EUROPEAN CENTRAL BANK

EUROSYSTEM

TRANSFORMATIONS

The representation of transformations in the BIRD database is based on the SDMX information model (see section II, 13.2 Model – Inheritance View, 13.2.1 Class Diagram).

According to the VTL model, a transformation scheme is an ordered list of transformations. Such a scheme therefore contains one or many transformations (i.e. one line of valid VTL code). The SDMX model specifies that transformations can contain one or many transformation nodes (i.e. the components of this line of valid VTL code). A transformation element is therefore a constant, an operation, or a BIRD model object (i.e. a variable or a cube, etc.). If the transformation element represents an operation, it can have a relationship with one or many transformation elements.

The BIRD database, up to version 5.0.2, contains information on the decomposition of each transformation scheme into its transformations and the decomposition of transformations into their transformation nodes according to the SDMX information model.

The sections “Example” and “Representation in the database” below explain the relationship between transformation schemes, transformations, and transformation nodes and how they are represented in the database. The section on “Description of VTL artefacts” also describes how the VTL that is regularly used in various transformation schemes works.

Conventions used in BIRD transformation rules/deviations from the VTL specification

1. Special Left Joins

The reason for not using the left join operator as defined in the VTL specification is that if we were to do so, we would have to handle instruments in different ways, depending on their relationship to other objects, such as protections. Take, for example, a loan associated with a protection and another loan without a protection. If we were to use the left join operator, we would not be able to (starting from the loan) generate a dataset that comprises both loans and includes protection information (because the second loan has no protection). From the perspective of the aggregated output layer (e.g. FINREP), there is no distinction between a non-existing protection and a protection with a value of 0.

For example, we consider three loans:

Loan		
[D] loanId	***Currency***	***value***
A	EUR	13
B	USD	17
C	EUR	19

And some protections:

Protection	
[D] protectionId	***value***
1	3
2	5
3	7

Two of these loans (A and C) are linked to protections while B is not:

instrumentProtection	
[D] loanId	***[D] protectionId***
a	1
a	2
c	1

A normal left join operator would omit the loan that is not linked to a protection, however, we specify our own special left join operator (for further details about the operator, see its specification in the transformation package) because we do not wish to treat the loans differently (at least from a business perspective):

specialLeftJoin(instrumentsAndTheirProtections, protection);				
[D] loanId	***[D] protectionId***	***currency***	***value***	***protection_value***
A	1	EUR	6.50	3
A	2	EUR	6.50	5
B	NULL	USD	17.00	NULL
C	1	EUR	19.00	3

This way, we can describe the data production process without having need for a technical distinction between loans with protections and loans without protections.

specialLeftJoin(instrumentsAndTheirProtections, protection);				
[D] loanId	***[D) protectionId***	***currency***	***value***	***protection_value***
A	1	EUR	6.50	3
A	2	EUR	6.50	5
B	NULL	USD	17.00	NULL
C	1	EUR	19.00	3

2. Keep with identifiers

Contrary to the VTL specification, we allow identifiers to be listed as arguments of a keep operation. We also use the keep operator to extract a list with unique vales, for example by using the expression result := someData[keep(identifierOfTheDataSet)], we indicate that the result should be a dataset with only one column with unique values similar to a SELECT DISTINCT identifierOfTheDataSet FROM someData.

3. Calc used for casting to identifier

We use the calc operator to describe such an expression because we did not find a cast operator in the VTL specification.

4. User defined operators with multiple expressions (not comprised in one expression)

According to the VTL specification, only one output parameter per user-defined operator is allowed, and theoretically it should be possible to merge multiple expressions into one expression (e.g. s := k * x and y := s + d -> y := (k * x) + d), however we do not comply with this constraint for two reasons:

(a) The official grammar is not capable of merging multiple expressions into one expression (at least not for all expressions), for example, the expression max(someData[keep itsObservation] group by itsIdentifier)[rename itsObservation to maximumOfTheObservation]; is not covered by the grammar, although the individual expressions are covered (i.e. firstStep := someData[keep itsObservation];, secondStep := max(firstStep group by itsIdentifier);) and result := secondStep[rename itsObservation to maximumOfTheObservation]; is valid according to the grammar (and the Reference Manual).

