and op <= to
```

The types of *op*, *from* and *to* must be compatible scalar types.

Input parameters type

op

```
dataset {measure<scalar> _}
| component<scalar>
| scalar
```

from, to

```
component<scalar>
| scalar
```

Result type

result

```
dataset { measure<boolean> bool_var }
| component<boolean>
| boolean
```

Additional Constraints

The type of the operand (i.e., the measure of the dataset, the type of the component, the scalar type) must be the same as that of *from* and *to*.

Behaviour

The operator has the typical behaviour of the “Operators changing the data type” (see the section “Typical behaviours of the ML Operators”)

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Id_4	Me_1
G	Total	Percentage	Total	6
R	Total	Percentage	Total	-2

Example 1

```
DS_r:= between(DS_1, 5, 10);
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Id_4	bool_var
G	Total	Percentage	Total	True
R	Total	Percentage	Total	False

Element of *in / not_in*

Syntax

op **in** collection
op **not_in** collection

collection ::= set | valueDomainName

Input parameters

op	the operand to be tested
collection	the Set or the Value Domain which contains the values
set	the Set which contains the values (it can be a Set name or a Set literal)
valueDomainName	the name of the Value Domain which contains the values

Examples of valid syntaxes

```
ds := ds_2 in {1,4,6}
ds := ds_3 in mySet
ds := ds_3 in myValueDomain
```

Semantics for scalar operations

The **in** operator returns TRUE if *op* belongs to the collection, FALSE otherwise. The **not_in** operator returns FALSE if *op* belongs to the collection, TRUE otherwise. For example:

`1 in { 1, 2, 3 }` returns TRUE
`“a” in { “c, “ab”, “bb”, “bc” }` returns FALSE
`“b” not_in { “b”, ”hello”, ”c”}` returns FALSE
`“b” not_in { “a”, ”hello”, ”c”}` returns TRUE

Input parameters type

op

```
dataset {measure<scalar> _}
| component<scalar>
| scalar
```

collection

```
set<scalar> | name<value_domain>
```

Result type

result

```
dataset { measure<boolean> bool_var }
| component<boolean>
| boolean
```

Additional Constraints

The operand must be of a basic scalar data type compatible with the basic scalar type of the collection.

Behavior

The **in** operator evaluates to TRUE if the operand is an element of the specified collection and FALSE otherwise, the **not_in** the opposite.

The operator has the typical behaviour of the “Operators changing the data type” (see the section “Typical behaviours of the ML Operators”).

The *collection* can be either a *set* of values defined in line or a name that references an externally defined Value Domain or Set.

Examples

Given the operand dataset DS_1 and the Value Domain named myGeoValueDomain (which has the basic scalar type *string*) defined by {“AF”, “BS”, “FJ”, “GA”, “KH”, “MO”, “PK”, “QA”, “UG”}:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1
2012	BS	0
2012	GZ	4
2012	SQ	9
2012	MO	6
2012	FJ	7
2012	CQ	2

Example 1

```
DS_r := DS_1 in { 0, 3, 6, 12 };
```

results in (see structure):

DS_r

Id_1	Id_2	bool_var
2012	BS	True
2012	GZ	False
2012	SQ	False
2012	MO	True
2012	FJ	False
2012	CQ	False

Example 2

```
DS_r := DS_1 [ calc Me_2:= Me_1 in { 0, 3, 6, 12 } ];
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
2012	BS	0	True
2012	GZ	4	False
2012	SQ	9	False
2012	MO	6	True
2012	FJ	7	False
2012	CQ	2	False

Example 3

```
DS_r := DS_1#Id_2 in myGeoValueDomain;
```

results in (see structure):

DS_r

Id_1	Id_2	bool_var
2012	BS	True
2012	GZ	False
2012	SQ	False
2012	MO	True
2012	FJ	True
2012	CQ	False

Match characters: *match_characters*

Syntax

match_characters (op , pattern **)**

Input parameters

op	the dataset to be checked
pattern	the regular expression to check the Data Set or the Component against

Examples of valid syntaxes

```
match_characters(ds1, "[abc]+\d\d")
ds1 [ calc m1 := match_characters(ds1, "[abc]+\d\d") ]
```

Semantics for scalar operations

match_characters returns TRUE if *op* matches the regular expression *regexp*, FALSE otherwise.

The string *regexp* is an Extended Regular Expression as described in the POSIX standard. Different implementations of VTL may implement different versions of the POSIX standard therefore it is possible that **match_characters** may behave in slightly different ways.

Input parameters type

op

```
dataset {measure<string> _}
| component<string>
| string
```

pattern

```
component<string>
| string
```

Result type

result

```
dataset { measure<boolean> bool_var }
| component<boolean>
| boolean
```

Additional Constraints

If *op* is a Data Set then it has exactly one measure.

pattern is a POSIX regular expression.

Behaviour

The operator has the typical behaviour of the “Operators changing the data type” (see the section “Typical behaviours of the ML Operators”).

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Id_4	Me_1
G	Total	Percentage	Total	AX123
R	Total	Percentage	Total	AX2J5

Example 1

```
DS_r:=match_characters(DS_1, "[:alpha:]{2}[:digit:]{3}");
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Id_4	bool_var
G	Total	Percentage	Total	True
R	Total	Percentage	Total	False

Is null: *isnull*

Syntax

isnull (op **)**

Input parameters

op	the operand

Examples of valid syntaxes

```
isnull(DS_1)
```

Semantics for scalar operations

The operator returns TRUE if the value of the operand is NULL, FALSE otherwise. For example:

`isnull(“Hello”)` gives FALSE
`isnull(NULL)` gives TRUE

Input parameters type

op

```
dataset {measure<scalar> _}
| component<scalar>
| scalar
```

Result type

result

```
dataset { measure<boolean> bool_var }
| component<boolean>
| boolean
```

Additional Constraints

If *op* is a Data Set then it has exactly one measure.

Behaviour

The operator has the typical behaviour of the “Operators changing the data type” (see the section “Typical behaviours of the ML Operators”).

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Id_4	Me_1
2012	B	Total	Total	11094850
2012	G	Total	Total	11123034
2012	S	Total	Total	
2012	M	Total	Total	46818219
2012	F	Total	Total	5401267
2012	W	Total	Total	

Example 1

```
DS_r := isnull(DS_1);
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Id_4	bool_var
2012	B	Total	Total	False
2012	G	Total	Total	False
2012	S	Total	Total	True
2012	M	Total	Total	False
2012	F	Total	Total	False
2012	W	Total	Total	True

Example 2

```
DS_r := DS_1[ calc Me_2 := isnull(Me_1) ];
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Id_4	Me_1	Me_2
2012	B	Total	Total	11094850	False
2012	G	Total	Total	11123034	False
2012	S	Total	Total		True
2012	M	Total	Total	46818219	False
2012	F	Total	Total	5401267	False
2012	W	Total	Total		True

Exists in: *exists_in*

Syntax

exists_in (op1, op2 { , retain } **)**

retain ::= **true** | **false** | **all**

Input parameters

op1	the operand dataset
op2	the operand dataset
retain	the optional parameter to specify the Data Points to be returned (default: **all**)

Examples of valid syntaxes

```
exists_in(DS_1, DS_2, true)
exists_in(DS_1, DS_2)
exists_in(DS_1, DS_2, all)
```

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

op1, op2

```
dataset
```

Result type

result

```
dataset { measure<boolean> bool_var }
```

Additional Constraints

op1 has at least all the identifier components of *op2* or *op2* has at least all the identifier components of *op1*.

Behaviour

The operator takes under consideration the common Identifiers of *op1* and *op2* and checks if the combinations of values of these Identifiers which are in *op1* also exist in *op2*.

The result has the same Identifiers as *op1* and a *boolean* Measure bool_var whose value, for each Data Point of *op1*, is TRUE if the combination of values of the common Identifier Components in *op1* is found in a Data Point of *op2*, FALSE otherwise.

If *retain* is **all** then both the Data Points having bool_var = TRUE and bool_var = FALSE are returned.

If *retain* is **true** then only the data points with bool_var = TRUE are returned.

If *retain* is **false** then only the Data Points with bool_var = FALSE are returned.

If the *retain* parameter is omitted, the default is **all**.

The operator has the typical behaviour of the “Operators changing the data type” (see the section “Typical behaviours of the ML Operators”).

Examples

Given the operand datasets DS_1 and DS_2:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Id_4	Me_1
2012	B	Total	Total	11094850
2012	G	Total	Total	11123034
2012	S	Total	Total	46818219
2012	M	Total	Total	417546
2012	F	Total	Total	5401267
2012	W	Total	Total	7954662

Input **DS_2** (see structure)

Id_1	Id_2	Id_3	Id_4	Me_1
2012	B	Total	Total	0.023
2012	G	Total	M	0.286
2012	S	Total	Total	0.064
2012	M	Total	M	0.043
2012	F	Total	Total	
2012	W	Total	Total	0.08

Example 1

```
DS_r := exists_in (DS_1, DS_2, all);
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Id_4	bool_var
2012	B	Total	Total	True
2012	G	Total	Total	False
2012	S	Total	Total	True
2012	M	Total	Total	False

2012	F	Total	Total	True
2012	W	Total	Total	True

Example 2

```
DS_r := exists_in (DS_1, DS_2, true);
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Id_4	bool_var
2012	B	Total	Total	True
2012	S	Total	Total	True
2012	F	Total	Total	True
2012	W	Total	Total	True

Example 3

```
DS_r := exists_in (DS_1, DS_2, false);
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Id_4	bool_var
2012	G	Total	Total	False
2012	M	Total	Total	False

VTL-ML - Boolean Operators

Logical conjunction: *and*

Syntax

op1 **and** op2

Input parameters

op1	the first operand
op2	the second operand

Examples of valid syntaxes

```
DS_1 and DS_2
```

Semantics for scalar operations

The **and** operator returns TRUE if both operands are TRUE, otherwise FALSE. The two operands must be of *boolean* type. For example: | `FALSE and FALSE` gives `FALSE` | `FALSE and TRUE` gives `FALSE` | `FALSE and NULL` gives `FALSE` | `TRUE and FALSE` gives `FALSE` | `TRUE and TRUE` gives `TRUE` | `TRUE and NULL` gives `NULL` | `NULL and NULL` gives `NULL`

Input parameters type

op1, op2

```
dataset {measure<boolean> _ }
| component<boolean>
| boolean
```

Result type

result

```
dataset {measure<boolean> _ }
| component<boolean>
| boolean
```

Additional Constraints

None.

Behavior

The operator has the typical behaviour of the “Behaviour of Boolean operators” (see the section “Typical behaviours of the ML Operators”).

Examples

Given the operand datasets DS_1 and DS_2:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Id_4	Me_1
M	15	B	2013	True
M	64	B	2013	False
M	65	B	2013	True
F	15	U	2013	False
F	64	U	2013	False
F	65	U	2013	True

Input **DS_2** (see structure)

Id_1	Id_2	Id_3	Id_4	Me_1
M	15	B	2013	False
M	64	B	2013	True
M	65	B	2013	True
F	15	U	2013	True
F	64	U	2013	False
F	65	U	2013	False

Example 1

```
DS_r:= DS_1 and DS_2;
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Id_4	Me_1
M	15	B	2013	False
M	64	B	2013	False
M	65	B	2013	True
F	15	U	2013	False
F	64	U	2013	False
F	65	U	2013	False

Example 2

```
DS_r := DS_1 [ calc Me_2:= Me_1 and true ];
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Id_4	Me_1	Me_2
M	15	B	2013	True	True
M	64	B	2013	False	False
M	65	B	2013	True	True
F	15	U	2013	False	False
F	64	U	2013	False	False
F	65	U	2013	True	True

Logical disjunction: *or*

Syntax

op1 **or** op2

Input parameters

op1	the first operand
op2	the second operand

Examples of valid syntaxes

```
DS_1 or DS_2
```

Semantics for scalar operations

The **or** operator returns TRUE if at least one of the operands is TRUE, otherwise FALSE. The two operands must be of *boolean* type. For example: | `FALSE or FALSE` gives FALSE | `FALSE or TRUE` gives TRUE | `FALSE or NULL` gives `NULL` | `TRUE or FALSE` gives `TRUE` | `TRUE or TRUE` gives `TRUE` | `TRUE or NULL` gives `TRUE` | `NULL or NULL` gives `NULL`

Input parameters type

op1, op2

```
dataset {measure<boolean> _ }
| component<boolean>
| boolean
```

Result type

result

```
dataset {measure<boolean> _ }
| component<boolean>
| boolean
```

Additional Constraints

None.

Behavior

The operator has the typical behaviour of the “Behaviour of Boolean operators” (see the section “Typical behaviours of the ML Operators”).

Examples

Given the operand datasets DS_1 and DS_2:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Id_4	Me_1
M	15	B	2013	True
M	64	B	2013	False
M	65	B	2013	True
F	15	U	2013	False
F	64	U	2013	False
F	65	U	2013	True

Input **DS_2** (see structure)

Id_1	Id_2	Id_3	Id_4	Me_1
M	15	B	2013	False
M	64	B	2013	True
M	65	B	2013	True
F	15	U	2013	True
F	64	U	2013	False
F	65	U	2013	False

Example 1

```
DS_r:= DS_1 or DS_2;
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Id_4	Me_1
M	15	B	2013	True

M	64	B	2013	True
M	65	B	2013	True
F	15	U	2013	True
F	64	U	2013	False
F	65	U	2013	True

Example 2

```
DS_r:= DS_1 [ calc Me_2:= Me_1 or true ];
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Id_4	Me_1	Me_2
M	15	B	2013	True	True
M	64	B	2013	False	True
M	65	B	2013	True	True
F	15	U	2013	False	True
F	64	U	2013	False	True
F	65	U	2013	True	True

Exclusive disjunction: *xor*

Syntax

op1 **xor** op2

Input parameters

op1	the first operand
op2	the second operand

Examples of valid syntaxes

```
DS_1 xor DS_2
```

Semantics for scalar operations

The **xor** operator returns TRUE if only one of the operand is TRUE (but not both), FALSE otherwise. The two operands must be of *boolean* type. For example: | `FALSE xor FALSE` gives FALSE | `FALSE xor TRUE` gives TRUE | `FALSE xor NULL` gives `NULL` | `TRUE xor FALSE` gives `TRUE` | `TRUE xor TRUE` gives FALSE | `TRUE xor NULL` gives `NULL` | `NULL xor NULL` gives `NULL`

Input parameters type

op1, op2

```
dataset {measure<boolean> _ }
| component<boolean>
| boolean
```

Result type

result

```
dataset {measure<boolean> _ }
| component<boolean>
| boolean
```

Additional Constraints

None.

Behaviour

The operator has the typical behaviour of the “Behaviour of Boolean operators” (see the section “Typical behaviours of the ML Operators”).

Examples

Given the operand datasets DS_1 and DS_2:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Id_4	Me_1
M	15	B	2013	True
M	64	B	2013	False
M	65	B	2013	True
F	15	U	2013	False
F	64	U	2013	False
F	65	U	2013	True

Input **DS_2** (see structure)

Id_1	Id_2	Id_3	Id_4	Me_1
M	15	B	2013	False
M	64	B	2013	True
M	65	B	2013	True
F	15	U	2013	True
F	64	U	2013	False
F	65	U	2013	False

Example 1

```
DS_r:=DS_1 xor DS_2;
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Id_4	Me_1
M	15	B	2013	True
M	64	B	2013	True
M	65	B	2013	False
F	15	U	2013	True

F	64	U	2013	False
F	65	U	2013	True

Example 2

```
DS_r:= DS_1 [ calc Me_2:= Me_1 xor true ];
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Id_4	Me_1	Me_2
M	15	B	2013	True	False
M	64	B	2013	False	True
M	65	B	2013	True	False
F	15	U	2013	False	True
F	64	U	2013	False	True
F	65	U	2013	True	False

Logical negation: *not*

Syntax

not op

Input parameters

op	the first operand

Examples of valid syntaxes

```
not DS_1
```

Semantics for scalar operations

The **not** operator returns TRUE if *op* is FALSE, otherwise TRUE. The input operand must be of *boolean* type. For example: | `not` FALSE gives TRUE | `not` TRUE gives FALSE | `not` NULL gives NULL

Input parameters type

op

```
dataset {measure<boolean> _ }
| component<boolean>
| boolean
```

Result type

result

```
dataset {measure<boolean> _ }
| component<boolean>
| boolean
```

Additional Constraints

None.

Behaviour

The operator has the typical behaviour of the “Behaviour of Boolean operators” (see the section “Typical behaviours of the ML Operators”).

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Id_4	Me_1
M	15	B	2013	True
M	64	B	2013	False
M	65	B	2013	True
F	15	U	2013	False
F	64	U	2013	False
F	65	U	2013	True

Example 1

```
DS_r:= not DS_1;
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Id_4	Me_1
M	15	B	2013	False
M	64	B	2013	True
M	65	B	2013	False
F	15	U	2013	True
F	64	U	2013	True
F	65	U	2013	False

Example 2

```
DS_r:= DS_1 [ calc Me_2 := not Me_1 ];
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Id_4	Me_1	Me_2
M	15	B	2013	True	False
M	64	B	2013	False	True
M	65	B	2013	True	False
F	15	U	2013	False	True

F	64	U	2013	False	True
F	65	U	2013	True	False

VTL-ML - Time Operators

This chapter describes the time operators, which are the operators dealing with time, date and time_period basic scalar types. The general aspects of the behaviour of these operators is described in the section “Behaviour of the Time Operators”. The time data type is the most general type and denotes a generic time interval, having start and end points in time and therefore a duration, which is the time intervening between the start and end points. The date data type denotes a generic time instant (a point in time), which is a time interval with zero duration. The time_period data type denotes a regular time interval whose regular duration is explicitly represented inside each time_period value and is named period_indicator. In some sense, we say that date and time_period are special cases of time, the former with coinciding extremes and zero duration and the latter with regular duration. The time data type is overarching in the sense that it comprises date and time_period. Finally, duration data type represents a generic time span, independently of any specific start and end date.

The time, date and time period formats used here are explained in the User Manual in the section “External representations and literals used in the VTL Manuals”.

The period indicator P id of the *duration* type and its possible values are:

D Day
W Week
M Month
Q Quarter
S Semester
A Year

As already said, these representation are not prescribed by VTL and are not part of the VTL standard, each VTL system can personalize the representation of time, date, time_period and duration as desired. The formats shown above are only the ones used in the examples.

For a fully-detailed explanation, please refer to the User Manual.

Period indicator: *period_indicator*

Syntax

period_indicator ({ *op* } **)**

Input parameters

op	the operand

Examples of valid syntaxes

::

period_indicator (ds_1) period_indicator // (if used in a clause the operand op can be omitted)

Semantics for scalar operations

period_indicator returns the period indicator of a *time_period* value. The period indicator is the part of the *time_period* value which denotes the duration of the time period (e.g. day, week, month…).

Input parameters type

op

```
dataset { identifier <time_period> _ , identifier _* }
| component<time_period>
| time_period
```

Result type

result

