Annex II: 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 mode, a Transformation scheme is an ordered list of transformations. Therefore such a transformation scheme 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). Therefore a transformation element is a constant, an operation or a BIRD model object (i.e. a variable, cube, etc.). In case the transformation element represents an operation such a transformation element itself can have a relation to one or many transformation elements.

The BIRD database contains the complete information about transformation schemes in the sense that not only the decomposition of each transformation scheme into its transformations but also the decomposition of transformations into its transformation nodes according to the SDMX information model is stored in the database.

The next sections (“Example”, “Representation in the database”) explain the relation between transformation schemes, transformations and transformation nodes and their representation in the database. In the section “New VTL artefacts” we also describe functionality of VTL that is used regularly in various transformation schemes.

Example

The following example will try to clarify the current status of the representation of transformations in the BIRD database:

Let’s 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:

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

Using VTL syntax we would write the following lines:

```
/*extract all records from the (database-)table coordinates and store
the result in a dataset named "coordinates"*/

coordinates := get("coordinates");

/*extract all records from the dataset "coordinates" where x and y are
greater or equal to zero, keep only x and y and store the result in a
dataset named "result"*/
```

```
result := coordinates [filter (x >= 0 and y >= 0), keep (x, y)];

/*apply a calculation on the dataset "result" which takes the square
root of the sum of x squared and y squared and stores the result in a
new column (i.e. variable) named "distance"*/

finalResult := result [calc sqrt (x * x + y * y) as "distance"];
```

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

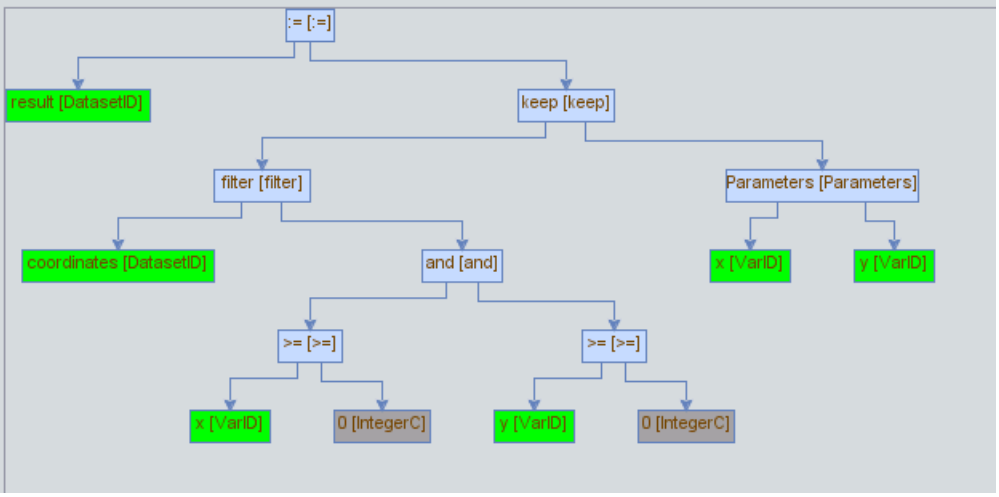

FIG 1: tree structure representation of *result := coordinates [filter (x >= 0 and y >= 0), keep (x, y)];*

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

Please 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 reengineered from this tree structure.

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

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

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

```
CREATE VIEW result AS SELECT x, y FROM coordinates WHERE x >= 0 AND
y >= 0;
```

Please note that – in order to generate such an SQL statement – one must additionally rearrange the nodes of the tree according to the SQL syntax. Please 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.

For the sake of completeness you find the tree representation of the first and second line here:

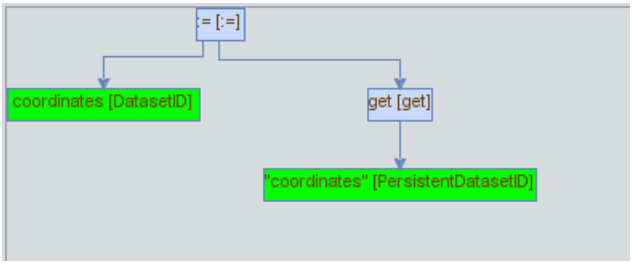

FIG 2: tree structure representation of *coordinates := get("coordinates");*

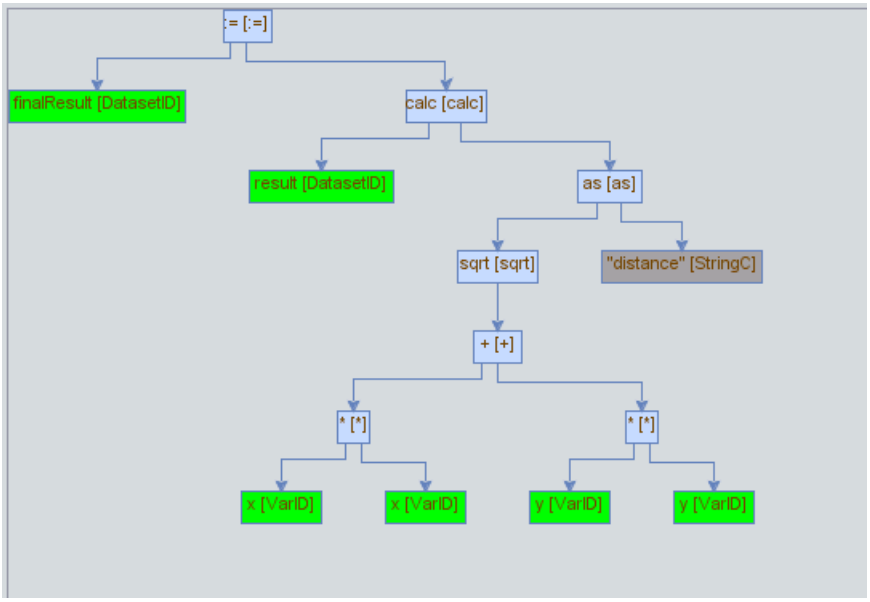

FIG 3: tree structure representation of *finalResult := result [calc sqrt(x * x + y * y) as "distance"];*

Representation in the database

The transformation scheme is stored in the table TRANSFORMATION_SCHEME:

SCHEME_ID	EXPRESSION	DESCRIPTION	NATURAL_LANGUAGE

TEST_SCHEME	/*extract all records from the (database-)table coordinates and store the result in a dataset named "coordinates"*/
	coordinates := get("coordinates");
	/*extract all records from the dataset "coordinates" where x and y are greater or equal to zero, keep only x and y and store the result in a dataset named "result"*/
	result := coordinates [filter (x >= 0 and y >= 0), keep (x, y)];
	/*apply a calculation on the dataset "result" which takes the square root of the sum of x squared and y squared and stores the result in a new column (i.e. variable) named "distance"*/
	finalResult := result [calc sqrt (x * x + y * y) as "distance"];
		Transformation scheme example	Transformation scheme deriving the distance between x and y

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

Each individual transformation is stored in the TRANSFORMATION table:

TRANSFORMATION_ID	EXPRESSION	SCHEME_ID	ORDER
156430	/*extract all records from the (database-)table coordinates and store the result in a dataset named "coordinates"*/ coordinates := get("coordinates");	TEST_SCHEME	0
156435	/*extract all records from the dataset "coordinates" where x and y are greater or equal to zero, keep only x and y and store the result in a dataset named "result"*/ result := coordinates [filter (x >= 0 and y >= 0), keep (x, y)];	TEST_SCHEME	1
156451	/*apply a calculation on the dataset "result" which takes the square root of the sum of x squared and y squared and stores the result in a new column (i.e. variable) named "distance"*/ finalResult := result [calc sqrt (x * x + y * y) as "distance"];	TEST_SCHEME	2

Using the SCHEME_ID we can connect these transformations with the related transformation scheme (which is similar to stating that “these transformations are children of the transformation scheme with SCHEME_ID ”TEST_SCHEME”).

All Transformation elements are stored in the table TRANSFORMATION_NODE:

TRANSFORMATION_ID	NODE_ID	EXPRESSION	TYPE_OF_NODE	LEVEL	PARENT	ORDER
156430	156431	:=	OperatorNode	0		
156430	156432	coordinates	ReferenceNode	1	156431	0
156430	156433	get	OperatorNode	1	156431	1
156430	156434	"coordinates"	ReferenceNode	2	156433	0
156435	156436	:=	OperatorNode	0		
156435	156437	result	ReferenceNode	1	156436	0
156435	156438	keep	OperatorNode	1	156436	1
156435	156439	filter	OperatorNode	2	156438	0
156435	156440	coordinates	ReferenceNode	3	156439	0
156435	156441	and	OperatorNode	3	156439	1
156435	156442	>=	OperatorNode	4	156441	0
156435	156443	x	ReferenceNode	5	156442	0
156435	156444	0	ConstantNode	5	156442	1
156435	156445	>=	OperatorNode	4	156441	1
156435	156446	y	ReferenceNode	5	156445	0
156435	156447	0	ConstantNode	5	156445	1