(b) From a readability perspective, it may prove disadvantageous to represent multiple expressions as only one statement or line.

5. Reusing names in a module / in the transformation schemes

According to the VTL specification, names in a transformation scheme need to be unique in order to build an oriented graph. For documentation purposes, this constraint seems rather burdensome because it would force us to apply certain naming conventions to the transformations used in a module (e.g. by adding an integer to every dataset used in a module). We believe that reusing names is a more flexible approach (consider, for example, an amendment to the first transformation of a module comprising multiple transformations) and the fact that a module comprises an order list of transformations also allows us to generate oriented graph structures.

6. Reusing names for variables (in an expression)

The argumentation for this inconsistency with the VTL specification is similar to the one described above.

7. Omitting the perspective identifier and the reference date in join operations (except in cases where they are relevant for business purposes)

The perspective identifier and the reference date act as dimensions (i.e. as part of the primary key) for most of the cubes in the BIRD Input Layer (IL). We therefore do not use them in the transformations, except in cases where these variables are relevant for business purposes, for example, when selecting a specific perspective or a particular reference date. The rationale behind this convention is that there is no additional value added by listing them in the transformations explicitly.

8. Amendments to the grammar

See the grammar on our GitHub page for an overview of all amendments made to the official grammar.

(8.1) Comments allowed for individual objects, such as datasets, if operators and else operators

We added our own token for comments (so as not to interfere with the official comment tokens) and allowed them in front of if operators, else operators and variable identifiers (varID).

(8.2) Labels added for parser rules

In order to process expressions more robustly, we added labels for some parser rules (see, for example, Vtl.g4, line 24 expr parser rule).

(8.3) A parser rule added for calling user-defined operatorsTransformation package structure

The transformation package in the dictionary is comprised by the following objects:

i. Modules
ii. Transformations
iii. Functions
iv. Transformation nodes
v. Transformation node relationships
vi. Transformation schemes

A Module is an ordered set of Transformations specifying manipulation of data based on (a) business requirements and/or (b) technical necessities.

A business requirement may be represented by stating that the type of amortisation for reverse repurchase loans is bullet (4) by default (see P_IMPLCT_RVRS_RPRCHS_LNS). A technical necessity may be expressed by the fact that in order to enrich a loan with its associated protection, we first need to enrich it with the linking table that establishes the many-to-many relationship between a loan and a

protection (“one loan might be linked to many protections, while one protection may be used for multiple loans”).

Each module is assigned to a phase in the BIRD process that allows us to identify its location between the IL and the Reference/Non-reference Output Layer (ROL/NROL). Additionally, each module is assigned to a type specifying its purpose.

A transformation is a statement expressed in VTL. See the conventions used in the BIRD transformation rules/deviations from the VTL specification to find out more about how VTL is used in the BIRD. Each transformation has one output and one or many inputs which may be identified by exploring the transformation nodes associated with a particular transformation.

Each transformation is decomposed into its transformation nodes, i.e. the elementary objects used to describe the expression. For example, the expression y := k * x + d consists of seven transformation nodes where y, k, x and d represent datasets and :=, * and + represent operators. Each operator has a predefined structure, for example, in the case of the assignment operator (i.e. :=) there is a left-hand side and a right-hand side; the same is the case for the multiplication operator (*) and the addition (+) operator. Note that most operators have a more complex structure than binary operators do. Note also that the defined structure for each operator was chosen arbitrarily by the person providing the parser for VTL syntax and is not part of the official VTL specification. Because of the structure assigned to each operator node, the transformation nodes associated with a transformation may be represented in a tree structure (i.e. a parent-child relationship between transformation nodes where each node has at most one parent). This structure (which is also called “abstract syntax tree structure”) may prove useful for technology independent implementation.

The transformation node relationships represent relationships between nodes that are not covered by the abstract syntax tree. These relationships show how the transformation nodes of different transformations are linked, i.e. the input and output parameters of each transformation. For example, the transformation node relationship between the transformations (T1) s := k * x and (T2) y := s + d specifies that the datasets in the first transformation (“T1.s”) would be used in the second transformation (“T2.s”), i.e. T1.s -> T2.s. In this way, the transformation package comprises the data lineage of a dataset in the BIRD process.