```
dataset { measure<duration> duration_var }
| component<duration>
| duration
```

Additional Constraints

If *op* is a Data Set then it has exactly one Identifier of type *time_period* and may have other Identifiers. If the operator is used in a clause and *op* is omitted, then the Data Set to which the clause is applied has exactly one Identifier of type *time_period* and may have other Identifiers.

Behaviour

The operator extracts the period indicator part of the *time_period* value. The period indicator is computed for each Data Point. When the operator is used in a clause, it extracts the period indicator from the *time_period* value the Data Set to which the clause is applied.

The operator returns a Data Set with the same Identifiers of *op* and one Measure of type *duration* named *duration_var*. As for all the Variables, a proper Value Domain must be defined to contain the possible values of the period indicator and *duration_var*. The values used in the examples are listed at the beginning of this chapter “VTL-ML Time operators”.

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Me_1
A	1	2010	10
A	1	2013Q1	50

Example 1

```
DS_r := period_indicator ( DS_1 );
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	duration_var
A	1	2010	A
A	1	2013-Q1	Q

Example 2

```
DS_r := DS_1 [ filter period_indicator ( Id_3 ) = "A" ];
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Me_1
A	1	2010	10

Fill time series: *fill_time_series*

Syntax

fill_time_series (op { , limitsMethod } **)**

limitsMethod ::= **single** | **all**

Input parameters

op	the operand
limitsMethod	method for determining the limits of the time interval to be filled (default: **all**)

Examples of valid syntaxes

```
fill_time_series ( ds )
fill_time_series ( ds, all )
```

Semantics for scalar operations

The **fill_time_series** operator does not perform scalar operations.

Input parameters type

op

```
dataset { identifier <time > _ , identifier _* }
```

Result type

result

```
dataset { identifier <time > _ , identifier _* }
```

Additional Constraints

The operand *op* has an Identifier of type *time*, *date* or *time_period* and may have other Identifiers.

Behaviour

This operator can be applied only on Data Sets of time series and returns a Data Set of time series.

The operator fills the possibly missing Data Points of all the time series belonging to the operand *op* within the time limits automatically determined by applying the *limit_method*.

If *limitsMethod* is **all**, the time limits are determined with reference to all the time_series of the Data Set: the limits are the minimum and the maximum values of the reference time Identifier Component of the Data Set.

If *limitsMethod* is **single**, the time limits are determined with reference to each single time_series of the Data Set: the limits are the minimum and the maximum values of the reference time Identifier Component of the time series.

The expected Data Points are determined, for each time series, by considering the limits above and the *period* (*frequency*) of the time series: all the Identifiers are kept unchanged except the reference time Identifier, which is increased of one *period* at a time (e.g. day, week, month, quarter, year) from the lower to the upper time limit. For each increase, an expected Data Point is identified.

If this expected Data Points is missing, it is added to the Data Set. For the added Data Points, Measures and Attributes assume the NULL value.

The output Data Set has the same Identifier, Measure and Attribute Components as the operand Data Set. The output Data Set contains the same time series as the operand, because the time series Identifiers (all the Identifiers except the reference time Identifier) are not changed.

As mentioned in the section “Behaviour of the Time Operators”, the operator is assumed to know which is the reference time Identifier as well as the period of each time series. Some valid examples could be: **fill_time_series (ds)**, **fill_time_series (ds, all)**.

Examples

As described in the User Manual, the *time* data type is the intervening time between two time points and using the ISO 8601 standard it can be expressed through a start date and an end date separated by a slash at any precision. In the examples relevant to the *time* data type the precision is set at the level of month and the time format YYYY-MM/YYYY-MM is used.

Given:

- The operand dataset DS_1, which contains *annual* time series, where *Id_2* is the reference time Identifier of *time* type;
- the operand dataset DS_2, which contains *annual* time series, where *Id_2* is the reference time Identifier of *date* type and conventionally each period is identified by its last day;
- the operand dataset DS_3, which contains *annual* time series, where *Id_2* is the reference time Identifier of *time_period* type;
- and the operand dataset DS_4, which contains both *quarterly* and *annual* time series relevant to the same phenomenon “A”, where *Id_2* is the reference time Identifier of *time_period* type:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1
A	2010M1/2010M12	hello world
A	2012M1/2012M12	say hello
A	2013M1/2013M12	he
B	2011M1/2011M12	hi, hello!
B	2012M1/2012M12	hi
B	2014M1/2014M12	hello!

Input **DS_2** (see structure)

Id_1	Id_2	Me_1
A	2010-12-31	hello world
A	2012-12-31	say hello
A	2013-12-31	he
B	2011-12-31	hi, hello!
B	2012-12-31	hi
B	2014-12-31	hello!

Input **DS_3** (see structure)

Id_1	Id_2	Me_1
A	2010	hello world
A	2012	say hello
A	2013	he
B	2011	hi, hello!
B	2012	hi
B	2014	hello!

Input **DS_4** (see structure)

Id_1	Id_2	Me_1
A	2010	hello world
A	2012	say hello
A	2010Q1	he
A	2010Q2	hi, hello!
A	2010Q4	hi
A	2011Q2	hello!

Example 1

```
DS_r := fill_time_series ( DS_1, single );
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1
A	2010M1/2010M12	hello world
A	2011M1/2011M12	
A	2012M1/2012M12	say hello
A	2013M1/2013M12	he
B	2011M1/2011M12	hi, hello!
B	2012M1/2012M12	hi
B	2013M1/2013M12	
B	2014M1/2014M12	hello!

Example 2

```
DS_r := fill_time_series ( DS_1, all );
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1
A	2010M1/2010M12	hello world
A	2011M1/2011M12	
A	2012M1/2012M12	say hello
A	2013M1/2013M12	he
A	2014M1/2014M12	
B	2010M1/2010M12	
B	2011M1/2011M12	hi, hello!
B	2012M1/2012M12	hi
B	2013M1/2013M12	
B	2014M1/2014M12	hello!

Example 3

```
DS_r := fill_time_series ( DS_2, single );
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1
A	2010-12-31	hello world
A	2011-12-31	
A	2012-12-31	say hello
A	2013-12-31	he
B	2011-12-31	hi, hello!
B	2012-12-31	hi
B	2013-12-31	
B	2014-12-31	hello!

Example 4

```
DS_r := fill_time_series ( DS_2, all );
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1
A	2010-12-31	hello world
A	2011-12-31	
A	2012-12-31	say hello
A	2013-12-31	he
A	2014-12-31	
B	2010-12-31	
B	2011-12-31	hi, hello!
B	2012-12-31	hi
B	2013-12-31	
B	2014-12-31	hello!

Example 5

```
DS_r := fill_time_series ( DS_3, single );
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1
A	2010	hello world

A	2011	
A	2012	say hello
A	2013	he
B	2011	hi, hello!
B	2012	hi
B	2013	
B	2014	hello!

Example 6

```
DS_r := fill_time_series ( DS_3, all );
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1
A	2010	hello world
A	2011	
A	2012	say hello
A	2013	he
A	2014	
B	2010	
B	2011	hi, hello!
B	2012	hi
B	2013	
B	2014	hello!

Example 7

```
DS_r := fill_time_series ( DS_4, single );
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1
A	2010	hello world
A	2011	
A	2012	say hello
A	2010Q1	he
A	2010Q2	hi, hello!
A	2010Q3	
A	2010Q4	hi
A	2011Q1	
A	2011Q2	hello!

Example 8

```
DS_r := fill_time_series ( DS_4, all );
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1
A	2010	hello world
A	2011	
A	2012	say hello
A	2010-Q1	he
A	2010-Q2	hi, hello!
A	2010-Q3	
A	2010-Q4	hi
A	2011-Q1	
A	2011-Q2	hello!
A	2011-Q3	
A	2011-Q4	
A	2012-Q1	
A	2012-Q2	
A	2012-Q3	
A	2012-Q4	

Flow to stock: *flow_to_stock*

Syntax

flow_to_stock (op **)**

Input parameters

op	the operand

Examples of valid syntaxes

```
flow_to_stock ( ds_1 )
```

Semantics for scalar operations

This operator does not perform scalar operations.

Input parameters type

op

```
dataset { identifier < time > _ , identifier _* , measure<number> _+ }
```

Result type

result

```
dataset { identifier < time > _ , identifier _* , measure<number> _+ }
```

Additional Constraints

The operand dataset has an Identifier of type *time*, *date* or *time_period* and may have other Identifiers.

Behavior

The statistical data that describe the “state” of a phenomenon on a given moment (e.g. resident population on a given moment) are often referred to as “stock data”.

On the contrary, the statistical data that describe “events” which can happen continuously (e.g. changes in the resident population, such as births, deaths, immigration, emigration), are often referred to as “flow data”.

This operator takes in input a Data Set which are interpreted as flows and calculates the change of the corresponding stock since the beginning of each time series by summing the relevant flows. In other words, the operator perform the cumulative sum from the first Data Point of each time series to each other following Data Point of the same time series.

The *flow_to_stock* operator can be applied only on Data Sets of time series and returns a Data Set of time series.

The result Data Set has the same Identifier, Measure and Attribute Components as the operand Data Set and contains the same time series as the operand, because the time series Identifiers (all the Identifiers except the reference time Identifier) are not changed.

As mentioned in the section “Behaviour of the Time Operators”, the operator is assumed to know which is the *time* Identifier as well as the *period* of each time series.

Examples

As described in the User Manual, the *time* data type is the intervening time between two time points and using the ISO 8601 standard it can be expressed through a start date and an end date separated by a slash at any precision. In the examples relevant to the *time* data type the precision is set at the level of month and the time format YYYY-MM/YYYY-MM is used.

Given:

- The operand dataset DS_1, which contains *annual* time series, where *Id_2* is the reference time Identifier of *time* type;
- the operand dataset DS_2, which contains *annual* time series, where *Id_2* is the reference time Identifier of *date* type (conventionally each period is identified by its last day);
- the operand dataset DS_3, which contains *annual* time series, where *Id_2* is the reference time Identifier of *time_period* type;
- the operand dataset DS_4, which contains both *quarterly* and *annual* time series relevant to the same phenomenon “A”, where *Id_2* is the reference time Identifier of *time_period* type:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1
A	2010M1/2010M12	2.0
A	2011M1/2011M12	5.0
A	2012M1/2012M12	-3.0
A	2013M1/2013M12	9.0
B	2010M1/2010M12	4.0
B	2011M1/2011M12	-8.0
B	2012M1/2012M12	0.0
B	2013M1/2013M12	6.0

Input **DS_2** (see structure)

Id_1	Id_2	Me_1
A	2010-12-31	2.0
A	2011-12-31	5.0
A	2012-12-31	-3.0
A	9999-12-31	9.0
B	2010-12-31	4.0
B	2011-12-31	-8.0
B	2012-12-31	0.0
B	9999-12-31	6.0

Input **DS_3** (see structure)

Id_1	Id_2	Me_1
A	2010	2.0
A	2011	5.0
A	2012	-3.0
A	2013	9.0
B	2010	4.0
B	2011	-8.0
B	2012	0.0
B	2013	6.0

Input **DS_4** (see structure)

Id_1	Id_2	Me_1
A	2010	2.0
A	2011	7.0
A	2012	4.0
A	2013	13.0
A	2010Q1	2.0
A	2010Q2	-3.0
A	2010Q3	7.0
A	2010Q4	-4.0

Example 1

```
DS_r := flow_to_stock ( DS_1 );
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1
A	2010M1/2010M12	2.0
A	2011M1/2011M12	7.0
A	2012M1/2012M12	4.0

A	2013M1/2013M12	13.0
B	2010M1/2010M12	4.0
B	2011M1/2011M12	-4.0
B	2012M1/2012M12	-4.0
B	2013M1/2013M12	2.0

Example 2

```
DS_r := flow_to_stock ( DS_2 );
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1
A	2010-12-31	2.0
A	2011-12-31	7.0
A	2012-12-31	4.0
A	9999-12-31	13.0
B	2010-12-31	4.0
B	2011-12-31	-4.0
B	2012-12-31	-4.0
B	9999-12-31	2.0

Example 3

```
DS_r := flow_to_stock ( DS_3 );
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1
A	2010	2.0
A	2011	7.0
A	2012	4.0
A	2013	13.0
B	2010	4.0
B	2011	-4.0
B	2012	-4.0
B	2013	2.0

Example 4

```
DS_r := flow_to_stock ( DS_4 );
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1
A	2010	2.0
A	2011	9.0
A	2012	13.0
A	2013	26.0
A	2010-Q1	2.0
A	2010-Q2	-1.0
A	2010-Q3	6.0
A	2010-Q4	2.0

Stock to flow: *stock_to_flow*

Syntax

stock_to_flow (op **)**

Input parameters

op	the operand

Examples of valid syntaxes

```
stock_to_flow ( ds_1 )
```

Semantics for scalar operations

This operator does not perform scalar operations.

Input parameters type

op

```
dataset { identifier < time > _ , identifier _* , measure<number> _+ }
```

Result type

result

```
dataset { identifier < time > _ , identifier _* , measure<number> _+ }
```

Additional Constraints

The operand dataset has an Identifier of type *time*, *date* or *time_period* and may have other Identifiers.

Behaviour

The statistical data that describe the “state” of a phenomenon on a given moment (e.g. resident population on a given moment) are often referred to as “stock data”.

On the contrary, the statistical data that describe “events” which can happen continuously (e.g. changes in the resident population, such as births, deaths, immigration, emigration), are often referred to as “flow data”.

This operator takes in input a Data Set of time series which is interpreted as stock data and, for each time series, calculates the corresponding flow data by subtracting from the measure values of each regular period the corresponding measure values of the previous one.

The *stock_to_flow* operator can be applied only on Data Sets of time series and returns a Data Set of time series. The result Data Set has the same Identifier, Measure and Attribute Components as the operand Data Set and contains the same time series as the operand, because the time series Identifiers (all the Identifiers except the reference time Identifier) are not changed.

The Attribute propagation rule is not applied.

As mentioned in the section “Behaviour of the Time Operators”, the operator is assumed to know which is the *time* Identifier as well as the *period* of each time series.

Examples

As described in the User Manual, the *time* data type is the intervening time between two time points and using the ISO 8601 standard it can be expressed through a start date and an end date separated by a slash at any precision. In the examples relevant to the *time* data type the precision is set at the level of month and the time format YYYY-MM/YYYY-MM is used.

Given:

- The operand dataset DS_1, which contains *annual* time series, where *Id_2* is the reference time Identifier of *time* type;
- the operand dataset DS_2, which contains *annual* time series, where *Id_2* is the reference time Identifier of *date* type (conventionally each period is identified by its last day);
- the operand dataset DS_3, which contains *annual* time series, where *Id_2* is the reference time Identifier of *time_period* type;
- and the operand dataset DS_4, which contains both *quarterly* and *annual* time series relevant to the same phenomenon “A”, where *Id_2* is the time Identifier of *time_period* type:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1
A	2010M1/2010M12	2.0
A	2011M1/2011M12	7.0
A	2012M1/2012M12	4.0
A	2013M1/2013M12	13.0
B	2010M1/2010M12	4.0
B	2011M1/2011M12	-4.0
B	2012M1/2012M12	-4.0
B	2013M1/2013M12	2.0

Input **DS_2** (see structure)

Id_1	Id_2	Me_1
A	2010-12-31	2.0
A	2011-12-31	7.0
A	2012-12-31	4.0
A	2013-12-31	13.0
B	2010-12-31	4.0
B	2011-12-31	-4.0
B	2012-12-31	-4.0
B	2013-12-31	2.0

Input **DS_3** (see structure)

Id_1	Id_2	Me_1

A	2010	2.0
A	2011	7.0
A	2012	4.0
A	2013	13.0
B	2010	4.0
B	2011	-4.0
B	2012	-4.0
B	2013	2.0

Input **DS_4** (see structure)

Id_1	Id_2	Me_1
A	2010	2.0
A	2011	9.0
A	2012	13.0
A	2013	26.0
A	2010Q1	2.0
A	2010Q2	-1.0
A	2010Q3	6.0
A	2010Q4	2.0

Example 1

```
DS_r := stock_to_flow ( DS_1 );
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1
A	2010M1/2010M12	2.0
A	2011M1/2011M12	5.0
A	2012M1/2012M12	-3.0
A	2013M1/2013M12	9.0
B	2010M1/2010M12	4.0
B	2011M1/2011M12	-8.0
B	2012M1/2012M12	0.0
B	2013M1/2013M12	6.0

Example 2

```
DS_r := stock_to_flow ( DS_2 );
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1
A	2010-12-31	2.0
A	2011-12-31	5.0
A	2012-12-31	-3.0
A	2013-12-31	9.0
B	2010-12-31	4.0
B	2011-12-31	-8.0
B	2012-12-31	0.0
B	2013-12-31	6.0

Example 3

```
DS_r := stock_to_flow ( DS_3 );
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1
A	2010	2.0
A	2011	5.0
A	2012	-3.0
A	2013	9.0
B	2010	4.0
B	2011	-8.0
B	2012	0.0
B	2013	6.0

Example 4

```
DS_r := stock_to_flow ( DS_4 );
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1
A	2010	2.0
A	2011	7.0
A	2012	4.0
A	2013	13.0
A	2010Q1	2.0
A	2010Q2	-3.0
A	2010Q3	7.0
A	2010Q4	-4.0

Time shift: *timeshift*

Syntax

timeshift (op **,** shiftNumber **)**

Input parameters

op	the operand
shiftNumber	the number of periods to be shifted

Examples of valid syntaxes

```
timeshift ( DS_1, 2 )
```

Semantics for scalar operations

This operator does not perform scalar operations.

Input parameters type

op

```
dataset { identifier < time > _ , identifier _* }
```

shiftNumber

```
integer
```

Result type

result

```
dataset { identifier < time > _ , identifier _* }
```

Additional Constraints

The operand dataset has an Identifier of type *time*, *date* or *time_period* and may have other Identifiers.

Behavior

This operator takes in input a Data Set of time series and, for each time series of the Data Set, shifts the reference time Identifier of a number of periods (of the time series) equal to the *shiftNumber* parameter. If *shiftNumber* is negative, the shift is in the past, otherwise it is in the future. For example, if the period of the time series is month and *shiftNumber* is -1 the reference time Identifier is shifted of two months in the past.

The operator can be applied only on Data Sets of time series and returns a Data Set of time series.

The result Data Set has the same Identifier, Measure and Attribute Components as the operand Data Set and contains the same time series as the operand, because the time series Identifiers (all the Identifiers except the reference time Identifier) are not changed.

The Attribute propagation rule is not applied.

As mentioned in the section “Behaviour of the Time Operators”, the operator is assumed to know which is the time Identifier as well as the period of each data point.

Examples

As described in the User Manual, the *time* data type is the intervening time between two time points and using the ISO 8601 standard it can be expressed through a start date and an end date separated by a slash at any precision. In the examples relevant to the *time* data type the precision is set at the level of month and the time format YYYY-MM/YYYY-MM is used.

Given:

- the operand dataset DS_1, which contains *annual* time series, where *Id_2* is the reference time Identifier of *time* type;
- the operand dataset DS_2, which contains *annual* time series, where *Id_2* is the reference time Identifier of *date* type (conventionally each period is identified by its last day);
- the operand dataset DS_3, which contains *annual* time series, where *Id_2* is the reference time Identifier of *time_period* type;
- and the operand dataset DS_4, which contains both *quarterly* and *annual* time series relevant to the same phenomenon “A”, where *Id_2* is the reference time Identifier of *time_period* type:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1
A	2010M1/2010M12	hello world
A	2011M1/2011M12	
A	2012M1/2012M12	say hello
A	2013M1/2013M12	he
B	2010M1/2010M12	hi, hello!
B	2011M1/2011M12	hi
B	2012M1/2012M12	
B	2013M1/2013M12	hello!

Input **DS_2** (see structure)

Id_1	Id_2	Me_1
A	2010-12-31	hello world
A	2011-12-31	
A	2012-12-31	say hello
A	2013-12-31	he
B	2010-12-31	hi, hello!
B	2011-12-31	hi
B	2012-12-31	
B	2013-12-31	hello!

Input **DS_3** (see structure)

Id_1	Id_2	Me_1
A	2010	hello world
A	2011	
A	2012	say hello
A	2013	he
B	2010	hi,hello!
B	2011	hi
B	2012	
B	2013	hello!

Input **DS_4** (see structure)

Id_1	Id_2	Me_1

A	2010	hello world
A	2011	
A	2012	say hello
A	2013	he
A	2010Q1	hi, hello!
A	2010Q2	hi
A	2010Q3	
A	2010Q4	hello!

Example 1

```
DS_r := timeshift ( DS_1 , -1 );
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1
A	2009M1/2009M12	hello world
A	2010M1/2010M12	
A	2011M1/2011M12	say hello
A	2012M1/2012M12	he
B	2009M1/2009M12	hi, hello!
B	2010M1/2010M12	hi
B	2011M1/2011M12	
B	2012M1/2012M12	hello!