A transformation scheme can be considered a VTL program, i.e. a set of transformations that are run together. The transformation schemes listed in the transformation package are the result of the oriented graph structure for each output cube. Each transformation scheme comprises the transformations of the modules involved.

Example

The following example aims to clarify the content of the transformation package in the BIRD database. To cover different aspects, we will organise our transformations into two modules.

In the first module we assume we have a (database-)table named “coordinates” containing the columns (i.e. variables) x and y, which (clearly) relate to some coordinate system. Our transformation scheme’s goal is to derive a new variable distance for all records where x and y are greater than, or equal to, 0 defined in the following way:

$$distance = sqrt(x * x + y * y)$$

The second module simply describes that we filter the resulting dataset only for those records where the distance is greater than 11.

Using VTL syntax we would write the following lines:

```
// first Module

/*filter all records of the data set coordinates where x and y are greater or equal to zero*/

firstResult := coordinates [filter x>=0 and y>=0];

/*keep only x and y*/

secondResult := firstResult [keep x, y];

/*create a new variable (=column) named distance which will be populated with the square root of the sum of x squared and y squared*/

result := secondResult [calc distance := sqrt(x*x+y*y)];


// second Module

/*finally we create a data set where that only contains records where the distance is greater than 11*/

finalResult := result [filter distance > 11];
```

The tree structure with respect to the second line can be illustrated as follows:

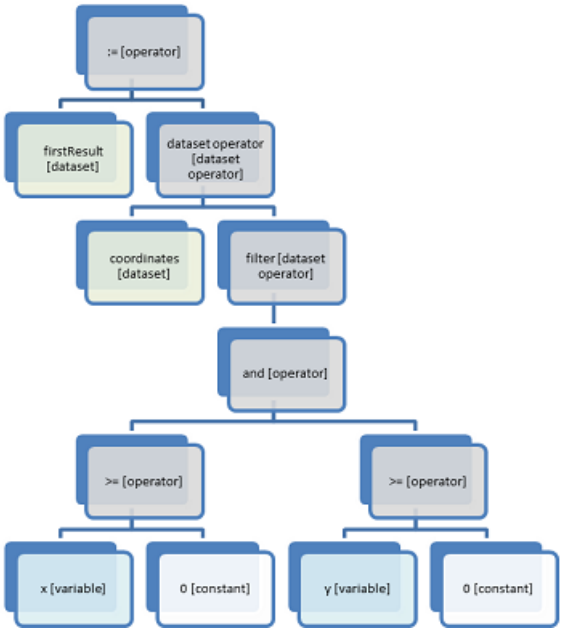

Figure 1: tree structure representation of result := coordinates [filter (x >= 0 and y >= 0)

The term written in brackets is the type of transformation element that can be used to identify constants and BIRD model objects (i.e. variables, cubes, etc.).

Note that the Boolean condition applied to the filter operator (i.e. “x>=0 and y>=0”) is completely decomposed into its components (i.e. transformation elements) in a structured way, in the sense that the Boolean condition can be re-engineered from this tree structure.

The decomposition of transformation into its transformation elements supports specific implementations of these transformations. For example, in the case of SQL implementation, we could apply the following mappings:

- := → CREATE VIEW _____ AS
- Filter → WHERE

Walk the tree and create the corresponding line of SQL code:

```
CREATE VIEW firstResult AS SELECT c.* FROM coordinates c WHERE x >= 0 AND y >= 0;
```

Note that, to generate such an SQL statement, one must also rearrange the tree’s nodes according to the SQL syntax. Also note that the elements after each keyword (i.e. CREATE VIEW, SELECT, FROM, WHERE) are similar to the elements represented in the tree structure.

Representation in the database

The transformation scheme is stored in the table TRANSFORMATION_SCHEME.

The module is stored in the MODULE table.

NAME	*CODE*	*DESCRIPTION*	*MODULE_ID*
Test module	TST_MDL	A test module to indicate how the derivation of the distance between x and y would look in the BIRD.	TST_MDL_1
Another test module	ANTHR_TST_MDL	Another test module that takes the results of the first test module to indicate how nodes of different modules are linked to each other.	ANTHR_TST_MDL_1

Note that this is a reduced version of the original table, presented for illustrative purposes.