Example 2

```
DS_r := timeshift ( DS_2 , 2 );
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1
A	2012-12-31	hello world
A	2013-12-31	
A	2014-12-31	say hello
A	2015-12-31	he
B	2012-12-31	hi, hello!
B	2013-12-31	hi
B	2014-12-31	
B	2015-12-31	hello!

Example 3

```
DS_r := timeshift ( DS_3 , 1 );
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1
A	2011	hello world
A	2012	
A	2013	say hello
A	2014	he
B	2011	hi,hello!
B	2012	hi
B	2013	
B	2014	hello!

Example 4

```
DS_r := timeshift( DS_4 , -1 );
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1
A	2009	hello world
A	2010	
A	2011	say hello
A	2012	he
A	2009Q4	hi, hello!
A	2010Q1	hi
A	2010Q2	
A	2010Q3	hello!

Time aggregation: *time_agg*

Syntax

time_agg (periodIndTo { , periodIndFrom } { , op } { , **first** | **last** } **)**

Input parameters

op	the scalar value, the Component or the Data Set to be converted. If not specified, then **time_agg** is used in combination within an aggregation operator

periodIndFrom	the source period indicator
periodIndTo	the target period indicator

Examples of valid syntaxes

```
sum ( DS group all time_agg ( “A” ) )
time_agg ( “A”, cast ( “2012Q1”, time_period , ”YYYY\Qq” ) )
time_agg(“M”, cast (“2012-12-23”, date, “YYYY-MM-DD”) )
time_agg(“M”, DS1)
ds_2 := ds_1[calc Me1 := time_agg(“M”,Me1)]
```

Semantics for scalar operations

The operator converts a *time*, *date* or *time_period* value from a smaller to a larger duration.

Input parameters type

op

```
dataset { identifier < time > _ , identifier _* }
| component<time>
| time
```

periodIndFrom, periodIndTo

```
duration
```

Result type

result

```
dataset { identifier < time > _ , identifier _* }
| component<time>
| time
```

Additional Constraints

If *op* is a Data Set then it has exactly one Identifier of type *time*, *date* or *time_period* and may have other Identifiers.

If time_agg is used in combination with an aggregation operator, op must not be specified, and the source dataset must have exactly one Identifier of type time, date or time_period (it may have additional Identifiers of other types).

It is only possible to convert smaller duration values to larger duration values (e.g. it is possible to convert *monthly* data to *annual* data but the contrary is not allowed).

Behaviour

The scalar version of this operator takes as input a *time*, *date* or *time_period* value, converts it to *periodIndTo* and returns a scalar of the corresponding type.

The Data Set version acts on a single Measure Data Set of type *time*, *date* or *time_period* and returns a Data Set having the same structure.

Finally, VTL also provides a component version, for use in combination with an aggregation operator, because the change of frequency requires an aggregation. In this case, the operator converts the period_indicator of the data points (e.g., convert *monthly* data to *annual* data).

On *time* type, the operator maps the input value into the comprising larger regular interval, whose duration is the one specified by the *periodIndTo* parameter.

On *date* type, the operator maps the input value into the comprising larger period, whose duration is the one specified by the *periodIndTo* parameter, which is conventionally represented either by the start or by the end date, according to the **first/last** parameter.

On *time_period* type, the operator maps the input value into the comprising larger time period specified by the *periodIndTo* parameter (the original period indicator is converted in the target one and the number of periods is adjusted correspondingly).

The input duration *periodIndFrom* is optional. In case of *time_period* Data Points, the input duration can be inferred from the internal representation of the value. In case of *time* or *date* types, it is inferred by the implementation.

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1
2010Q1	A	20
2010Q2	A	20
2010Q3	A	20
2010Q1	B	50
2010Q2	B	50
2010Q1	C	10
2010Q2	C	10

Example 1

```
DS_r := sum ( DS_1  group all time_agg ( "A" ));
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1
2010	A	60
2010	B	100
2010	C	20

Example 2

```
DS_r := time_agg ( "Q", cast ( "2012M01", time_period, "YYYY\MMM" ) );
```

returns “2012Q1”.

Example 3

The following example maps a *date* to quarter level, 2012 (end of the period):

```
DS_r := time_agg( “Q”, cast(“20120213”, date, “YYYYMMDD”), last );
```

and produces a *date* value corresponding to the *string* “20120331”.

Example 4

The following example maps a *date* to year level, 2012 (beginning of the period):

```
DS_r := time_agg(cast( “A”, “2012M1”, date, “YYYYMMDD”), first );
```

and produces a *date* value corresponding to the *string* “20120101”.

Actual time: *current_date*

Syntax

current_date()

Input parameters

None.

Examples of valid syntaxes

```
current_date()
```

Semantics for scalar operations

The operator **current_date** returns the current time as a *date* type.

Input parameters type

This operator has no input parameters.

Result type

result

```
date
```

Additional Constraints

None.

Behavior

The operator returns the current date.

Examples

Example 1

DS_r := current_date();

Example 2

DS_r := cast (current_date(), string, “YYYY.MM.DD”);

Days between two dates: *datediff*

Syntax

datediff (dateFrom , dateTo **)**

Input parameters

dateFrom	the starting date/time period
dateTo	the ending date/time period

Examples of valid syntaxes

```
datediff (2022Q1, 2023Q2)
datediff (2020-12-14,2021-04-20)
datediff (2021Q2, 2021-11-04)
ds2 := ds1[calc Me3 := datediff(Me1, Me2)]
```

Semantics for scalar operations

The operator datediff returns the number of days between two dates or time periods. The last day of the time period is assumed as the starting/ending date.

Input parameters type

dateFrom, dateTo:

```
component<time>
| time
```

Result type

result

```
component<integer>
| integer
```

Additional Constraints

None.

Behaviour

The scalar version of this operator takes as input two date or time_period values and returns a scalar integer value.

In the component version, that can be used in a calc clause, a new component of type integer is added to the dataset.

Examples

Given the dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1
G	2019-01-01	2020Q2
G	2019-07-01	2021Q1
T	2020-12-31	2021Q1

Example 1

```
DS_r:= DS_1[calc Me2 := datediff(Id_2, Me_1)];
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
G	2019-01-01	2020Q2	546
G	2019-07-01	2021Q1	273
T	2020-12-31	2021Q1	90

Example 2

```
dayofyear (2020-04-07);
```

returns 98.

Add a time unit to a dete: *dateadd*

Syntax

dateadd (op , shiftNumber, periodInd **)**

Input parameters

op	the operand
shiftNumber	the number of periods to be shifted
periodInd	the period indicator

Examples of valid syntaxes

```
dateadd (2022Q1, 5, “M”)
dateadd (2020-12-14, -3 , “Y”)
ds2 := ds1[calc Me2 := dateadd(Me1, 3, “W”)]
DS_r := dateadd(DS_1, 1, “M”)
```

Semantics for scalar operations

The operator dateadd returns the date resulting from adding (or subtracting) the given time units. The last day of the time period is assumed as the starting date. Please note that adding months to a given date returns the date plus integer months, adding years to a given date returns the date plus integer years; for years the “Y” is used. For example:

`dateadd (2020-02-10, 1, “M”)` gives `2020-03-10`
`dateadd (2020-02-10, 30, “D”)` gives `2020-03-11`
`dateadd (2020-02-10, 4, “W”)` gives `2020-03-09`
`dateadd (2020-02-10, 1, “Y”)` gives `2021-02-10`
`dateadd (2020-02-10, 365, “D”)` gives `2021-02-09`

Input parameters type

op:

```
dataset{ identifier < time > _ , identifier _* }
| component<time>
| time
```

shiftNumber:

```
Integer
```

periodInd:

```
Duration
```

Result type

result

```
dataset{ identifier < time > _ , identifier _* }
| component<time>
| time
```

Additional Constraints

None.

Behaviour

The scalar version of this operator takes as input one date or time_period value and returns a date adding/substracting the indicated number of time units.

In the component version, that can be used in a calc clause, a new component of type date is added to the dataset.

The operator can be applied also Data Sets; the result Data Set has the same Identifier, Measure and Attribute Components as the operand Data Set.

Examples

Given the dataset DS_1:

Input **DS_1** (see structure)

Id_1	Me_1
G	2019-01-01
H	2019-07-01
T	2020-12-31

Example 1

```
DS_r:= DS_1[calc Me2 := dateadd(Me_1, 2, "M")];
```

results in (see structure):

DS_r

Id_1	Me_1	Me_2
G	2019-01-01	2019-03-01
H	2019-07-01	2020-09-01
T	2020-12-31	2021-02-17

Example 2

```
dateadd (2021-11-04, -3, “W”) ;
```

returns 2021-10-14.

Extract time period from a date: *year, month, dayofmonth, dayofyear*

Syntax

{**year** | **month** | **dayofmonth** | **dayofyear**}[1] **(** op **)**

Input parameters

op	the input date/time period

Examples of valid syntaxes

```
year (2022Q1)
dayofyear (2020-12-14)
ds2 := ds1[calc Me2 := dayofmonth(Me1)]
```

Semantics for scalar operations

The operator **year** returns the year of the given date/time period. The operator **month** returns the month of the given date/time period (between 1 and 12). The operator **dayofmonth** returns the ordinal day within the month (between 1 and 31). The operator **dayofyear** returns the ordinal day within the year (between 1 and 366).

Input parameters type

op:

```
component<time>
| time
```

Result type

result

```
component<integer>
| integer
```

Additional Constraints

None.

Behaviour

The scalar version of this operators takes as input one date or time_period value and returns a integer value corresponding to the specified time period.

In the component version, that can be used in a calc clause, a new component of type integer is added to the dataset.

Examples

Given the dataset DS_1:

Input **DS_1** (see structure)

Id_1	Me_1
G	2019-01-01
H	2019-07-01
T	2020-12-31

Example 1

```
DS_r:= DS_1[calc Me2 := month(Me_1, 2, "M")];
```

results in (see structure):

DS_r

Id_1	Me_1	Me_2
G	2019-01-01	1
G	2019-07-01	7

T	2020-12-31	12

Example 2

```
datediff (2021Q2, 2021-11-04);
```

returns 127.

Number of days to duration: *daytoyear, daytomonth*

Syntax

{**daytoyear** | **daytomonth** }[1] **(** op **)**

Input parameters

op	an integer representing the number of days to transform

Examples of valid syntaxes

```
daytoyear (422)
daytomonth (146)
ds2 := ds1[calc Me2 := daytomonth(Me1)]
```

Semantics for scalar operations

The operator **daytoyear** returns a duration having the following mask: PYYDDDD. The operator **daytomonth** returns a duration having the following mask: PMMDDD.

Input parameters type

op:

```
component<integer>
| integer
```

Result type

result

```
component<duration>
| duration
```

Additional Constraints

None.

Behaviour

The scalar version of the **daytoyear** operator takes as input an integer representing t he number of days and returns the corresponding number of years and days; according to ISO 8601 Y = 365D.

The scalar version of the **daytomonth** operator takes as input an integer representing the numberoof days and returns the corresponding number of months and days; according to ISO 8601 M = 30D.

In the component version, that can be used in a calc clause, a new component of type duration is added to the dataset.

Examples

Given the dataset DS_1:

Input **DS_1** (see structure)

Id_1	Me_1
G	240
H	724
T	1056

Example 1

```
DS_r:= DS_1[calc Me2 := daytoyear(Me_1)];
```

results in (see structure):

DS_r

Id_1	Me_1	Me_2
G	240	P0Y240D
G	724	P1Y359D
T	1056	P2Y326D

Example 2

```
DS_r:= DS_1[calc Me2 := daytomonth(Me_1)];
```

results in (see structure):

DS_r

Id_1	Me_1	Me_2
G	240	P8M0D
G	724	P24M4D
T	1056	P35M6D

Example 2

```
daytoyear (782);
```

returns P2Y52D.

Example 3

```
daytomonth (134);
```

returns P4M14D.

Duration to number of days: *yeartoday, monthtoday*

Syntax

yeartoday (yearDuration **)**

monthtoday (monthDuration **)**

Input parameters

yearDuration	a duration having the following mask: *PYYDDDD*
monthDuration	a duration having the following mask: *PMMDDD*

Examples of valid syntaxes

```
yeartoday (P1Y)
monthtoday (P3M)
ds2 := ds1[calc Me2 := yeartoday(Me1)]
```

Semantics for scalar operations

The operators return an integer representing the number of days corresponding
to the given duration.

Input parameters type

op:

```
component<duration>
| duration
```

Result type

result

```
component<integer>
| integer
```

Additional Constraints

None.

Behaviour

The scalar version of the **yeartoday** operator takes as input a duration having the following mask: *PYYDDDD*; returns the corresponding number of years and days (according to ISO 8601 Y = 365D).

The scalar version of the monthtoday operator takes as input a duration having the following mask: *PMMDDD*; returns the corresponding number of months and days; according to ISO 8601 M = 30D).

In the component version, that can be used in a calc clause, a new component of type integer is added to the dataset.

Examples

Given the dataset DS_1:

Input **DS_1** (see structure)

Id_1	Me_1
G	P2Y230D
H	P1Y23D
T	P3Y152D

Example 1

```
DS_r:= DS_1[calc Me2 := yeartoday (Me_1)];
```

results in (see structure):

DS_r

Id_1	Me_1	Me_2
G	240	P0Y240D
G	724	P1Y359D
T	1056	P2Y326D

Example 2

```
yeartoday (P1Y20D);
```

returns 385.

Example 3

```
monthtoday (P3M10D);
```

returns 100.

VTL-ML - Set Operators

Union: *union*

Syntax

union (dsList **)**

dsList ::= ds { , ds }*

Input parameters

dsList	the list of Data Sets in the union

Examples of valid syntaxes

```
union (ds2, ds3)
```

Semantics for scalar operations

This operator does not perform scalar operations.

Input parameters type

ds

```
dataset
```

Result type

result

```
dataset
```

Additional Constraints

All the Data Sets in dsList have the same Identifier, Measure and Attribute Components.

Behaviour

The **union** operator implements the union of functions (i.e., Data Sets). The resulting Data Set has the same Identifier, Measure and Attribute Components of the operand Data Sets specified in the *dsList*, and contains the Data Points belonging to any of the operand Data Sets.

The operand Data Sets can contain Data Points having the same values of the Identifiers. To avoid duplications of Data Points in the resulting Data Set, those Data Points are filtered by choosing the Data Point belonging to the left most operand Data Set. For instance, let's assume that in **union** (ds1, ds2) the operand *ds1* contains a Data Point *dp1* and the operand *ds2* contains a Data Point *dp2* such that *dp1* has the same Identifiers values of *dp2*, then the resulting Data Set contains *dp1* only.

The operator has the typical behaviour of the "Behaviour of the Set operators" (see the section "Typical behaviours of the ML Operators").

The automatic Attribute propagation is not applied.

Examples

Given the operand datasets DS_1, DS_2 and DS_3:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Id_4	Me_1
2012	B	Total	Total	5
2012	G	Total	Total	2
2012	F	Total	Total	3

Input **DS_2** (see structure)

Id_1	Id_2	Id_3	Id_4	Me_1
2012	N	Total	Total	23
2012	S	Total	Total	5

Input **DS_3** (see structure)

Id_1	Id_2	Id_3	Id_4	Me_1
2012	B	Total	Total	23
2012	S	Total	Total	5

Example 1

```
DS_r := union( DS_1, DS_2 );
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Id_4	Me_1
2012	B	Total	Total	5
2012	G	Total	Total	2
2012	F	Total	Total	3
2012	N	Total	Total	23
2012	S	Total	Total	5

Example 2

```
DS_r := union ( DS_1, DS_3 );
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Id_4	Me_1
2012	B	Total	Total	5
2012	G	Total	Total	2
2012	F	Total	Total	3
2012	S	Total	Total	5

Intersection: *interesect*

Syntax

intersect (dsList)

dsList ::= ds { , ds }*

Input parameters

dsList	the list of Data Sets in the intersection

Examples of valid syntaxes

```
intersect (ds2, ds3)
```

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

ds

```
dataset
```

Result type

result

```
dataset
```

Additional Constraints

All the Data Sets in *dsList* have the same Identifier, Measure and Attribute Components.

Behavior

The **intersect** operator implements the intersection of functions (i.e., Data Sets). The resulting Data Set has the same Identifier, Measure and Attribute Components of the operand Data Sets specified in the *dsList*, and contains the Data Points belonging to all the operand Data Sets.

The operand Data Sets can contain Data Points having the same values of the Identifiers. To avoid duplications of Data Points in the resulting Data Set, those Data Points are filtered by choosing the Data Point belonging to the leftmost operand Data Set. For instance, let's assume that in **intersect (ds1, ds2)** the operand *ds1* contains a Data

Point *dp1* and the operand *ds2* contains a Data Point *dp2*, such that *dp1* has the same Identifiers values of *dp2*, then the resulting Data Set contains *dp1* only.

The operator has the typical behaviour of the “Behaviour of the Set operators” (see the section “Typical behaviours of the ML Operators”).

The automatic Attribute propagation is not applied.

Examples

Given the operand datasets DS_1 and DS_2:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Id_4	Me_1
2012	B	Total	Total	1
2012	G	Total	Total	2
2012	F	Total	Total	3

Input **DS_2** (see structure)

Id_1	Id_2	Id_3	Id_4	Me_1
2011	B	Total	Total	10
2012	G	Total	Total	2
2012	M	Total	Total	40

Example 1

```
DS_r := intersect(DS_1, DS_2);
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Id_4	Me_1
2012	G	Total	Total	2

Set difference: *setdiff*

Syntax

setdiff (ds1, ds2 **)**

Input parameters

ds1	the first Data Set in the difference (the minuend)
ds2	the second Data Set in the difference (the subtrahend)

Examples of valid syntaxes

```
setdiff (ds2, ds3)
```

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

ds1, ds2

`dataset`

Result type

result

`dataset`

Additional Constraints

The operand Data Sets have the same Identifier, Measure and Attribute Components.

Behavior

The operator implements the set difference of functions (i.e. Data Sets), interpreting the Data Points of the input Data Sets as the elements belonging to the operand sets, the minuend and the subtrahend, respectively. The operator returns one single Data Set, with the same Identifier, Measure and Attribute Components as the operand Data Sets, containing the Data Points that appear in the first Data Set but not in the second. In other words, for **setdiff (ds1, ds2)**, the resulting Dataset contains all the data points Data Point *dp1* of the operand *ds1*, such that there is no Data Point *dp2* of *ds2* having the same values for homonym Identifier Components.

The operator has the typical behaviour of the “Behaviour of the Set operators” (see the section “Typical behaviours of the ML Operators”).

The automatic Attribute propagation is not applied.

Examples

Given the operand datasets DS_1, DS_2, DS_3 and DS_4:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Id_4	Me_1
2012	B	Total	Total	10
2012	G	Total	Total	20
2012	F	Total	Total	30
2012	M	Total	Total	40
2012	I	Total	Total	50
2012	S	Total	Total	60

Input **DS_2** (see structure)

Id_1	Id_2	Id_3	Id_4	Me_1
2011	B	Total	Total	10
2012	G	Total	Total	20
2012	F	Total	Total	30
2012	M	Total	Total	40
2012	I	Total	Total	50
2012	S	Total	Total	60

Input **DS_3** (see structure)

Id_1	Id_2	Id_3	Me_1
R	M	2011	7
R	F	2011	10

R	T	2011	12

Input **DS_4** (see structure)

Id_1	Id_2	Id_3	Me_1
R	M	2011	7
R	F	2011	10

Example 1

```
DS_r := setdiff ( DS_1, DS_2 );
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Id_4	Me_1
2012	B	Total	Total	10.0

Example 2

```
DS_r := setdiff ( DS_3, DS_4 );
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Me_1
R	T	2011	12

Symmetric difference: *symdiff*

Syntax

symdiff (ds1, ds2 **)**

Input parameters

ds1	the first Data Set in the difference
ds2	the second Data Set in the difference

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

ds1, ds2

`dataset`

Result type

result

`dataset`

Additional Constraints

The operand Data Sets have the same Identifier, Measure and Attribute Components.

Behaviour

The operator implements the symmetric set difference between functions (i.e. Data Sets), interpreting the Data Points of the input Data Sets as the elements in the operand Sets. The operator returns one Data Set, with the same Identifier, Measure and Attribute Components as the operand Data Sets, containing the Data Points that appear in the first Data Set but not in the second and the Data Points that appear in the second Data Set but not in the first one.

Data Points are compared to one another by Identifier Components. For **symdiff (ds1, ds2)**, the resulting Data Set contains all the Data Points *dp1* contained in *ds1* for which there is no Data Point *dp2* in *ds2* with the same values for homonym Identifier components and all the Data Points *dp2* contained in *ds2* for which there is no Data Point *dp1* in *ds1* with the same values for homonym Identifier Components.

The operator has the typical behaviour of the “Behaviour of the Set operators” (see the section “Typical behaviours of the ML Operators”).

The automatic Attribute propagation is not applied.

Examples

Given the operand datasets DS_1 and DS_2:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Id_4	Me_1
2012	B	Total	Total	1
2012	G	Total	Total	2
2012	F	Total	Total	3
2012	M	Total	Total	4
2012	I	Total	Total	5
2012	S	Total	Total	6

Input **DS_2** (see structure)

Id_1	Id_2	Id_3	Id_4	Me_1
2011	B	Total	Total	1
2012	G	Total	Total	2
2012	F	Total	Total	3
2012	M	Total	Total	4
2012	I	Total	Total	5
2012	S	Total	Total	6

Example 1

```
DS_r := symdiff ( DS_1, DS_2 );
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Id_4	Me_1
2012	B	Total	Total	1.0

2011	B	Total	Total	1.0

VTL-ML - Hierarchical aggregation

Hierarchical roll-up: *hierarchy*

Syntax

hierarchy (op , hr { **condition** condComp { , condComp }* } { **rule** ruleComp } { mode } { input } { output })

mode ::= **non_null | non_zero | partial_null | partial_zero | always_null | always_zero**

input ::= **dataset | rule | rule_priority**

output ::= **computed | all**

Input parameters

op	the operand Data Set
hr	the hierarchical ruleset to be applied
condComp	*condComp* is a Component of *op* to be associated (in positional order) to the conditioning Value Domains or Variables defined in *hr* (if any)
ruleComp	*ruleComp* is the Identifier of *op* to be associated to the rule Value Domain or Variable defined in *hr*
mode	this parameter specifies how to treat the possible missing Data Points corresponding to the Code Items in the right side of a rule and which Data Points are produced in output. The meaning of the possible values of the parameter is explained below.
input	this parameter specifies the source of the values used as input of the hierarchical rules. The meaning of the possible values of the parameter is explained below.
output	this parameter specifies the content of the resulting Data Set. The meaning of the possible values of the parameter is explained below.

Examples of valid syntaxes

```
hierarchy ( DS1, HR1 rule Id_1 non_null all )
hierarchy ( DS2, HR2 condition Comp_1, Comp_2 rule Id_3 non_zero rule computed )
```

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

op

```
dataset { measure<number> _ }
```

hr

```
name<hierarchical>
```

condComp

```
name<component>
```

ruleComp

```
name<identifier>
```

Result type

result

```
dataset { measure<number> _ }
```

Additional Constraints

If *hr* is defined on Value Domains then it is mandatory to specify the *condition* (if any) and the *rule* parameters. Moreover, the Components specified as *condComp* and *ruleComp* must belong to the operand *op* and must take values on the Value Domains corresponding, in positional order, to the ones specified in the *condition* and *rule* parameter of *hr*.

If *hr* is defined on Variables, the specification of *condComp* and *ruleComp* is not needed, but they can be specified all the same if it is desired to show explicitly in the invocation which are the involved Components: in this case, the *condComp* and *ruleComp* must be the same and in the same order as the Variables specified in in the *condition* and *rule* signatures of *hr*.

Behaviour

The **hierarchy** operator applies the rules of *hr* to *op* as specified in the parameters. The operator returns a Data Set with the same Identifiers and the same Measure as *op*. The Attribute propagation rule is applied on the groups of Data Points which contribute to the same Data Points of the result.

The behaviours relevant to the different options of the input parameters are the following.

First, the parameter **input** is considered to determine the source of the Data Points used as input of the Hierarchy. The possible options of the parameter *input* and the corresponding behaviours are the following:

dataset	For each Rule of the Ruleset and for each item on the right hand side of the Rule, the operator takes the input Data Points exclusively from the operand *op*.
rule	For each Rule of the Ruleset and for each item on the right-hand side of the Rule: · if the item is not defined as the result (left-hand side) of another Rule, the current Rule takes the input Data Points from the operand *op* · if the item is defined as the result of another Rule, the current Rule takes the input Data Points from the computed output of such other Rule

rule_priority	For each Rule of the Ruleset and for each item on the right-hand side of the Rule: · if the item is not defined as the result (left-hand side) of another rule, the current Rule takes the input Data Points from the operand *op* · if the item is defined as the result of another Rule, then: > if an expected input Data Point exists in the computed output of such other Rule and its Measure is not NULL, then the current Rule takes such Data Point; > if an expected input Data Point does not exist in the computed output of such other Rule or its measure is NULL, then the current Rule takes the Data Point from op (if any) having the same values of the Identifiers;

if the parameter *input* is not specified then it is assumed to be *rule*.

Then the parameter *mode* is considered, to determine the behaviour for missing Data Points and for the Data Points to be produced in the output. The possible options of the parameter *mode* and the corresponding behaviours are the following:

non_null	the result Data Point is produced when its computed Measure value is not NULL (i.e., when no Data Point corresponding to the Code Items of the right side of the rule is missing or has NULL Measure value); in the calculation, the possible missing Data Points corresponding to the Code Items of the right side of the rule are considered existing and having a Measure value equal to NULL;
non_zero	the result Data Point is produced when its computed Measure value is not equal to 0 (zero); the possible missing Data Points corresponding to the Code Items of the right side of the rule are considered existing and having a Measure value equal to 0;
partial_null	the result Data Point is produced if at least one Data Point corresponding to the Code Items of the right side of the rule is found (whichever is its Measure value); the possible missing Data Points corresponding to the Code Items of the right side of the rule are considered existing and having a NULL Measure value;
partial_zero	the result Data Point is produced if at least one Data Point corresponding to the Code Items of the right side of the rule is found (whichever is its Measure value); the possible missing Data Points corresponding to the Code Items of the right side of the rule are considered existing and having a Measure value equal to 0 (zero);

always_null	the result Data Point is produced in any case; the possible missing Data Points corresponding to the Code Items of the right side of the rule are considered existing and having a Measure value equal to NULL;
always_zero	the result Data Point is produced in any case; the possible missing Data Points corresponding to the Code Items of the right side of the rule are considered existing and having a Measure value equal to 0 (zero);

If the parameter *mode* is not specified, then it is assumed to be *non_null*.

The following table summarizes the behaviour of the options of the parameter "*mode*":

OPTION of the MODE PARAMETER:	Missing Data Points are considered:	Null Data Points are considered:	Condition for evaluating the rule	Returned Data Points
Non_null	NULL	NULL	If all the involved Data Points are not NULL	Only not NULL Data Points (Zeros are returned too)
Non_zero	Zero	NULL	If at least one the involved Data Points is <> zero	Only not zero Data Points (NULL are returned too)
Partial_null	NULL	NULL	If at least the involved Data Points is not NULL	Data Points of any value (NULL or not NULL and zero too)
Partial_zero	Zero	NULL	If at least the involved Data Points is not NULL	Data Points of any value (NULL or not NULL and zero too)
Always_null	NULL	NULL	Always	Data Points of any value (NULL or not NULL and zero too)
Always_zero	Zero	NULL	Always	Data Points of any value (NULL or not NULL and zero too)

Finally the parameter *output* is considered, to determine the content of the resulting Data Set. The possible options of the parameter *output* and the corresponding behaviours are the following:

computed	the resulting Data Set contains only the set of Data Points computed according to the Ruleset
all	the resulting Data Set contains the union between the set of Data Points "R" computed according to the Ruleset and the set of Data Points of op that have different combinations of values for the Identifiers. In other words, the result is the outcome of the following (virtual)expression: **union (setdiff (op , R) , R)**

If the parameter *output* is not specified then it is assumed to be *computed*.

Examples

Given the following hierarchical ruleset:

```
define hierarchical ruleset HR_1 ( valuedomain rule VD_1 ) is
    A = J + K + L
    ; B = M + N + O
    ; C = P + Q
    ; D = R + S
    ; E = T + U + V
    ; F = Y + W + Z
    ; G = B + C
    ; H = D + E
    ; I = D + G
end hierarchical ruleset;
```

And given the operand dataset DS_1 (where *At_1* is viral and the propagation rule says that the alphabetic order prevails the NULL prevails on the alphabetic characters and the Attribute value for missing Data Points is assumed as NULL):

Input **DS_1** (see structure)

Id_1	Id_2	Me_1	At_1
2010	M	2	Dx
2010	N	5	Pz
2010	O	4	Pz
2010	P	7	Pz
2010	Q	-7	Pz
2010	S	3	Ay
2010	T	9	Bq
2010	U		Nj
2010	V	6	Ko

Example 1

```
DS_r := hierarchy ( DS_1, HR_1 rule Id_2 non_null );
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	At_1
2010	B	11	Dx
2010	C	0	Pz
2010	G	19	Dx

Example 2

```
DS_r := hierarchy ( DS_1, HR_1 rule Id_2 non_zero );
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	At_1
2010	B	11	Dx

2010	D	3	
2010	E		Bq
2010	G	11	Dx
2010	H		
2010	I	14	

Example 3

```
DS_r := hierarchy ( DS_1, HR_1 rule Id_2 partial_null );
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	At_1
2010	B	11	Dx
2010	C	0	Pz
2010	D		
2010	E		Bq
2010	G	11	Dx
2010	H		
2010	I		

VTL-ML - Aggregate and Analytic operators

The following table lists the operators that can be invoked in the Aggregate or in the Analytic invocations described below and their main characteristics.

Operator	Description	Allowed invocations	Type of resulting Measure	Type of operand Measures
count	number of Data Points	Aggregate Analytic	integer	any
min	minimum value of a set of values	Aggregate Analytic	any	any
max	maximum value of a set of values	Aggregate Analytic	any	any
median	median value of a set of numbers	Aggregate Analytic	number	number
sum	sum of a set of numbers	Aggregate Analytic	number	number
avg	average value of a set of numbers	Aggregate Analytic	number	number
stddev_pop	population standard deviation of a set of numbers	Aggregate Analytic	number	number
stddev_samp	sample standard deviation of a set of numbers	Aggregate Analytic	number	number

var_pop	population variance of a set of numbers	Aggregate Analytic	number	number
var_samp	sample variance of a set of numbers	Aggregate Analytic	number	number
first_value	first value in an ordered set of values	Analytic	any	any
last_value	last value in an ordered set of values	Analytic	any	any
lag	in an ordered set of Data Points it returns the value(s) taken from a Data Point at a given physical offset prior to the current Data Point	Analytic	any	any
lead	in an ordered set of Data Points it returns the value(s) taken from a Data Point at a given physical offset beyond the current Data Point	Analytic	any	any
rank	rank (order number) of a Data Point in an ordered set of Data Points	Analytic	integer	any
ratio_to_report	ratio of a value to the sum of a set of values	Analytic	number	number

Aggregate invocation

Syntax

- In a Data Set expression:

  aggregateOperator **(** firstOperand { , additionalOperand }* { groupingClause } **)**

- In a Component expression within an aggr clause:

  aggregateOperator **(** firstOperand { , additionalOperand }* **)** { groupingClause }

  aggregateOperator ::= **avg | count | max | median | min | stddev_pop | stddev_samp | sum | var_pop | var_samp**

  groupingClause ::=

  { **group by** groupingId {, groupingId}*
  | **group except** groupingId {, groupingId}*
  | **group all** conversionExpr }[1]
  { **having** havingCondition }

Input parameters

aggregateOperator	the keyword of the aggregate operator to invoke (e.g., avg, count, max…)
firstOperand	the first operand of the invoked aggregate operator (a Data Set for an invocation at Data Set level or a Component of the input Data Set for an invocation at Component level within an **aggr** operator or an **aggr** clause in a join operation)
additionalOperand	an additional operand (if any) of the invoked operator. The various operators can have a different number of parameters. The number of parameters, their types and if they are mandatory or optional depend on the invoked operator
groupingClause	the following alternative grouping options: · **group by**: the Data Points are grouped by the values of the specified Identifiers (*groupingId*). The Identifiers not specified are dropped in the result. · **group except**: the Data Points are grouped by the values of the Identifiers not specified as *groupingId*. The Identifiers specified as *groupingId* are dropped in the result. · **group all**: converts the values of an Identifier Component using *conversionExpr* and keeps all the resulting Identifiers.
groupingId	Identifier Component to be kept (in the **group by** clause) or dropped (in the **group except** clause).
conversionExpr	specifies a conversion operator (e.g., **time_agg**) to convert data from finer to coarser granularity. The conversion operator is applied on an Identifier of the operand Data Set *op*.

havingCondition	a condition (*boolean* expression) at component level, having only Components of the input Data Sets as operands (and possibly constants), to be fulfilled by the groups of Data Points: only groups for which *havingCondition* evaluates to TRUE appear in the result. The *havingCondition* refers to the groups specified through the *groupingClause*, therefore it must invoke aggregate operators (e.g. **avg**, **count**, **max**, …, see also the corresponding sections). A correct example of *havingCondition* is: *max(obs_value) < 1000*, while the condition *obs_value < 1000* is not a right havingCondition, because it refers to the values of single Data Points and not to the groups. The count operator is used in a *havingCondition* without parameters, e.g.: *sum (ds group by id1 having count () >= 10)* .

Examples of valid syntaxes

```
avg ( DS_1 )
avg ( DS_1 group by Id_1, Id_2  )
avg ( DS_1 group except  Id_1, Id_2  )
avg ( DS_1 group all time_agg ( "Q" ) )
```

Semantics for scalar operations

The aggregate operators cannot be applied to scalar values.

Input parameters type

firstOperand

```
dataset
| component
```

additionalOperand

```
see the type of the additional parameter (if any) of the invoked
aggregateOperator. The aggregate operators and their parameters are
described in the following sections.
```

groupingId

```
name < identifier >
```

conversionExpr

```
identifier
```

havingCondition

```
component < boolean >
```

Result type

result

```
dataset
| component
```

Additional Constraints

The Aggregate invocation cannot be nested in other Aggregate or Analytic invocations.

The aggregate operations at component level can be invoked within the **aggr** clause, both as part of a join operator and the **aggr** operator (see the parameter *aggrExpr* of those operators).

The basic scalar types of *firstOperand* and *additionalOperand* (if any) must be compliant with the specific basic scalar types required by the invoked operator (the required basic scalar types are described in the table at the beginning of this chapter and in the sections of the various operators below).

The *conversionExpr* parameter applies just one conversion operator to just one Identifier belonging to the input Data Set. The basic scalar type of the Identifier must be compatible with the basic scalar type of the conversion operator.

If the grouping clause is omitted, then all the input Data Points are aggregated in a single group and the clause returns a Data Set that contains a single Data Point and has no Identifiers.

Behaviour

The *aggregateOperator* is applied as usual to all the measures of the *firstOperand* Data Set (if invoked at Data Set level) or to the *firstOperand* Component of the input Data Set (if invoked at Component level). In both cases, the operator calculates the required aggregated values for groups of Data Points of the input Data Set. The groups of Data Points to be aggregated are specified through the groupingClause, which allows the following alternative options.

group by	the Data Points are grouped by the values of the specified Identifiers. The Identifiers not specified are dropped in the result.
group except	the Data Points are grouped by the values of the Identifiers not specified in the clause. The specified Identifiers are dropped in the result.
group all	converts an Identifier Component using *conversionExpr* and keeps all the Identifiers.

The **having** clause is used to filter groups in the result by means of an aggregate condition evaluated on the single groups (for example the minimum number of rows in the group).

If no grouping clause is specified, then all the input Data Points are aggregated in a single group and the operator returns a Data Set that contains a single Data Point and has no Identifiers.

For the invocation at Data Set level, the resulting Data Set has the same Measures as the operand. For the invocation at Component level, the resulting Data Set has the Measures explicitly calculated (all the other Measures are dropped because no aggregation behaviour is specified for them).

For invocation at Data Set level, the Attribute propagation rule is applied. For invocation at Component level, the Attributes calculated within the *aggr* clause are maintained in the result; for all the other Attributes that are defined as **viral**, the Attribute propagation rule is applied (for the semantics, see the Attribute Propagation Rule section in the User Manual).

As mentioned, the Aggregate invocation at component level can be done within the **aggr** clause, both as part of a Join operator and the **aggr** operator (see the parameter *aggrExpr* of those operators), therefore, for a better comprehension fo the behaviour at Component level, see also those operators.