Each individual transformation is stored in the TRANSFORMATION table:

NAME	*CODE*	*DESCRIPTION*	*TRANSFORMATION_ID*	*MODULE_ID*	*EXPRESSION*	*ORDER*
Filter	TST_MDL_1_0	Filter records where x and y are greater or equal to zero	TST_MDL_1_0	TST_MDL_1	/*filter all records of the data set coordinates where x and y are greater or equal to zero*/firstResult := coordinates [filter x>=0 and y>=0];	0
Keep	TST_MDL_1_1	Keep only x and y	TST_MDL_1_1	TST_MDL_1	/*keep only x and y*/secondResult := firstResult [keep x, y];	1
Calculate	TST_MDL_1_2	Calculate the distance between x and y	TST_MDL_1_2	TST_MDL_1	/*create a new variable (=column) named distance which will be populated with the square root of the sum of x squared and y squared*/result := secondResult [calc distance := sqrt(x*x+y*y)];	2
Filter greater than 11	ANTHR_TST_MDL_1_0	Calculate the distance between x and y	ANTHR_TST_MDL_1_0	ANTHR_TST_MDL_1	/*finally we create a data set where that only contains records where the distance is greater than 11*/finalResult := result [filter	0

					distance > 11];	

Using the MODULE_ID we can connect these transformations with the related module.

All transformation elements are stored in the TRANSFORMATION_NODE table:

TRANSFORMATION_NODE_ID	*PARENT_NODE_ID*	*TRANSFORMATION_ID*	*EXPRESSION*	*TYPE_OF_NODE*	*LEVEL*	*ORDER*
ANTHR_TST_MDL_1_0_0		ANTHR_TST_MDL_1_0	:=	OPERATOR	1	0
ANTHR_TST_MDL_1_0_0_0	ANTHR_TST_MDL_1_0_0	ANTHR_TST_MDL_1_0	finalResult	DATASET	2	0
ANTHR_TST_MDL_1_0_0_1	ANTHR_TST_MDL_1_0_0	ANTHR_TST_MDL_1_0	dataset operator	DATASET_OPERATOR	2	1
ANTHR_TST_MDL_1_0_0_1_0	ANTHR_TST_MDL_1_0_0_1	ANTHR_TST_MDL_1_0	result	DATASET	3	0
ANTHR_TST_MDL_1_0_0_1_1	ANTHR_TST_MDL_1_0_0_1	ANTHR_TST_MDL_1_0	filter	DATASET_OPERATOR	3	1
ANTHR_TST_MDL_1_0_0_1_1_0	ANTHR_TST_MDL_1_0_0_1_1	ANTHR_TST_MDL_1_0	>	OPERATOR	4	0
ANTHR_TST_MDL_1_0_0_1_1_0_0	ANTHR_TST_MDL_1_0_0_1_1_0	ANTHR_TST_MDL_1_0	distance	VARIABLE	5	0
ANTHR_TST_MDL_1_0_0_1_1_0_1	ANTHR_TST_MDL_1_0_0_1_1_0	ANTHR_TST_MDL_1_0	11	CONSTANT	5	1
TST_MDL_1_0_0		TST_MDL_1_0	:=	OPERATOR	1	0
TST_MDL_1_0_0_0	TST_MDL_1_0_0	TST_MDL_1_0	firstResult	DATASET	2	0
TST_MDL_1_0_0_1	TST_MDL_1_0_0	TST_MDL_1_0	dataset operator	DATASET_OPERATOR	2	1
TST_MDL_1_0_0_1_0	TST_MDL_1_0_0_1	TST_MDL_1_0	coordinates	DATASET	3	0
TST_MDL_1_0_0_1_1	TST_MDL_1_0_0_1	TST_MDL_1_0	filter	DATASET_OPERATOR	3	1
TST_MDL_1_0_0_1_1_0	TST_MDL_1_0_0_1_1	TST_MDL_1_0	and	OPERATOR	4	0
TST_MDL_1_0_0_1_1_0_0	TST_MDL_1_0_0_1_1_0	TST_MDL_1_0	=	OPERATOR	5	0
TST_MDL_1_0_0_1_1_0_0_0	TST_MDL_1_0_0_1_1_0_0	TST_MDL_1_0	x	VARIABLE	6	0
TST_MDL_1_0_0_1_1_0_0_1	TST_MDL_1_0_0_1_1_0_0	TST_MDL_1_0	0	CONSTANT	6	1
TST_MDL_1_0_0_1_1_0_1	TST_MDL_1_0_0_1_1_0	TST_MDL_1_0	=	OPERATOR	5	1
TST_MDL_1_0_0_1_1_0_1_0	TST_MDL_1_0_0_1_1_0_1	TST_MDL_1_0	y	VARIABLE	6	0
TST_MDL_1_0_0_1_1_0_1_1	TST_MDL_1_0_0_1_1_0_1	TST_MDL_1_0	0	CONSTANT	6	1
TST_MDL_1_1_0		TST_MDL_1_1	:=	OPERATOR	1	0
TST_MDL_1_1_0_0	TST_MDL_1_1_0	TST_MDL_1_1	secondResult	DATASET	2	0
TST_MDL_1_1_0_1	TST_MDL_1_1_0	TST_MDL_1_1	dataset operator	DATASET_OPERATOR	2	1
TST_MDL_1_1_0_1_0	TST_MDL_1_1_0_1	TST_MDL_1_1	firstResult	DATASET	3	0
TST_MDL_1_1_0_1_1	TST_MDL_1_1_0_1	TST_MDL_1_1	keep	DATASET_OPERATOR	3	1