Examples

Given the operand datasets DS_1 and DS_2:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Me_1	At_1
2010	E	XX	20	
2010	B	XX	1	H

2010	R	XX	1	A
2010	F	YY	23	
2011	E	XX	20	P
2011	B	ZZ	1	N
2011	R	YY	-1	P
2011	F	XX	20	Z
2012	L	ZZ	40	P
2012	E	YY	30	P

Input **DS_2** (see structure)

Id_1	Id_2	Id_3	Me_1	At_1
2010	E	XX	20	
2010	B	XX	1	H
2010	R	XX	1	A
2010	F	YY	23	
2011	E	XX	20	P
2011	B	ZZ	1	N
2011	R	YY	-1	P
2011	F	XX	20	Z
2012	L	ZZ	40	P
2012	E	YY	30	P

Example 1

```
DS_r := avg ( DS_1 group by Id_1 );
```

results in (see structure):

DS_r

Id_1	Me_1
2010	11.25
2011	10.0
2012	35.0

Example 2

```
DS_r := sum ( DS_1 group by Id_1, Id_3 );
```

results in (see structure):

DS_r

Id_1	Id_3	Me_1
2010	XX	22.0
2010	YY	23.0

2011	XX	40.0
2011	ZZ	1.0
2011	YY	-1.0
2012	ZZ	40.0
2012	YY	30.0

Example 3

```
DS_r := avg ( DS_1 );
```

results in (see structure):

DS_r

Me_1
15.5

Example 4

```
DS_r := DS_1 [ aggr Me_2 := max ( Me_1 ) , Me_3 := min ( Me_1 ) group by Id_1 ];
/*provided that At_1 is viral and the first letter in alphabetic order prevails
and NULL prevails on all the other characters.*/
```

results in (see structure):

DS_r

Id_1	Me_2	Me_3	At_1
2010	23	1	
2011	20	-1	N
2012	40	30	P

Note: the first example can be rewritten equivalently in the following forms:

DS_r := avg (DS_1 group except Id_2, Id_3)

DS_r := avg (DS_1#Me_1 group by Id_1)

Analytic invocation

Syntax

analyticOperator **(** firstOperand { , additionalOperand }* **over (** analyticClause **))**

analyticOperator ::= **avg | count | max | median | min | stddev_pop | stddev_samp | sum | var_pop | var_samp | first_value | lag | last_value | lead | rank | ratio_to_report**

analyticClause ::= { partitionClause } { orderClause } { windowClause }

partitionClause ::= **partition by** identifier { , identifier }*

orderClause ::= **order by** component { **asc | desc** } { , component { **asc | desc** } }*

windowClause ::= { **data points | range** }[1] **between** limitClause **and** limitClause

limitClause ::= { num **preceding** | num **following** | **current data point** | **unbounded preceding** | **unbounded following** }[1]

Input parameters

analyticOperator	the keyword of the analytic operator to invoke (e.g., **avg, count, max**…)
firstOperand	the first operand of the invoked analytic operator (a Data Set for an invocation at Data Set level or a Component of the input Data Set for an invocation at Component level within a **calc** operator or a **calc** clause in a join operation)
additionalOperand	an additional operand (if any) of the invoked operator. The various operators can have a different number of parameters. The number of parameters, their types and if they are mandatory or optional depend on the invoked operator
analyticClause	clause that specifies the analytic behaviour
partitionClause	clause that specifies how to partition Data Points in groups to be analysed separately. The input Data Set is partitioned according to the values of one or more Identifier Components. If the clause is omitted, then the Data Set is partitioned by the Identifier Components that are not specified in the *orderClause*.
orderClause	clause that specifies how to order the Data Points. The input Data Set is ordered according to the values of one or more Components, in ascending order if **asc** is specified, in descending order if **desc** is specified, by default in ascending order if the **asc** and **desc** keywords are omitted.
windowClause	clause that specifies how to apply a sliding window on the ordered Data Points. The keyword **data points** means that the sliding window includes a certain number of Data Points before and after the current Data Point in the order given by the *orderClause*. The keyword **range** means that the sliding windows includes all the Data Points whose values are in a certain range in respect to the value, for the current Data Point, of the Measure which the analytic is applied to.

limitCause	clause that can specify either the lower or the upper boundaries of the sliding window. Each boundary is specified in relationship either to the whole partition or to the current data point under analysis by using the following keywords: * **unbounded preceding** means that the sliding window starts at the first Data Point of the partition (it make sense only as the first limit of the window) * **unbounded following** indicates that the sliding window ends at the last Data Point of the partition (it makes sense only as the second limit of the window) * **current data point** specifies that the window starts or ends at the current Data Point. * *num* **preceding** specifies either the number of **data points** to consider preceding the current data point in the order given by the *orderClause* (when **data points** is specified in the window clause), or the maximum difference to consider, as for the Measure which the analytic is applied to, between the value of the current Data Point and the generic other Data Point (when **range** is specified in the windows clause). num following specifies either the number of data points to consider following the current data point in the order given by the orderClause (when data points is specified in the window clause), or the maximum difference to consider, as for the Measure which the analytic is applied to, between the values of the generic other Data Point and the current Data Point (when range is specified in the windows clause). If the whole *windowClause* is omitted then the default is **data points between unbounded preceding and unbounded following**.
identifier	an Identifier Component of the input Data Set
component	a Component of the input Data Set
num	a scalar number

Examples of valid syntaxes

```
sum ( DS_1 over ( partition by  Id_1  order by  Id_2  ) )
sum ( DS_1 over ( order by  Id_2  ) )
avg ( DS_1 over ( order by  Id_1  data points between 1 preceding and 1 following ) )
DS_1 [ calc M1 := sum ( Me_1 over ( order by  Id_1  ) ) ]
```

Semantics for scalar operations

The analytic operators cannot be applied to scalar values.

Input parameters type

firstOperand

```
dataset
| component
```

additionalOperand

```
see the type of the additional parameter (if any) of the invoked
aggregateOperator. The aggregate operators and their parameters are
described in the following sections.
```

groupingId

```
name < identifier >
```

conversionExpr

```
identifier
```

havingCondition

```
component < boolean >
```

Result type

result

```
dataset
| component
```

Additional Constraints

The analytic invocation cannot be nested in other Aggregate or Analytic invocations.

The analytic operations at component level can be invoked within the **calc** clause, both as part of a Join operator and the **calc** operator (see the parameter *calcExpr* of those operators).

The basic scalar types of *firstOperand* and *additionalOperand* (if any) must be compliant with the specific basic scalar types required by the invoked operator (the required basic scalar types are described in the table at the beginning of this chapter and in the sections of the various operators below).

Behaviour

The analytic Operator is applied as usual to all the Measures of the input Data Set (if invoked at Data Set level) or to the specified Component of the input Data Set (if invoked at Component level). In both cases, the operator calculates the desired output values for each Data Point of the input Data Set.

The behaviour of the analytic operations can be procedurally described as follows:

- The Data Points of the input Data Set are first partitioned (according to *partitionBy*) and then ordered (according to *orderBy*).
- The operation is performed for each Data Point (named “current Data Point”) of the input Data Set. For each input Data Point, one output Data Point is returned, having the same values of the Identifiers. The analytic operator is applied to a “window” which includes a set of Data Points of the input Data Set and returns the values of the Measure(s) of the output Data Point.
    - If *windowClause* is not specified, then the set of Data Points which contribute to the analytic operation is the whole partition which the current Data Point belongs to
    - If *windowClause* is specified, then the set of Data Points is the one specified by *windowClause* (see *windowsClause* and *LimitClause* explained above).

For the invocation at Data Set level, the resulting Data Set has the same Measures as the input Data Set *firstOperand*. For the invocation at Component level, the resulting Data Set has the Measures of the input Data Set plus the Measures explicitly calculated through the **calc** clause.

For the invocation at Data Set level, the Attribute propagation rule is applied. For invocation at Component level, the Attributes calculated within the *calc* clause are maintained in the result; for all the other Attributes that are defined as viral, the Attribute propagation rule is applied (for the semantics, see the Attribute Propagation Rule section in the User Manual).

As mentioned, the Analytic invocation at component level can be done within the **calc** clause, both as part of a Join operator and the **calc** operator (see the parameter *aggrCalc* of those operators), therefore, for a better comprehension fo the behaviour at Component level, see also those operators.

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Me_1
2010	E	XX	5
2010	B	XX	-3
2010	R	XX	9
2010	E	YY	13
2011	E	XX	11
2011	B	ZZ	7
2011	E	YY	-1
2011	F	XX	0
2012	L	ZZ	-2
2012	E	YY	3

Example 1

```
DS_r := sum ( DS_1 over ( order by Id_1, Id_2, Id_3 data points between 1 preceding and 1 fo
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Me_1
2010	B	XX	2
2010	E	XX	15
2010	E	YY	27
2010	R	XX	29
2011	B	ZZ	27
2011	E	XX	17
2011	E	YY	10
2011	F	XX	2
2012	E	YY	1
2012	L	ZZ	1

Counting the number of data points: *count*

Syntax

- Aggregate syntax

in a Data Set expression	**count (** dataset { groupingClause } **)**
in a Component expression within an aggr clause	**count (** component **)** { groupingClause }
in a Data Set expression	**count ( )**

- Analytic syntax

in a Data Set expression	**count (** dataset **over(** analyticClause **))**
in a Component expression within a calc clause	**count (** component **over(** analyticClause **))**

Input parameters

dataset	the operand Data Set
component	the operand Component
groupingClause	see Aggregate invocation
analyticClause	see Analytic invocation

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

dataset

```
dataset
```

component

```
component
```

Result type

result

```
dataset { measure<integer> int_var }
| component<integer>
```

Additional Constraints

None.

Behaviour

The operator returns the number of the input Data Points. For other details, see Aggregate and Analytic invocations.

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Me_1
2011	A	XX	iii

2011	A	YY	jjj
2011	B	YY	iii
2012	A	XX	kkk
2012	B	YY	iii

Example 1

```
DS_r := count ( DS_1 group by Id_1 );
```

results in (see structure):

DS_r

Id_1	int_var
2011	3
2012	2

Example 2

```
DS_r := count ( DS_1 group by Id_1 having count() > 2 );
```

results in (see structure):

DS_r

Id_1	int_var
2011	3

Minimun value: *min*

Syntax

- Aggregate syntax

in a Data Set expression	**min (** dataset { groupingClause } **)**
in a Component expression within an aggr clause	**min (** component **)** { groupingClause }

- Analytic syntax

in a Data Set expression	**min (** dataset **over (** analyticClause **))**
in a Component expression within a calc clause	**min (** component **over (** analyticClause **))**

Input parameters

dataset	the operand Data Set
component	the operand Component
groupingClause	see Aggregate invocation
analyticClause	see Analytic invocation

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

dataset

```
dataset
```

component

```
component
```

Result type

result

```
dataset
| component
```

Additional Constraints

None.

Behaviour

The operator returns the minimum value of the input values. For other details, see Aggregate and Analytic invocations.

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Me_1
2011	A	XX	3
2011	A	YY	5
2011	B	YY	7
2012	A	XX	2
2012	B	YY	4

Example 1

```
DS_r := min ( DS_1 group by Id_1 );
```

results in (see structure):

DS_r

Id_1	Me_1
2011	3
2012	2

Maximum value: *max*

Syntax

- Aggregate syntax

in a Data Set expression	**max (** dataset { groupingClause } **)**

in a Component expression within an aggr clause	**max (** component **)** { groupingClause }

- Analytic syntax

in a Data Set expression	**max (** dataset **over (** analyticClause **))**
in a Component expression within a calc clause	**max (** component **over (** analyticClause **))**

Input parameters

dataset	the operand Data Set
component	the operand Component
groupingClause	see Aggregate invocation
analyticClause	see Analytic invocation

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

dataset

```
dataset
```

component

```
component
```

Result type

result

```
dataset
| component
```

Additional Constraints

None.

Behaviour

The operator returns the maximum of the input values. For other details, see Aggregate and Analytic invocations.

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Me_1
2011	A	XX	3
2011	A	YY	5
2011	B	YY	7
2012	A	XX	2
2012	B	YY	4

Example 1

```
DS_r := max ( DS_1 group by Id_1 );
```

results in (see structure):

DS_r

Id_1	Me_1
2011	7
2012	4

Median value: *median*

Syntax

- Aggregate syntax

in a Data Set expression	**median (** dataset { groupingClause } **)**
in a Component expression within an aggr clause	**median (** component **)** { groupingClause }

- Analytic syntax

in a Data Set expression	**median (** dataset **over (** analyticClause **))**
in a Component expression within a calc clause	**median (** component **over (** analyticClause **))**

Input parameters

dataset	the operand Data Set
component	the operand Component
groupingClause	see Aggregate invocation
analyticClause	see Analytic invocation

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

dataset

```
dataset { measure<number> _+ }
```

component

```
component<number>
```

Result type

result

```
dataset { measure<number> _+ }
| component<number>
```

Additional Constraints

None.

Behaviour

The operator returns the median value of the input values. For other details, see Aggregate and Analytic invocations.

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Me_1
2011	A	XX	3
2011	A	YY	5
2011	B	YY	7
2012	A	XX	2
2012	B	YY	4

Example 1

```
DS_r := median ( DS_1 group by Id_1 );
```

results in (see structure):

DS_r

Id_1	Me_1
2011	5.0
2012	3.0

Sum: *sum*

Syntax

- Aggregate syntax

in a Data Set expression	**sum (** dataset { groupingClause } **)**
in a Component expression within an aggr clause	**sum (** component **)** { groupingClause }

- Analytic syntax

in a Data Set expression	**sum (** dataset **over (** analyticClause **))**
in a Component expression within a calc clause	**sum (** component **over (** analyticClause **))**

Input parameters

dataset	the operand Data Set
component	the operand Component
groupingClause	see Aggregate invocation
analyticClause	see Analytic invocation

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

dataset

```
dataset { measure<number> _+ }
```

component

```
component<number>
```

Result type

result

```
dataset { measure<number> _+ }
| component<number>
```

Additional Constraints

None.

Behaviour

The operator returns the sum of the input values. For other details, see Aggregate and Analytic invocations.

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Me_1
2011	A	XX	3
2011	A	YY	5
2011	B	YY	7
2012	A	XX	2
2012	B	YY	4

Example 1

```
DS_r := sum ( DS_1 group by Id_1 );
```

results in (see structure):

DS_r

Id_1	Me_1
2011	15.0
2012	6.0

Average value: *avg*

Syntax

- Aggregate syntax

in a Data Set expression	**avg (** dataset { groupingClause } **)**
in a Component expression within an aggr clause	**avg (** component **)** { groupingClause }

- Analytic syntax

in a Data Set expression	**avg (** dataset **over (** analyticClause **))**
in a Component expression within a calc clause	**avg (** component **over (** analyticClause **))**

Input parameters

dataset	the operand Data Set
component	the operand Component
groupingClause	see Aggregate invocation
analyticClause	see Analytic invocation

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

dataset

```
dataset { measure<number> _+ }
```

component

```
component<number>
```

Result type

result

```
dataset { measure<number> _+ }
| component<number>
```

Additional Constraints

None.

Behaviour

The operator returns the average of the input values. For other details, see Aggregate and Analytic invocations.

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Me_1
2011	A	XX	3
2011	A	YY	5
2011	B	YY	7
2012	A	XX	2
2012	B	YY	4

Example 1

```
DS_r := avg ( DS_1 group by Id_1 );
```

results in (see structure):

DS_r

Id_1	Me_1
2011	5.0
2012	3.0

Population standard deviation: *stddev_pop*

Syntax

- Aggregate syntax

in a Data Set expression	**stddev_pop (** dataset { groupingClause } **)**
in a Component expression within an aggr clause	**stddev_pop (** component **)** { groupingClause }

- Analytic syntax

in a Data Set expression	**stddev_pop (** dataset **over (** analyticClause **))**
in a Component expression within a calc clause	**stddev_pop (** component **over (** analyticClause **))**

Input parameters

dataset	the operand Data Set
component	the operand Component
groupingClause	see Aggregate invocation
analyticClause	see Analytic invocation

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

dataset

```
dataset { measure<number> _+ }
```

component

```
component<number>
```

Result type

result

```
dataset { measure<number> _+ }
| component<number>
```

Additional Constraints

None.

Behaviour

The operator returns the “population standard deviation” of the input values. For other details, see Aggregate and Analytic invocations.

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Me_1
2011	A	XX	3
2011	A	YY	5
2011	B	YY	7
2012	A	XX	2
2012	B	YY	4

Example 1

```
DS_r := stddev_pop ( DS_1 group by Id_1 );
```

results in (see structure):

DS_r

Id_1	Me_1
2011	1.632993
2012	1.0

Sample standard deviation: *stddev_samp*

Syntax

- Aggregate syntax

in a Data Set expression	**stddev_samp (** dataset { groupingClause } **)**
in a Component expression within an aggr clause	**stddev_samp (** component **)** { groupingClause }

- Analytic syntax

in a Data Set expression	**stddev_samp (** dataset **over (** analyticClause **))**
in a Component expression within a calc clause	**stddev_samp (** component **over (** analyticClause **))**

Input parameters

dataset	the operand Data Set
component	the operand Component
groupingClause	see Aggregate invocation
analyticClause	see Analytic invocation

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

dataset

```
dataset { measure<number> _+ }
```

component

```
component<number>
```

Result type

result

```
dataset { measure<number> _+ }
| component<number>
```

Additional Constraints

None.

Behaviour

The operator returns the “sample standard deviation” of the input values. For other details, see Aggregate and Analytic invocations.

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Me_1
2011	A	XX	3
2011	A	YY	5
2011	B	YY	7
2012	A	XX	2
2012	B	YY	4

Example 1

```
DS_r := stddev_samp ( DS_1 group by Id_1 );
```

results in (see structure):

DS_r

Id_1	Me_1
2011	2.0
2012	1.414214

Population variance: *var_pop*

Syntax

- Aggregate syntax

in a Data Set expression	**var_pop (** dataset { groupingClause } **)**
in a Component expression within an aggr clause	**var_pop (** component **)** { groupingClause }

- Analytic syntax

in a Data Set expression	**var_pop (** dataset **over (** analyticClause **))**
in a Component expression within a calc clause	**var_pop (** component **over (** analyticClause **))**

Input parameters

dataset	the operand Data Set
component	the operand Component
groupingClause	see Aggregate invocation
analyticClause	see Analytic invocation

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

dataset

```
dataset { measure<number> _+ }
```

component

```
component<number>
```

Result type

result

```
dataset { measure<number> _+ }
| component<number>
```

Additional Constraints

None.

Behaviour

The operator returns the “population variance” of the input values. For other details, see Aggregate and Analytic invocations.

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Me_1
2011	A	XX	3
2011	A	YY	5
2011	B	YY	7
2012	A	XX	2
2012	B	YY	4

Example 1

```
DS_r := var_pop ( DS_1 group by Id_1 );
```

results in (see structure):

DS_r

Id_1	Me_1
2011	2.666667
2012	1.0

Sample variance: *var_samp*

Syntax

- Aggregate syntax

in a Data Set expression	**var_samp (** dataset { groupingClause } **)**
in a Component expression within an aggr clause	**var_samp (** component **)** { groupingClause }

- Analytic syntax

in a Data Set expression	**var_samp (** dataset **over (** analyticClause **))**
in a Component expression within a calc clause	**var_samp (** component **over (** analyticClause **))**

Input parameters

dataset	the operand Data Set
component	the operand Component
groupingClause	see Aggregate invocation
analyticClause	see Analytic invocation

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

dataset

```
dataset { measure<number> _+ }
```

component

```
component<number>
```

Result type

result

```
dataset { measure<number> _+ }
| component<number>
```

Additional Constraints

None.

Behaviour

The operator returns the sample variance of the input values. For other details, see Aggregate and Analytic invocations.

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Me_1
2011	A	XX	3
2011	A	YY	5
2011	B	YY	7
2012	A	XX	2
2012	B	YY	4

Example 1

```
DS_r := var_samp ( DS_1 group by Id_1 );
```

results in (see structure):

DS_r

Id_1	Me_1
2011	4.0
2012	2.0

First value: *first_value*

Syntax

in a Data Set expression	**first_value (** dataset **over (** analyticClause **))**
in a Component expression within a calc clause	**first_value (** component **over (** analyticClause **))**

Input parameters

dataset	the operand Data Set
component	the operand Component
analyticClause	see Analytic invocation

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

dataset

```
dataset { measure<scalar> _+ }
```

component

```
component<scalar>
```

Result type

result

```
dataset
| component<scalar>
```

Additional Constraints

The Aggregate invocation is not allowed.

Behaviour

The operator returns the first value (in the value order) of the set of Data Points that belong to the same analytic window as the current Data Point.

When invoked at Data Set level, it returns the first value for each Measure of the input Data Set. The first value of different Measures can result from different Data Points. When invoked at Component level, it returns the first value of the specified Component.

For other details, see Analytic invocation.

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Me_1	Me_2
A	XX	1993	3	1
A	XX	1994	4	9
A	XX	1995	7	5
A	XX	1996	6	8
A	YY	1993	9	3
A	YY	1994	5	4
A	YY	1995	10	2
A	YY	1996	2	7

Example 1

```
DS_r := first_value ( DS_1 over ( partition by Id_1, Id_2 order by Id_3 data points between
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Me_1	Me_2
A	XX	1993	3	1
A	XX	1994	3	1
A	XX	1995	4	9
A	XX	1996	7	5
A	YY	1993	9	3
A	YY	1994	9	3
A	YY	1995	5	4
A	YY	1996	10	2

Last value: *last_value*

Syntax

in a Data Set expression	**last_value (** dataset **over (** analyticClause **))**
in a Component expression within a calc clause	**last_value (** component **over (** analyticClause **))**

Input parameters

dataset	the operand Data Set
component	the operand Component
analyticClause	see Analytic invocation

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

dataset

```
dataset { measure<scalar> _+ }
```

component

```
component<scalar>
```

Result type

result

```
dataset
| component<scalar>
```

Additional Constraints

The Aggregate invocation is not allowed.

Behaviour

The operator returns the last value (in the value order) of the set of Data Points that belong to the same analytic window as the current Data Point.

When invoked at Data Set level, it returns the last value for each Measure of the input Data Set. The last value of different Measures can result from different Data Points. When invoked at Component level, it returns the last value of the specified Component.

For other details, see Analytic invocation.

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Me_1	Me_2
A	XX	1993	3	1
A	XX	1994	4	9
A	XX	1995	7	5
A	XX	1996	6	8

A	YY	1993	9	3
A	YY	1994	5	4
A	YY	1995	10	2
A	YY	1996	2	7

Example 1

```
DS_r := last_value ( DS_1 over ( partition by Id_1, Id_2 order by Id_3 data points between 1
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Me_1	Me_2
A	XX	1993	4	9
A	XX	1994	7	5
A	XX	1995	6	8
A	XX	1996	6	8
A	YY	1993	5	4
A	YY	1994	10	2
A	YY	1995	2	7
A	YY	1996	2	7

Lag: *lag*

Syntax

in a Data Set expression	**lag (** dataset {, offset {, defaultValue } } **over (** { partitionClause } orderClause **))**
in a Component expression within a calc clause	**lag (** component {, offset {, defaultValue } } **over (** { partitionClause } orderClause **))**

Input parameters

dataset	the operand Data Set
component	the operand Component
offset	the relative position prior to the current Data Point
defaultValue	the value returned when the *offset* goes outside of the partition
partitionClause	see Analytic invocation
orderClause	see Analytic invocation

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

dataset

```
dataset
```

component

```
component
```

offset

```
integer [ value > 0 ]
```

default value

```
scalar
```

Result type

result

```
dataset
| component
```

Additional Constraints

The Aggregate invocation is not allowed.

The *windowClause* of the Analytic invocation syntax is not allowed.

Behaviour

In the ordered set of Data Points of the current partition, the operator returns the value(s) taken from the Data Point at the specified physical *offset* prior to the current Data Point.

If *defaultValue* is not specified then the value returned when the offset goes outside the partition is NULL.

For other details, see Analytic invocation.

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Me_1	Me_2
A	XX	1993	3	1
A	XX	1994	4	9
A	XX	1995	7	5
A	XX	1996	6	8
A	YY	1993	9	3
A	YY	1994	5	4
A	YY	1995	10	2
A	YY	1996	2	7

Example 1

```
DS_r := lag ( DS_1 , 1 over ( partition by Id_1 , Id_2 order by Id_3 ) );
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Me_1	Me_2

A	XX	1993		
A	XX	1994	3	1
A	XX	1995	4	9
A	XX	1996	7	5
A	YY	1993		
A	YY	1994	9	3
A	YY	1995	5	4
A	YY	1996	10	2

Lead: *lead*

Syntax

in a Data Set expression	**lead (** dataset {, offset {, defaultValue } } **over (** { partitionClause } orderClause **))**
in a Component expression within a calc clause	**lead (** component {, offset {, defaultValue } } **over (** { partitionClause } orderClause **))**

Input parameters

dataset	the operand Data Set
component	the operand Component
offset	the relative position beyond the current Data Point
defaultValue	the value returned when the *offset* goes outside of the partition
partitionClause	see Analytic invocation
orderClause	see Analytic invocation

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

dataset

```
dataset
```

component

```
component
```

offset

```
integer [ value > 0 ]
```

default value

```
scalar
```

Result type

result

```
dataset
| component
```

Additional Constraints

The Aggregate invocation is not allowed.

The *windowClause* of the Analytic invocation syntax is not allowed.

Behaviour

In the ordered set of Data Points of the current partition, the operator returns the value(s) taken from the Data Point at the specified physical *offset* beyond the current Data Point.

If *defaultValue* is not specified, then the value returned when the offset goes outside the partition is NULL.

For other details, see Analytic invocation.

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Me_1	Me_2
A	XX	1993	3	1
A	XX	1994	4	9
A	XX	1995	7	5
A	XX	1996	6	8
A	YY	1993	9	3
A	YY	1994	5	4
A	YY	1995	10	2
A	YY	1996	2	7

Example 1

```
DS_r := lead ( DS_1 , 1 over ( partition by Id_1 , Id_2 order by Id_3 ) );
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Me_1	Me_2
A	XX	1993	4	9
A	XX	1994	7	5
A	XX	1995	6	8
A	XX	1996		
A	YY	1993	5	4
A	YY	1994	10	2
A	YY	1995	2	7
A	YY	1996		

Rank: *rank*

Syntax

in a Component expression within a calc clause	**rank (over (** { partitionClause } orderClause **))**

Input parameters

partitionClause	see Analytic invocation
orderClause	see Analytic invocation

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

dataset

```
dataset
```

component

```
component
```

Result type

result

```
dataset { measure<integer> int_var }
| component<integer>
```

Additional Constraints

The invocation at Data Set level is not allowed.

The Aggregate invocation is not allowed.

The *windowClause* of the Analytic invocation syntax is not allowed.

Behaviour

The operator returns an order number (rank) for each Data Point, starting from the number 1 and following the order specified in the *orderClause*. If some Data Points are in the same order according to the specified *orderClause*, the same order number (rank) is assigned and a gap appears in the sequence of the assigned ranks (for example, if four Data Points have the same rank 5, the following assigned rank would be 9).

For other details, see Analytic invocation.

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Me_1	Me_2
A	XX	2000	3	1
A	XX	2001	4	9
A	XX	2002	7	5
A	XX	2003	6	8
A	YY	2000	9	3

A	YY	2001	5	4
A	YY	2002	10	2
A	YY	2003	5	7

Example 1

```
DS_r := DS_1 [ calc Me_2 := rank ( over ( partition by Id_1 , Id_2 order by Me_1 ) ) ];
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Me_1	Me_2
A	XX	2000	3	1
A	XX	2001	4	2
A	XX	2002	7	4
A	XX	2003	6	3
A	YY	2000	9	3
A	YY	2001	5	1
A	YY	2002	10	4
A	YY	2003	5	1

Ratio to report: *ratio_to_report*

Syntax

in a Data Set expression	**ratio_to_report (** dataset **over (** partitionClause **))**
in a Component expression within a calc clause	**ratio_to_report (** component **over (** partitionClause **))**

Input parameters

dataset	the operand Data Set
component	the operand Component
partitionClause	see Analytic invocation

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

dataset

```
dataset { measure<number> _+ }
```

component

```
component<number>
```

Result type

result

```
dataset { measure<number> _+ }
| component<number>
```

Additional Constraints

The Aggregate invocation is not allowed.

The *orderClause* and *windowClause* of the Analytic invocation syntax are not allowed.

Behaviour

The operator returns the ratio between the value of the current Data Point and the sum of the values of the partition which the current Data Point belongs to.

For other details, see Analytic invocation.

Examples

Given the operand dataset DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Me_1	Me_2
A	XX	2000	3	1
A	XX	2001	4	3
A	XX	2002	7	5
A	XX	2003	6	1
A	YY	2000	12	0
A	YY	2001	8	8
A	YY	2002	6	5
A	YY	2003	14	-3

Example 1

```
DS_r := ratio_to_report ( DS_1 over ( partition by Id_1, Id_2 ) );
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Me_1	Me_2
A	XX	2000	0.15	0.1
A	XX	2001	0.2	0.3
A	XX	2002	0.35	0.5
A	XX	2003	0.3	0.1
A	YY	2000	0.3	0
A	YY	2001	0.2	0.8
A	YY	2002	0.15	0.5
A	YY	2003	0.35	-0.3

VTL-ML - Data Validation Operators

Check datapoint: *check_datapoint*

Syntax

check_datapoint (op , dpr { **components** listComp } { output } **)**

listComp ::= comp { , comp }*

output ::= **invalid | all | all_measures**

Input parameters

op	the Data Set to check
dpr	the Data Point Ruleset to be used
listComp	if *dpr* is defined on Value Domains then *listComp* is the list of Components of *op* to be associated (in positional order) to the conditioning Value Domains defined in *dpr*. If *dpr* is defined on Variables then *listComp* is the list of Components of *op* to be associated (in positional order) to the conditioning Variables defined in *dpr* (for documentation purposes).
comp	Component of *op*
output	specifies the Data Points and the Measures of the resulting Data Set: | · **invalid**: the resulting Data Set contains a Data Point for each Data Point of *op* and | each Rule in *dpr* that evaluates to FALSE on that Data Point. The resulting | Data Set has the Measures of *op*. | · **all**: the resulting Data Set contains a data point for each Data Point of *op* and | each Rule in *dpr*. The resulting Data Set has the *boolean* Measure bool_var. | · **all_measures**: the resulting Data Set contains a Data Point for each Data Point of *op* | and each Rule in *dpr*. The resulting dataset has the Measures of *op* and the | *boolean* Measure bool_var. | If not specified then *output* is assumed to be *invalid*. See the Behaviour for further details.

Examples of valid syntaxes

```
check_datapoint ( DS1, DPR invalid )
check_datapoint ( DS1, DPR all_measures )
```

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

op

```
dataset
```

dpr

```
name<datapoint>
```

comp

```
name<component>
```

Result type

result

```
dataset
```

Additional Constraints

If *dpr* is defined on Value Domains then it is mandatory to specify *listComp*. The Components specified in *listComp* must belong to the operand *op* and be defined on the Value Domains specified in the signature of *dpr*.

If *dpr* is defined on Variables then the Components specified in the signature of *dpr* must belong to the operand *op*.

If *dpr* is defined on Variables and *listComp* is specified then the Components specified in *listComp* are the same, in the same order, as those specified in *op* (they are provided for documentation purposes).

Behaviour

It returns a Data Set having the following Components:

- the Identifier Components of *op*
- the Identifier Component *ruleid* whose aim is to identify the Rule that has generated the actual Data Point (it contains at least the Rule name specified in *dpr* 8)
- if the *output* parameter is **invalid**: the original Measures of *op* (no *boolean* measure)
- if the *output* parameter is **all**: the *boolean* Measure bool_var whose value is the result of the evaluation of a rule on a Data Point (TRUE, FALSE or NULL).
- if the *output* parameter is **all_measures**: the original measures of *op* and the *boolean* Measure bool_var whose value is the result of the evaluation of a rule on a Data Point (TRUE, FALSE or NULL).
- the Measure errorcode that contains the errorcode specified in the rule
- the Measure errorlevel that contains the errorlevel specified in the rule

A Data Point of *op* can produce several Data Points in the resulting Data Set, each of them with a different value of ruleid. If *output* is **invalid** then the resulting Data Set contains a Data Point for each Data Point of *op* and each rule of *dpr* that evaluates to FALSE. If *output* is **all** or **all_measures** then the resulting Data Set contains a Data Point for each Data Point of *op* and each rule of *dpr*.

Examples

Given the operand dataset DS_1 and the datapoint ruleset dpr1:

```
define datapoint ruleset dpr1 ( variable Id_3, Me_1 ) is
    when Id_3 = “CREDIT” then Me_1 >= 0 errorcode “Bad credit”
    ; when Id_3 = “DEBIT” then Me_1 >= 0 errorcode “Bad debit”
end datapoint ruleset
```

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Me_1
2011	I	CREDIT	10
2011	I	DEBIT	-2
2012	I	CREDIT	10
2012	I	DEBIT	2

Example 1

```
DS_r := check_datapoint ( DS_1, dpr1 );
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Me_1	ruleid	errorcode	errorlevel
2011	I	DEBIT	-2	2	Bad debit	

Example 2

```
DS_r := check_datapoint ( DS_1, dpr1 all );
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	bool_var	ruleid	errorcode	errorlevel
2011	I	CREDIT	true	1		
2011	I	DEBIT	true	1		
2012	I	CREDIT	true	1		
2012	I	DEBIT	true	1		
2011	I	CREDIT	true	2		
2011	I	DEBIT	false	2	Bad debit	
2012	I	CREDIT	true	2		
2012	I	DEBIT	true	2		

Check hierarchy: *check_hierarchy*

Syntax

check_hierarchy (op , hr { **condition** condComp { , condComp }* } { **rule** ruleComp } { mode } { input } { output } **)**

mode ::= **non_null | non_zero | partial_null | partial_zero | always_null | always_zero**

input ::= **dataset | dataset_priority**

output ::= **invalid | all | all_measures**

Input parameters

op	the Data Set to be checked
hr	the hierarchical Ruleset to be used
condComp	*condComp* is a Component of *op* to be associated (in positional order) to the conditioning Value Domains or Variables defined in *hr* (if any).
ruleComp	*ruleComp* is the Identifier Component of *op* to be associated to the rule Value Domain or Variable defined in *hr*.

mode	this parameter specifies how to treat the possible missing Data Points corresponding to the Code Items in the left and right sides of the rules and which Data Points are produced in output. The meaning of the possible values of the parameter is explained below.
input	this parameter specifies the source of the values used as input of the comparisons. The meaning of the possible values of the parameter is explained below.
output	this parameter specifies the structure and the content of the resulting dataset. The meaning of the possible values of the parameter is explained below.

Examples of valid syntaxes

```
check_hierarchy ( DS1, HR_2  non_null  dataset  invalid )
check_hierarchy ( DS1, HR_3  non_zero dataset_priority  all )
```

Input parameters type

op