TST_MDL_1_1_0_1_1_0	TST_MDL_1_1_0_1_1	TST_MDL_1_1	x	VARIABLE	4	0
TST_MDL_1_1_0_1_1_1	TST_MDL_1_1_0_1_1	TST_MDL_1_1	y	VARIABLE	4	1
TST_MDL_1_2_0		TST_MDL_1_2	:=	OPERATOR	1	0
TST_MDL_1_2_0_0	TST_MDL_1_2_0	TST_MDL_1_2	result	DATASET	2	0
TST_MDL_1_2_0_1	TST_MDL_1_2_0	TST_MDL_1_2	dataset operator	DATASET_OPERATOR	2	1
TST_MDL_1_2_0_1_0	TST_MDL_1_2_0_1	TST_MDL_1_2	secondResult	DATASET	3	0
TST_MDL_1_2_0_1_1	TST_MDL_1_2_0_1	TST_MDL_1_2	calc	DATASET_OPERATOR	3	1
TST_MDL_1_2_0_1_1_0	TST_MDL_1_2_0_1_1	TST_MDL_1_2	:=	OPERATOR	4	0
TST_MDL_1_2_0_1_1_0_0	TST_MDL_1_2_0_1_1_0	TST_MDL_1_2	distance	VARIABLE	5	0
TST_MDL_1_2_0_1_1_0_1	TST_MDL_1_2_0_1_1_0	TST_MDL_1_2	sqrt	OPERATOR	5	1
TST_MDL_1_2_0_1_1_0_1_0	TST_MDL_1_2_0_1_1_0_1	TST_MDL_1_2	+	OPERATOR	6	0
TST_MDL_1_2_0_1_1_0_1_0_0	TST_MDL_1_2_0_1_1_0_1_0	TST_MDL_1_2	*	OPERATOR	7	0
TST_MDL_1_2_0_1_1_0_1_0_0_0	TST_MDL_1_2_0_1_1_0_1_0_0	TST_MDL_1_2	x	VARIABLE	8	0
TST_MDL_1_2_0_1_1_0_1_0_0_1	TST_MDL_1_2_0_1_1_0_1_0_0	TST_MDL_1_2	x	VARIABLE	8	1
TST_MDL_1_2_0_1_1_0_1_0_1	TST_MDL_1_2_0_1_1_0_1_0	TST_MDL_1_2	*	OPERATOR	7	1
TST_MDL_1_2_0_1_1_0_1_0_1_0	TST_MDL_1_2_0_1_1_0_1_0_1	TST_MDL_1_2	y	VARIABLE	8	0
TST_MDL_1_2_0_1_1_0_1_0_1_1	TST_MDL_1_2_0_1_1_0_1_0_1	TST_MDL_1_2	y	VARIABLE	8	1

This structure helps the components of each transformation to be accessed easily. If, for example, we are interested in the operators that are used in the second line (*firstResult := coordinates [filter x >= 0 and y >= 0];* compare FIG 1), we simply select all rows where the TRANSFORMATION_ID equals TST_MDL_1_0 and restrict the result to those records where the TYPE_OF_NODE contains OPERATOR.