```
dataset { measure<number> _ }
```

hr

```
name<hierarchical>
```

condComp

```
name<component>
```

ruleComp

```
name<identifier>
```

Result type

result

```
dataset { measure<number> _ }
```

Additional Constraints

If *hr* is defined on Value Domains then it is mandatory to specify the *condition* (if any in the ruleset *hr*) and the *rule* parameters. Moreover, the Components specified as *condComp* and *ruleComp* must belong to the operand *op* and must take values on the Value Domains corresponding, in positional order, to the ones specified in the condition and rule parameter of *hr*.

If *hr* is defined on Variables, the specification of *condComp* and *ruleComp* is not needed, but they can be specified all the same if it is desired to show explicitly in the invocation which are the involved Components: in this case, the *condComp* and *ruleComp* must be the same and in the same order as the Variables specified in the *condition* and *rule* signatures of *hr*.

Behaviour

The **check_hierarchy** operator applies the Rules of the Ruleset *hr* to check the Code Items Relations between the Code Items present in *op* (as for the Code Items Relations, see the User Manual - section “Generic Model for Variables and Value Domains”). The operator checks if the relation between the left and the right member is fulfilled, giving TRUE in positive case and FALSE in negative case.

The Attribute propagation rule is applied on each group of Data Points which contributes to the same Data Point of the result.

The behaviours relevant to the different options of the input parameters are the following.

First, the parameter *input* is used to determine the source of the Data Points used as input of the *check_hierarchy*. The possible options of the parameter *input* and the corresponding behaviours are the following:

dataset	this option addresses the case where all the input Data Points of all the Rules of the Ruleset are expected to be taken from the input Data Set (the operand *op*). For each Rule of the Ruleset and for each item on the left and right sides of the Rule, the operator takes the input Data Points exclusively from the operand *op*.
dataset_priority	this option addresses the case where the input Data Points of all the Rules of the Ruleset are preferably taken from the input Data Set (the operand *op*). However, if a valid Measure value for an expected Data Point is not found in *op*, the attempt is made to take it from the computed output of a (possible) other Rule. For each Rule of the Ruleset and for each item on the left and right sides of the Rule: · if the item is not defined as the result (left side) of another Rule that applies the Code Item relation “is equal to” (=), the current Rule takes the input Data Points from the operand *op*. · if the item is defined as result of another Rule *R* that applies the Code Item relation “is equal to” (=), then: > if an expected input Data Point exists in *op* and its Measure is not NULL, then the current Rule takes such Data Point from *op*; > if an expected input Data Point does not exist in *op* or its measure is NULL, then the current Rule takes the Data Point (if any) that has the same Identifiers’ values from the computed output of the other Rule *R*;

If the parameter *input* is not specified then it is assumed to be *dataset*.

Then the parameter *mode* is considered, to determine the behaviour for missing Data Points and for the Data Points to be produced in the output. The possible options of the parameter *mode* and the corresponding behaviours are the following:

non_null	the result Data Point is produced when all the items involved in the comparison exist and have not NULL Measure value (i.e., when no Data Point corresponding to the Code Items of the left and right sides of the rule is missing or has NULL Measure value); under this option, in evaluating the comparison, the possible missing Data Points corresponding to the Code Items of the left and right sides of the rule are considered existing and having a NULL Measure value;
non_zero	the result Data Point is produced when at least one of the items involved in the comparison exist and have Measure not equal to 0 (zero); the possible missing Data Points corresponding to the Code Items of the left and right sides of the rule are considered existing and having a Measure value equal to 0;
partial_null	the result Data Point is produced if at least one Data Point corresponding to the Code Items of the left and right sides of the rule is found (whichever is its Measure value); the possible missing Data Points corresponding to the Code Items of the left and right sides of the rule are considered existing and having a NULL Measure value;
partial_zero	the result Data Point is produced if at least one Data Point corresponding to the Code Items of the left and right sides of the rule is found (whichever is its Measure value); the possible missing Data Points corresponding to the Code Items of the left and right sides of the rule are considered existing and having a Measure value equal to 0 (zero);
always_null	the result Data Point is produced in any case; the possible missing Data Points corresponding to the Code Items of the left and right sides of the rule are considered existing and having a Measure value equal to NULL;
always_zero	the result Data Point is produced in any case; the possible missing Data Points corresponding to the Code Items of the left and right sides of the rule are considered existing and having a Measure value equal to 0 (zero);

If the parameter *mode* is not specified, then it is assumed to be *non_null*.

The following table summarizes the behaviour of the options of the parameter “mode”:

OPTION of the MODE PARAMETER:	Missing Data Points are considered:	Null Data Points are considered:	Condition for evaluating the rule	Returned Data Points
Non_null	NULL	NULL	If all the involved Data Points are not NULL	Only not NULL Data Points (Zeros are returned too)

Non_zero	Zero	NULL	If at least one the involved Data Points is <> zero	Only not zero Data Points (NULL are returned too)
Partial_null	NULL	NULL	If at least the involved Data Points is not NULL	Data Points of any value (NULL or not NULL and zero too)
Partial_zero	Zero	NULL	If at least the involved Data Points is not NULL	Data Points of any value (NULL or not NULL and zero too)
Always_null	NULL	NULL	Always	Data Points of any value (NULL or not NULL and zero too)
Always_zero	Zero	NULL	Always	Data Points of any value (NULL or not NULL and zero too)

Finally the parameter *output* is considered, to determine the structure and content of the resulting Data Set. The possible options of the parameter *output* and the corresponding behaviours are the following:

all	all the Data Points produced by the comparison are returned, both the valid ones (TRUE) and the invalid ones (FALSE) besides the possible NULL ones. The result of the comparison is returned in the *boolean* Measure *bool_var*. The original Measure Component of the Data Set *op* is not returned.
invalid	only the invalid (FALSE) Data Points produced by the comparison are returned. The result of the comparison (*boolean* Measure *bool_var*) is not returned. The original Measure Component of the Data Set *op* is returned and contains the Measure values taken from the Data Points on the left side of the rule.
all_measures	all the Data Points produced by the comparison are returned, both the valid ones (TRUE) and the invalid ones (FALSE) besides the possible NULL ones. The result of the comparison is returned in the *boolean* Measure *bool_var*. The original Measure Component of the Data Set *op* is returned and contains the Measure values taken from the Data Points on the left side of the rule.

If the parameter *output* is not specified then it is assumed to be *invalid*.

In conclusion, the operator returns a Data Set having the following Components:

- all the Identifier Components of *op*
- the additional Identifier Component *ruleid*, whose aim is to identify the Rule that has generated the actual Data Point (it contains at least the Rule name specified in *hr* (The content of *ruleid* maybe personalised in the implementation))
- if the *output* parameter is *all*: the *boolean* Measure *bool_var* whose values are the result of the evaluation of the Rules (TRUE, FALSE or NULL).
- if the *output* parameter is *invalid*: the original Measure of *op*, whose values are taken from the Measure values of the Data Points of the left side of the Rule

- if the *output* parameter is *all_measures*: the *boolean* Measure *bool_var*, whose value is the result of the evaluation of a Rule on a Data Point (TRUE, FALSE or NULL), and the original Measure of *op*, whose values are taken from the Measure values of the Data Points of the left side of the Rule
- the Measure *imbalance*, which contains the difference between the Measure values of the Data Points on the left side of the Rule and the Measure values of the corresponding calculated Data Points on the right side of the Rule
- the Measure *errorcode*, which contains the *errorcode* value specified in the Rule
- the Measure *errorlevel*, which contains the *errorlevel* value specified in the Rule

Note that a generic Data Point of *op* can produce several Data Points in the resulting Data Set, one for each Rule in which the Data Point appears as the left member of the comparison.

Examples

See also the examples in define hierarchical ruleset.

Given the following hierarchical ruleset:

```
define hierarchical ruleset HR_1 ( valuedomain rule VD_1 ) is
    R010 : A = J + K + L errorlevel 5
    ; R020 : B = M + N + O errorlevel 5
    ; R030 : C = P + Q errorcode XX errorlevel 5
    ; R040 : D = R + S errorlevel 1
    ; R060 : F = Y + W + Z errorlevel 7
    ; R070 : G = B + C
    ; R080 : H = D + E errorlevel 0
    ; R090 : I = D + G errorcode YY errorlevel 0
    ; R100 : M >= N errorlevel 5
    ; R110 : M <= G errorlevel 5
end hierarchical ruleset
```

And given the operand Data Set DS_1 (where *At_1* is viral and the propagation rule says that the alphabetic order prevails the NULL prevails on the alphabetic characters and the Attribute value for missing Data Points is assumed as NULL):

Input **DS_1** (see structure)

Id_1	Id_2	Me_1
2010	A	5
2010	B	11
2010	C	0
2010	G	19
2010	H	
2010	I	14
2010	M	2
2010	N	5
2010	O	4
2010	P	7
2010	Q	-7
2010	S	3
2010	T	9
2010	U	
2010	V	6

Example 1

```
DS_r := check_hierarchy ( DS_1, HR_1 rule Id_2 partial_null all );
```

results in (see structure):

DS_r

Id_1	Id_2	ruleid	bool_var	imbalance	errorcode	errorlevel
2010	A	R010				5
2010	B	R020	true	0		5
2010	C	R030	true	0	XX	5
2010	D	R040				1
2010	E	R050				0
2010	F	R060				7
2010	G	R070	false	8		
2010	H	R080				0
2010	I	R090			YY	0
2010	M	R100	false	-3		5
2010	M	R110	true	-17		5

Check : *check*

Syntax

check (op { **errorcode** errorcode } { **errorlevel** errorlevel } { **imbalance** imbalance } { output })

output ::= **invalid | all**

Input parameters

op	a *boolean* Data Set (a *boolean* condition expressed on one or more Data Sets)
errorcode	the error code to be produced when the condition evaluates to FALSE. It must be a valid value of the *errorcode_vd* Value Domain (or *string* if the *errorcode_vd* Value Domain is not found). It can be a Data Set or a *scalar*. If not specified then *errorcode* is NULL.
errorlevel	the error level to be produced when the condition evaluates to FALSE. It must be a valid value of the *errorlevel_vd* Value Domain (or *integer* if the errorcode_vd Value Domain is not found). It can be a Data Set or a *scalar*. If not specified then *errorlevel* is NULL.
imbalance	the imbalance to be computed. *imbalance* is a *numeric* mono-measure Data Set with the same Identifiers of *op*. If not specified then *imbalance* is NULL.

output	specifies which Data Points are returned in the resulting Data Set: · **invalid** returns the Data Points of op for which the condition evaluates to FALSE · **all** returns all Data Points of *op* If not specified then *output* is **all**.

Examples of valid syntaxes

```
check ( DS1 > DS2 errorcode myerrorcode errorlevel myerrorlevel imbalance DS1 - DS2  invalid
```

Input parameters type

op

`dataset`

errorcode

`errorcode_vd`

errorlevel

`errorlevel_vd`

imbalance

`number`

Result type

result

`dataset`

Additional Constraints

op has exactly a *boolean* Measure Component.

Behaviour

It returns a Data Set having the following components:

- the Identifier Components of *op*
- a *boolean* Measure named *bool_var* that contains the result of the evaluation of the boolean dataset *op*
- the Measure *imbalance* that contains the specified imbalance
- the Measure *errorcode* that contains the specified *errorcode*
- the Measure *errorlevel* that contains the specified *errorlevel*

If *output* is **all** then all data points are returned. If *output* is **invalid** then only the Data Points where *bool_var* is FALSE are returned.

Examples

Given the Data Sets DS_1 and DS_2:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1
2010	I	1
2011	I	2

2012	I	10
2013	I	4
2014	I	5
2015	I	6
2010	D	25
2011	D	35
2012	D	45
2013	D	55
2014	D	50
2015	D	75

Input **DS_2** (see structure)

Id_1	Id_2	Me_1
2010	I	9
2011	I	2
2012	I	10
2013	I	7
2014	I	5
2015	I	6
2010	D	50
2011	D	35
2012	D	40
2013	D	55
2014	D	65
2015	D	75

Example 1

```
DS_r := check ( DS_1 >= DS_2 imbalance DS_1 - DS_2 );
```

results in (see structure):

DS_r

Id_1	Id_2	bool_var	imbalance	errorcode	errorlevel
2010	I	false	-8		
2011	I	true	0		
2012	I	true	0		
2013	I	false	-3		
2014	I	true	0		
2015	I	true	0		
2010	D	false	-25		
2011	D	true	0		
2012	D	true	5		

2013	D	true	0		
2014	D	false	-15		
2015	D	true	0		

VTL-ML - Conditional Operators

if-then-else: *if*

Syntax

if condition **then** thenOperand **else** elseOperand

Input parameters

condition	a Boolean condition (dataset, component or scalar)
thenOperand	the operand returned when *condition* evaluates to **true**
elseOperand	the operand returned when *condition* evaluates to **false**

Examples of valid syntaxes

```
if A > B then A else B
```

Semantics for scalar operations

The **if** operator returns *thenOperand* if *condition* evaluates to **true**, *elseOperand* otherwise. For example, considering the statement:

```
if x1 > x2 then 2 else 5,
    for x1 = 3, x2 =0 it returns 2
    for x1 = 0, x2 =3 it returns 5
```

Input parameters type

condition

```
dataset { measure <boolean> _ }
| component<Boolean>
| boolean
```

thenOperand

```
dataset
| component
| scalar
```

elseOperand

```
dataset
| component
| scalar
```

Result type

result

```
dataset
| component
| scalar
```

Additional Constraints

- The operands *thenOperand* and *elseOperand* must be of the same scalar type.
- If the operation is at scalar level, *thenOperand* and *elseOperand* are scalar then *condition* must be scalar too (a *boolean* scalar).
- If the operation is at Component level, at least one of *thenOperand* and *elseOperand* is a Component (the other one can be scalar) and *condition* must be a Component too (a *boolean* Component); *thenOperand*, *elseOperand* and the other Components referenced in *condition* must belong to the same Data Set.
- If the operation is at Data Set level, at least one of *thenOperand* and *elseOperand* is a Data Set (the other one can be scalar) and *condition* must be a Data Set too (having a unique *boolean* Measure) and must have the same Identifiers as *thenOperand* or/and *ElseOperand*
  - If *thenOperand* and *elseOperand* are both Data Sets then they must have the same Components in the same roles
  - If one of *thenOperand* and *elseOperand* is a Data Set and the other one is a scalar, the Measures of the operand Data Set must be all of the same scalar type as the scalar operand.

Behaviour

For operations at Component level, the operation is applied for each Data Point of the unique input Data Set, the **if-then-else** operator returns the value from the *thenOperand* Component when *condition* evaluates to **true**, otherwise it returns the value from the *elseOperand* Component. If one of the operands *thenOperand* or *elseOperand* is scalar, such a scalar value can be returned depending on the outcome of the *condition*.

For operations at Data Set level, the **if-then-else** operator returns the Data Point from *thenOperand* when the Data Point of *condition* having the same Identifiers' values evaluates to **true**, and returns the Data Point from *elseOperand* otherwise. If one of the operands *thenOperand* or *elseOperand* is scalar, such a scalar value can be returned (depending on the outcome of the *condition*) and in this case it feeds the values of all the Measures of the result Data Point.

The behaviour for two Data Sets can be procedurally explained as follows. First the *condition* Data Set is evaluated, then its true Data Points are inner joined with *thenOperand* and its false Data Points are inner joined with *elseOperand*, finally the union is made of these two partial results (the *condition* ensures that there cannot be conflicts in the union).

Examples

Given the operand Data Sets DS_cond, DS_1 and DS_2:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Id_4	Me_1
2012	S	Total	F	25.8
2012	F	Total	F	
2012	I	Total	F	20.9
2012	A	Total	M	6.3

Input **DS_2** (see structure)

Id_1	Id_2	Id_3	Id_4	Me_1
2012	B	Total	M	0.12
2012	G	Total	M	22.5
2012	S	Total	M	23.7
2012	A	Total	F	

Input **DS_COND** (see structure)

Id_1	Id_2	Id_3	Id_4	Me_1

2012	B	Total	M	5451780
2012	B	Total	F	5643070
2012	G	Total	M	5449803
2012	G	Total	F	5673231
2012	S	Total	M	23099012
2012	S	Total	F	23719207
2012	F	Total	M	31616281
2012	F	Total	F	33671580
2012	I	Total	M	28726599
2012	I	Total	F	30667608
2012	A	Total	M	
2012	A	Total	F	

Example 1

```
DS_r := if ( DS_cond#Id_4 = "F" ) then DS_1 else DS_2;
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Id_4	Me_1
2012	S	Total	F	25.8
2012	F	Total	F	
2012	I	Total	F	20.9
2012	B	Total	M	0.12
2012	G	Total	M	22.5
2012	S	Total	M	23.7

Case: *case*

Syntax

case when condition **then** thenOperand {**when** condition **then** thenOperand}*
else elseOperand

Input parameters

condition	a Boolean condition (dataset, component or scalar)
thenOperand	the operand returned when *condition* evaluates to **true**
elseOperand	the operand returned when *condition* evaluates to **false**

Examples of valid syntaxes

```
case when A > B then A when A = B then A else B
```

Semantics for scalar operations

The **case** operator returns the first *thenOperand* whose corresponding *condition* evaluates to **true**, *elseOperand* if none of the **when** conditions evaluates to **true**.

For example, considering the statement:

```
case when x1 > x2  then  2 when x1 = x2 then 0 else 5;

for  x1 = 3,  x2 = 0         it returns  2
for  x1 = x2 = 3         it returns  0
for  x1 = 0,  x2 = 3         it returns  5
```

Input parameters type

condition

```
dataset { measure <boolean> _ }
| component<Boolean>
| boolean
```

thenOperand

```
dataset
| component
| scalar
```

elseOperand

```
dataset
| component
| scalar
```

Result type

result

```
dataset
| component
| scalar
```

Additional Constraints

The same rules apply as for the if-then-else operator.

Behaviour

For operations at Component level, the operation is applied for each Data Point of the unique input Data Set, the **case** operator returns the value from the *thenOperand* Component whose corresponding *condition* evaluates to *true*; if none of the **when** conditions evaluates to **true**, it returns the value from the *elseOperand* Component. If one of the operands *thenOperand* or *elseOperand* is scalar, such a scalar value can be returned depending on the outcome of the condition.

For operations at Data Set level, the **case** operator returns the Data Point from the *thenOperand* when the first Data Point of *condition* having the same Identifiers' values evaluates to **true**; returns the Data Point from *elseOperand* if none of the **when** conditions evaluates to **true**. If one of the operands *thenOperand* or *elseOperand* is scalar, such a scalar value can be returned (depending on the outcome of the *condition*) and in this case it feeds the values of all the Measures of the result Data Point.

The behaviour for two Data Sets can be procedurally explained as follows. First the *condition* Data Set is evaluated, then its true Data Points are inner joined with *thenOperand* and its false Data Points are inner joined with *elseOperand*, finally the union is made of these two partial results (the *condition* ensures that there cannot be conflicts in the union).

Examples

Given the operand Data Set DS_1:

Input **DS_1** (see structure)

Id_1	Me_1
1	0.12
2	3.5
3	10.7
4	

Example 1

```
DS_r := DS_1
   [calc Me_2 :=
         case when Me_1 <= 1 then 0
                                        when Me_1 > 1 and Me_1 <= 10 then 1
              when Me_1 > 10  then 10
              else 100]
```

results in (see structure):

DS_r

Id_1	Me_1	Me_2
1	0.12	0
2	3.5	1
3	10.7	10
4		100

Nvl: *nvl*

Syntax

nvl (op1 , op2 **)**

Input parameters

op1	the first operand
op2	the second operand

Examples of valid syntaxes

```
nvl ( ds1#m1, 0 )
```

Semantics for scalar operations

The operator *nvl* returns *op2* when *op1* is **null**, otherwise *op1*.

For example:

```
nvl ( 5, 0 ) returns 5
nvl ( null, 0 ) returns 0
```

Input parameters type

op1

```
dataset
| component
| scalar
```

op2

```
dataset
| component
| scalar
```

Result type

result

```
dataset
| component
| scalar
```

Additional Constraints

If *op1* and *op2* are scalar values then they must be of the same type.

If *op1* and *op2* are Components then they must be of the same type.

If *op1* and *op2* are Data Sets then they must have the same Components.

Behaviour

The operator **nvl** returns the value from *op2* when the value from *op1* is null, otherwise it returns the value from *op1*.

The operator has the typical behaviour of the operators applicable on two scalar values or Data Sets or Data Set Components.

Also the following statement gives the same result: *if isnull (op1) then op2 else op1*.

Examples

Given the input Data Set DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Id_4	Me_1
2012	B	Total	Total	11094850
2012	G	Total	Total	11123034
2012	S	Total	Total	
2012	M	Total	Total	417546
2012	F	Total	Total	5401267
2012	N	Total	Total	

Example 1

```
DS_r := nvl ( DS_1, 0 );
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Id_4	Me_1

2012	B	Total	Total	11094850.0
2012	G	Total	Total	11123034.0
2012	S	Total	Total	0.0
2012	M	Total	Total	417546.0
2012	F	Total	Total	5401267.0
2012	N	Total	Total	0.0

VTL-ML - Clause Operators

Filtering Data Points: *filter*

Syntax

op **[filter** filterCondition **]**

Input parameters

op	the operand
filterCondition	the filter condition

Examples of valid syntaxes

```
DS_1 [ filter Me_3 > 0 ]
DS_1 [ filter  Me_3 + Me_2 <= 0 ]
```

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

op

`dataset`

filterCondition

`component<boolean>`

Result type

result

`dataset`

Additional Constraints

None.

Behavior

The operator takes as input a Data Set (*op*) and a *boolean* Component expression (*filterCondition*) and filters the input Data Points according to the evaluation of the *condition*. When the expression is TRUE the Data Point is kept in the result, otherwise it is not kept (in other words, it filters out the Data Points of the operand Data Set for which *filterCondition* condition evaluates to FALSE or NULL).

Examples

Given the Data Set DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Me_1	At_1
1	A	XX	2	E
1	A	YY	2	F
1	B	XX	20	F
1	B	YY	1	F
2	A	XX	4	E
2	A	YY	9	F

Example 1

```
DS_r := DS_1 [ filter Id_1 = 1 and Me_1 < 10 ];
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Me_1	At_1
1	A	XX	2	E
1	A	YY	2	F
1	B	YY	1	F

Calculation of a Component: *calc*

Syntax

op **[calc** { calcRole } calcComp := calcExpr { , { calcRole } calcComp := calcExpr }* **]**

calcRole ::= **identifier | measure | attribute | viral attribute**

Input parameters

op	the operand
calcRole	the role to ba assigned to a Component to be calculated
calcComp	the name of a Component to be calculated
calcExpr	expression at component level, having only Components of the input Data Sets as operands, used to calculate a Component

Examples of valid syntaxes

```
DS_1 [ calc Me_3 := Me_1 + Me_2 ]
```

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

op

`dataset`

calcComp

`name<component>`

calcExpr

`component<scalar>`

Result type

result

`dataset`

Additional Constraints

The *calcComp* parameter cannot be the name of an Identifier component.