To understand the dependencies between these transformation nodes we need to analyse the information provided in the NODE_RELATIONSHIP table. For the given example, it contains the following information:

SOURCE_NODE_ID	*TARGET_NODE_ID*
ANTHR_TST_MDL_1_0_0_1_0	ANTHR_TST_MDL_1_0_0_0
TST_MDL_1_0_0_0	TST_MDL_1_1_0_1_0
TST_MDL_1_0_0_1_0	TST_MDL_1_0_0_0
TST_MDL_1_1_0_0	TST_MDL_1_2_0_1_0
TST_MDL_1_1_0_1_0	TST_MDL_1_1_0_0
TST_MDL_1_2_0_0	ANTHR_TST_MDL_1_0_0_1_0
TST_MDL_1_2_0_1_0	TST_MDL_1_2_0_0

Each row indicates that one or many datasets are input to another dataset. For example, the first row indicates that the dataset *result* (with TRANSFORMATION_NODE_ID =

ANTHR_TST_MDL_1_0_0_1_0) is an input for the dataset *finalResult* (with TRANSFORMATION_NODE_ID = ANTHR_TST_MDL_1_0_0_0). Please note that the sixth row represents a relationship between different modules.

Description of VTL artefacts

Functions/user-defined operators

VTL allows the available operators to be extended by defining functions. Functions take some variables as input and give a predefined calculation as a result.

The following example shows a function sued to calculate the carrying amount from the required input variables:

```
/*map: (Accounting classification, Fair value, Gross carrying amount excluding accrued interest, Accrued interest, Fair value changes due to hedge accounting, Accumulated impairment) → Carrying amount*/

define operator D_CRRYNG_AMNT(ACCNTNG_CLSSFCTN, FV, GRSS_CRRYNG_AMNT_E_INTRST, ACCRD_INTRST, FV_CHNG_HDG_ACCNTNG,    ACCMLTD_IMPRMNT)

returns string as

if (ACCNTNG_CLSSFCTN in ("2", "4", "8", "41")) then FV

elseif (ACCNTNG_CLSSFCTN in ("6", "14")) then (GRSS_CRRYNG_AMNT_E_INTRST + ACCRD_INTRST - ACCMLTD_IMPRMNT + FV_CHNG_HDG_ACCNTNG)

else null

end operator
```

This function can then be sued to derive new data:

```
RESULT := CUBE [calc Measure CRRYNG_AMNT := D_CRRYNG_AMNT(ACCNTNG_CLSFCTN, FV, GRSS_CRRYNG_AMNT_E_INTRST, ACCRD_INTRST, FV_CHNGS_HDG_ACCNTNG, ACCMLTD_IMPRMNT)];
```

The line illustrated above adds a column named “CRRYNG_AMNT” to the dataset CUBE, where the value of this new column for each row is determined by the function D_CRRNG_AMNT, and stores the result in a dataset named RESULT.

Transformation parser

We developed a parser for VTL to visualize tree structures and support the production of the output data model required for the BIRD database. The parser is written in Java, and available in GitHub.

Dependencies of transformation schemes

Owing to the fact that most of the transformation schemes depend on other schemes (in the sense that they use datasets that are generated in other schemes), we provide a graphical illustration of these dependencies for each transformation scheme following the header “Scheme dependencies” in the “Natural language” section. These dependencies can be computed from the transformation content in the database (i.e. the tables TRANSFORMATION_SCHEME, TRANSFORMATION and TRANSFORMATION_NODE).

Note that we implemented some restrictions to the transformation schemes that are taken into account when computing these dependencies: first, we do not include validation and put schemes in the dependency tree; and second, such a dependency tree does not contain any duplication of related schemes (although multiple schemes in the tree may depend on the same transformation scheme).