All the components used in *calcComp* must belong to the operand Data Set *op*.

Behavior

The operator calculates new Identifier, Measure or Attribute Components on the basis of sub-expressions at Component level. Each Component is calculated through an independent sub-expression. It is possible to specify the role of the calculated Component among **measure**, **identifier**, **attribute**, or **viral attribute**, therefore the calc clause can be used also to change the role of a Component when possible. The keyword **viral** allows controlling the virality of the calculated Attributes (for the attribute propagation rule see the User Manual). When the role is omitted, the following rule is applied: if the component exists in the operand Data Set then it maintains its role; if the component does not exist in the operand Data Set then its role is Measure.

The *calcExpr* sub-expressions are independent one another, they can only reference Components of the input Data Set and cannot use Components generated, for example, by other *calcExpr*. If the calculated Component is a new Component, it is added to the output Data Set. If the Calculated component is a Measure or an Attribute that already exists in the input Data Set, the calculated values overwrite the original values. If the calculated Component is an Identifier that already exists in the input Data Set, an exception is raised because overwriting an Identifier Component is forbidden for preserving the functional behaviour. Analytic invocations can be used in the **calc** clause.

Examples

Given the Data Set DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Me_1
1	A	CA	20
1	B	CA	2
2	A	CA	2

Example 1

```
DS_r := DS_1 [ calc Me_1:= Me_1 * 2 ];
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Me_1

1	A	CA	40
1	B	CA	4
2	A	CA	4

Example 2

```
DS_r := DS_1 [ calc attribute At_1:= "EP" ];
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Me_1	At_1
1	A	CA	20	EP
1	B	CA	2	EP
2	A	CA	2	EP

Aggregation: *aggr*

Syntax

op **[aggr** aggrClause { groupingClause } **]**

aggrClause ::= { aggrRole } aggrComp := aggrExpr { , { aggrRole } aggrComp:= aggrExpr }*

groupingClause ::= { group by groupingId {, groupingId }*

| **group except** groupingId {, groupingId }*
| **group all** conversionExpr }[1]
{ **having** havingCondition }

aggrRole ::= **measure | attribute | viral attribute**

Input parameters

op	the operand
aggrClause	clause that specifies the required aggregations, i.e., the aggregated Components to be calculated, their roles and their calculation algorithm, to be applied on the joined and filtered Data Points
aggrRole	the role of the aggregated Component to be calculated
aggrComp	the name of the aggregated Component to be calculated; this is a dependent Component of the result (Measure or Attribute, not Identifier)

aggrExpr	expression at component level, having only Components of the input Data Sets as operands, which invokes an aggregate operator (e.g. **avg, count, max**…, see also the corresponding sections) to perform the desired aggregation. Note that the **count** operator is used in an *aggrClause* without parameters, e.g.: *DS_1 [aggr Me_1 := count ( ) group by Id_1)]*
groupingClause	the following alternative grouping options: · **group by**: the Data Points are grouped by the values of the specified Identifiers (*groupingId*). The Identifiers not specified are dropped in the result. · **group except**: the Data Points are grouped by the values of the Identifiers not specified as *groupingId*. The Identifiers specified as *groupingId* are dropped in the result. · **group all**: converts the values of an Identifier Component using *conversionExpr* and keeps all the resulting Identifiers.
groupingId	Identifier Component to be kept (in the **group by** clause) or dropped (in the **group except** clause).
conversionExpr	specifies a conversion operator (e.g., **time_agg**) to convert an Identifier from finer to coarser granularity. The conversion operator is applied on an Identifier of the operand Data Set *op*.
havingCondition	a condition (*boolean* expression) at component level, having only Components of the input Data Sets as operands (and possibly constants), to be fulfilled by the groups of Data Points: only groups for which *havingCondition* evaluates to TRUE appear in the result. The *havingCondition* refers to the groups specified through the *groupingClause*, therefore it must invoke aggregate operators (e.g. **avg, count, max**…, see also the section Aggregate invocation). A correct example of *havingCondition* is: *max(obs_value) < 1000* instead the condition *obs_value < 1000* is not a right *havingCondition*, because it refers to the values of the single Data Points and not to the groups. The **count** operator is used in a *havingCondition* without parameters, e.g.: *sum (DS_1 group by id1 having count ( ) >= 10)*

Examples of valid syntaxes

```
DS_1 [ aggr M1 := min ( Me_1 ) group by  Id_1, Id_2  ]
DS_1 [ aggr M1 := min ( Me_1 ) group except  Id_1, Id_2 ]
```

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

op

`dataset`

aggrComp

`name<component>`

aggrExpr

`component<scalar>`

groupingId

`name<identifier>`

conversionExpr

`identifier<scalar>`

havingCondition

`component<boolean>`

Result type

result

`dataset`

Additional Constraints

The *aggrComp* parameter cannot be the name of an Identifier component.

All the components used in *aggrExpr* must belong to the operand Data Set *op*.

The *conversionExpr* parameter applies just one conversion operator to just one Identifier belonging to the input Data Set. The basic scalar type of the Identifier must be compatible with the basic scalar type of the conversion operator.

Behaviour

The operator **aggr** calculates aggregations of dependent Components (Measures or Attributes) on the basis of sub-expressions at Component level. Each Component is calculated through an independent sub-expression. It is possible to specify the role of the calculated Component among **measure**, **attribute**, or **viral attribute**. The substring **viral** allows to control the virality of Attributes, if the Attribute propagation rule is adopted (see the User Manual). When the role is omitted, the following rule is applied: if the component exists in the operand Data Set then it maintains its role; if the component does not exist in the operand Data Set then its role is Measure.

The *aggrExpr* sub-expressions are independent of one another, they can only reference Components of the input Data Set and cannot use Components generated, for example, by other *aggrExpr* sub-expressions. The **aggr** computed Measures and Attributes are the only Measures and Attributes returned in the output Data Set (plus the possible viral Attributes). The sub-expressions must contain only Aggregate operators, which are able to compute an aggregated Value relevant to a group of Data Points. The groups of Data Points to be aggregated are specified through the *groupingClause*, which allows the following alternative options.

group by	by the Data Points are grouped by the values of the specified Identifiers. The Identifiers not specified are dropped in the result.
group except	the Data Points are grouped by the values of the Identifiers not specified in the clause. The specified Identifiers are dropped in the result.
group all	converts an Identifier Component using *conversionExpr* and keeps all the other Identifiers.

The **having** clause is used to filter groups in the result by means of an aggregate condition evaluated on the single groups (for example the minimum number of Data Points in the group).

If no grouping clause is specified, then all the input Data Points are aggregated in a single group and the clause returns a Data Set that contains a single Data Point and has no Identifiers.

The Attributes calculated through the **aggr** clauses are maintained in the result. For all the other Attributes that are defined as **viral**, the Attribute propagation rule is applied (for the semantics, see the Attribute Propagation Rule section in the User Manual).

Examples

Given the Data Set DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Me_1
1	A	XX	0
1	A	YY	2
1	B	XX	3
1	B	YY	5
2	A	XX	7
2	A	YY	2

Example 1

```
DS_r := DS_1 [ aggr Me_1:= sum( Me_1 ) group by Id_1 , Id_2 ];
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1
1	A	2
1	B	8
2	A	9

Example 2

```
DS_r := DS_1 [ aggr Me_3:= min( Me_1 ) group except Id_3 ];
```

results in (see structure):

DS_r

Id_1	Id_2	Me_3
1	A	0
1	B	3
2	A	2

Example 3

```
DS_r := DS_1 [ aggr Me_1:= sum( Me_1 ), Me_2 := max( Me_1) group by Id_1 , Id_2 having avg (
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1	Me_2
1	B	8	5
2	A	9	7

Maintaining Components: *keep*

Syntax

op **[keep** comp {, comp }* **]**

Input parameters

op	the operand
comp	a Component to keep

Examples of valid syntaxes

::

DS_1 [keep Me_2, Me_3]

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

op

`dataset`

comp

`name<component>`

Result type

result

`dataset`

Additional Constraints

All the Components *comp* must belong to the input Data Set *op*.

The Components *comp* cannot be Identifiers in *op*.

Behaviour

The operator takes as input a Data Set (*op*) and some Component names of such a Data Set (*comp*). These Components can be Measures or Attributes of *op* but not Identifiers. The operator maintains the specified Components, drops all the other dependent Components of the Data Set (Measures and Attributes) and maintains the independent Components (Identifiers) unchanged. This operation corresponds to a projection in the usual relational join semantics (specifying which columns will be projected in among Measures and Attributes).

Examples

Given the Data Set DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Me_1	Me_2	At_1
2010	A	XX	20	36	E
2010	A	YY	4	9	F
2010	B	XX	9	10	F

Example 1

```
DS_r := DS_1 [ keep Me_1 ];
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Me_1
2010	A	XX	20
2010	A	YY	4
2010	B	XX	9

Removal of Components: *drop*

Syntax

op **[drop** comp {, comp }* **]**

Input parameters

op	the operand
comp	a Component to drop

Examples of valid syntaxes

```
DS_1 [ drop Me_2, Me_3 ]
```

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

op

`dataset`

comp

`name<component>`

Result type

result

`dataset`

Additional Constraints

All the Components *comp* must belong to the input Data Set *op*.

The Components *comp* cannot be Identifiers in *op*.

Behaviour

The operator takes as input a Data Set (*op*) and some Component names of such a Data Set (*comp*). These Components can be Measures or Attributes of *op* but not Identifiers. The operator drops the specified Components and maintains all the other Components of the Data Set. This operation corresponds to a projection in the usual relational join semantics (specifying which columns will be projected out).

Examples

Given the Data Set DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Me_1	At_1
2010	A	XX	20	E
2010	A	YY	4	F
2010	B	XX	9	F

Example 1

```
DS_r := DS_1 [ drop At_1 ];
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Me_1
2010	A	XX	20
2010	A	YY	4
2010	B	XX	9

Change of Component name: *rename*

Syntax

op **[rename** comp_from **to** comp_to { , comp_from **to** comp_to}* **]**

Input parameters

op	the operand
comp_from	the original name of the Component to rename
comp_to	the new name of the Component after the renaming

Examples of valid syntaxes

```
DS_1 [ rename Me_2 to Me_3 ]
```

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

op

`dataset`

comp_from

`name<component>`

comp_to

`name<component>`

Result type

result

`dataset`

Additional Constraints

The corresponding pairs of Components before and after the renaming (*dsc_from* and *dsc_to*) must be defined on the same Value Domain and the same Value Domain Subset.

The components used in *dsc_from* must belong to the input Data Set and the component used in the *dsc_to* cannot have the same names as other Components of the result Data Set.

Behaviour

The operator assigns new names to one or more Components (Identifier, Measure or Attribute Components). The resulting Data Set, after renaming the specified Components, must have unique names of all its Components (otherwise a runtime error is raised). Only the Component name is changed and not the Component Values, therefore the new Component must be defined on the same Value Domain and Value Domain Subset as the original Component (see also the IM in the User Manual). If the name of a Component defined on a different Value Domain or Set is assigned, an error is raised. In other words, **rename** is a transformation of the variable without any change in its values.

Examples

Given the Data Set DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Me_1	At_1
1	B	XX	20	F
1	B	YY	1	F
2	A	XX	4	E

2	A	YY	9	F

Example 1

```
DS_r := DS_1 [ rename Me_1 to Me_2, At_1 to At_2];
```

results in (see structure):

DS_r

Id_1	Id_2	Id_3	Me_2	At_2
1	B	XX	20	F
1	B	YY	1	F
2	A	XX	4	E
2	A	YY	9	F

Pivoting: *pivot*

Syntax

op **[pivot** identifier , measure **]**

Input parameters

op	the operand
identifier	the Identifier Component of *op* to pivot
measure	the Measure Component of *op* to pivot

Examples of valid syntaxes

```
DS_1 [ pivot  Id_2,  Me_1 ]
```

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

op

```
dataset
```

identifier

```
name<identifier>
```

measure

```
name<measure>
```

Result type

result

```
dataset
```

Additional Constraints

The Measures created by the operator according to the behaviour described below must be defined on the same Value Domain as the input Measure.

Behaviour

The operator transposes several Data Points of the operand Data Set into a single Data Point of the resulting Data Set. The semantics of **pivot** can be procedurally described as follows.

1. It creates a virtual Data Set VDS as a copy of *op*
2. It drops the Identifier Component identifier and all the Measure Components from VDS.
3. It groups VDS by the values of the remaining Identifiers.
4. For each distinct value of *identifier* in *op*, it adds a corresponding measure to VDS, named as the value of *identifier*. These Measures are initialized with the NULL value.
5. For each Data Point of *op*, it finds the Data Point of VDS having the same values as for the common Identifiers and assigns the value of *measure* (taken from the current Data Point of *op*) to the Measure of VDS having the same name as the value of *identifier* (taken from the Data Point of *op*).

The result of the last step is the output of the operation.

Note that **pivot** may create Measures whose names are non-regular (i.e. they may contain special characters, reserved keywords, etc.) according to the rules about the artefact names described in the User Manual (see the section “The artefact names” in the chapter “VTL Transformations”). As said in the User Manual, those names must be quoted to be referenced within an expression.

Examples

Given the Data Set DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Me_1	At_1
1	A	5	E
1	B	2	F
1	C	7	F
2	A	3	E
2	B	4	E
2	C	9	F

Example 1

```
DS_r := DS_1 [ pivot Id_2, Me_1 ];
```

results in (see structure):

DS_r

Id_1	A	B	C
1	5	2	7
2	3	4	9

Unpivoting: *unpivot*

Syntax

op **[unpivot** identifier , measure **]**

Input parameters

op	the operand
identifier	the Identifier Component to be created
measure	the Measure Component to be created

Examples of valid syntaxes

```
DS_1 [ unpivot   Id_5,  Me_3 ]
```

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

op

```
dataset
```

identifier

```
name<identifier>
```

measure

```
name<measure>
```

Result type

result

```
dataset
```

Additional Constraints

All the measures of *op* must be defined on the same Value Domain.

Behaviour

The **unpivot** operator transposes a single Data Point of the operand Data Set into several Data Points of the result Data set. Its semantics can be procedurally described as follows.

1. It creates a virtual Data Set VDS as a copy of *op*
2. It adds the Identifier Component *identifier* and the Measure Component *measure* to VDS
3. For each Data Point DP and for each Measure M of *op* whose value is not NULL, the operator inserts a Data Point into VDS whose values are assigned as specified in the following points
4. The VDS Identifiers other than *identifier* are assigned the same values as the corresponding Identifiers of the *op* Data Point
5. The VDS *identifier* is assigned a value equal to the **name** of the Measure M of *op*
6. The VDS *measure* is assigned a value equal to the **value** of the Measure M of *op*

The result of the last step is the output of the operation.

When a Measure is NULL then **unpivot** does not create a Data Point for that Measure. Note that in general pivoting and unpivoting are not exactly symmetric operations, i.e., in some cases the unpivot operation applied to the pivoted Data Set does not recreate exactly the original Data Set (before pivoting).

Examples

Given the Data Set DS_1:

Input **DS_1** (see structure)

Id_1	A	B	C
1	5	2	7
2	3	4	9

Example 1

```
DS_r := DS_1 [ unpivot Id_2, Me_1 ];
```

results in (see structure):

DS_r

Id_1	Id_2	Me_1
1	A	5
2	A	3
1	B	2
2	B	4
1	C	7
2	C	9

Subspace: *sub*

Syntax

op **[sub** identifier **=** value { , identifier = value }* **]**

Input parameters

op	dataset
identifier	Identifier Component of the input Data Set *op*
value	valid value for *identifier*

Examples of valid syntaxes

```
DS_r := DS_1 [sub Id_2 = "A", Id_5 = 1 ]
```

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

op

```
dataset
```

identifier

```
name<identifier>
```

value

`scalar`

Result type

result

`dataset`

Additional Constraints

The specified Identifier Components *identifier* (s) must belong to the input Data Set *op*.

Each Identifier Component can be specified only once.

The specified *value* must be an allowed value for *identifier*.

Behaviour

The operator returns a Data Set in a subspace of the one of the input Dataset. Its behaviour can be procedurally described as follows:

1. It creates a virtual Data Set VDS as a copy of *op*
2. It maintains the Data Points of VDS for which *identifier* = *value* (for all the specified *identifier*) and eliminates all the Data Points for which *identifier* <> *value* (even for only one specified *identifier*)
3. It projects out (“drops”, in VTL terms) all the *identifier* (s)

The result of the last step is the output of the operation.

The resulting Data Set has the Identifier Components that are not specified as *identifier* (s) and has the same Measure and Attribute Components of the input Data Set.

The result Data Set does not violate the functional constraint because after the filter of the step 2, all the remaining *identifier* (s) do not contain the same Values for all the Data Points. In other words, given that the input Data Set is a 1st order function and therefore does not contain duplicates, the result Data Set is a 1st order function as well. To show this, let K■,…,K■,…,K■ be the Identifier components for the generic input Data Set *DS*. Let us suppose that K■,…,K■ are assigned to fixed values by using the subspace operator. A duplicate could arise only if in the result there are two Data Points DP■■ and DP■■ having the same value for K■■■,…,K■ , but this is impossible since such Data Points had same K■,…,K■ in the original Data Set *DS*, which did not contain duplicates.

If we consider the vector space of Data Points individuated by the n-uples of Identifier components of a Data Set DS(K■,…,K■,…) (along, e.g., with the operators of sum and multiplication), we have that the subspace operator actually performs a subsetting of such space into another space with fewer Identifiers. This can be also seen as the equivalent of a *dice* operation performed on hyper-cubes in multi-dimensional data warehousing.

Examples

Given the Data Set DS_1:

Input **DS_1** (see structure)

Id_1	Id_2	Id_3	Me_1	At_1
1	A	XX	20	F
1	A	YY	1	F
1	B	XX	4	E
1	B	YY	9	F
2	A	XX	7	F
2	A	YY	5	E
2	B	XX	12	F
2	B	YY	15	F

Example 1

```
DS_r := DS_1 [ sub Id_1 = 1, Id_2 = "A" ];
```

results in (see structure):

DS_r

Id_3	Me_1	At_1
XX	20	F
YY	1	F

Example 2

```
DS_r := DS_1 [ sub Id_1 = 1, Id_2 = "B", Id_3 = "YY" ];
```

results in (see structure):

DS_r

Me_1	At_1
9	F

Example 3

```
DS_r := DS_1 [ sub Id_2 = "A" ] + DS_1 [ sub Id_2 = "B" ];
```

results in (see structure):

DS_r

Id_1	Id_3	Me_1
1	XX	24
1	YY	10
2	XX	19
2	YY	20

PDF Version

PDF Version