156435	156448	Parameters	OperatorNode	2	156438	1
156435	156449	x	ReferenceNode	3	156448	0
156435	156450	y	ReferenceNode	3	156448	1
156451	156452	:=	OperatorNode	0		
156451	156453	finalResult	ReferenceNode	1	156452	0
156451	156454	calc	OperatorNode	1	156452	1
156451	156455	result	ReferenceNode	2	156454	0
156451	156456	as	OperatorNode	2	156454	1
156451	156457	sqrt	OperatorNode	3	156456	0
156451	156458	+	OperatorNode	4	156457	0
156451	156459	*	OperatorNode	5	156458	0
156451	156460	x	ReferenceNode	6	156459	0
156451	156461	x	ReferenceNode	6	156459	1
156451	156462	*	OperatorNode	5	156458	1
156451	156463	y	ReferenceNode	6	156462	0
156451	156464	y	ReferenceNode	6	156462	1
156451	156465	"distance"	ConstantNode	3	156456	1

This structure supports easy access to the components of each Transformation. If, for example, we are interested in the operators that are used in the second line (*result := coordinates [filter (x >= 0 and y >= 0), keep (x, y)];* compare FIG 1) we simply select all rows where the TRANSFORMATION_ID equals 156435 and restrict the result to those records where the TYPE_OF_NODE equals OperatorNode. The result reflects the blue squares in FIG 1.

Please note that not only Transformations are represented in this structure but also Functions, Datasets and Procedures.

New VTL artefacts

Procedures

Procedures are aimed at automating common processing tasks, and can be used as a means for shortening the code by replacing common processing tasks with a procedure call.

Procedures take input and output arguments and describe the set of transformations performed with those arguments.

The following example shows a procedure for checking the identifiers that are present in a cube (FRGN_CB) but are not present in another cube (PRMRY_CB).

```
define procedure PRCDR_RFRNTL_INTGRTY(input FRGN_CB as dataset, input
FRGN_VRBL as string, input PRMRY_CB as dataset, input PRMRY_VRBL as string,
input VLDTN_ID as string, output RSLT as dataset) {
```

```
        /*extract a set of Foreign variable (FRGN_VRBL), rename to ID, store
        in dataset Foreign identifiers (FRGN_IDS)*/

        FRGN_IDS := FRGN_CB[keep (FRGN_VRBL), rename FGN_VRBL as "ID" ];

        /*extract a set of Primary variable (PRMRY_VRBL), rename to ID, store
        in dataset Primary identifiers (PRMRY_IDS)*/

        PRMRY_IDS := PRMRY_CB[keep (PRMRY_VRBL), rename PRMRY_VRBL as "ID" ];

        /*calculate the set difference between Foreign identifiers (FRGN_IDS)
        and Primary identifiers (PRMRY_IDS)*/

        RSLT :=  setdiff (FRGN_IDS, PRMRY_IDS);

        /*rename ID to FRGN_IDS, make VLDTN_ID a measure variable*/

        RSLT := RSLT [rename (ID as "FRGN_IDS", VLDTN_ID role Measure)];

}
```

The procedure can be called afterwards with concrete arguments:

```
call PRCDR_RFRNTL_INTGRTY(TRNSCTNS_CNTRPRTS, CNTRPRTY_ID, CNTRPRTS,
CNTRPRTY_ID, V_TRNSCTNS_CNTRPRTS_ID);
```

Functions

VTL allows extending the available operators by defining functions. Functions take as input some variables, and give as a result a predefined calculation. Currently the BIRD uses functions as linear maps n:1 maps creating one output value while taking into account n input parameters.

The following example shows a function 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*/

create function D_CRRYNG_AMNT(ACCNTNG_CLSSFCTN, FV,
GRSS_CRRYNG_AMNT_E_INTRST, ACCRD_INTRST, FV_CHNG_HDG_ACCNTNG,
ACCMLTD_IMPRMNT)  {

returns

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

as integer}
```

This function can be then used to derive new data:

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

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.

Rulesets

Rulesets define validation rules between variables that have to be applied to each individual record of a given dataset. Rulesets take as input the variables to be validated, and contain at least one consistency rule that the validations need to comply with. Each rule has two conditions (introduced by the clauses when and then), and the validation will be satisfied if both conditions are satisfied.

As an example, the following ruleset includes two consistency rules between the variables accounting classification and accumulated changes in the fair value due to credit risk

```
define datapoint ruleset DR_ACCMLTD_IMPRMNT1(ACCNTNG_CLSSFCTN,
ACCMLTD_IMPRMNT) {

RL1:

      when ACCNTNG_CLSSFCTN in ("2", "4", "41")

      then isnull(ACCMLTD_IMPRMNT)

      errorcode("Instruments classified as 'IFRS: Financial assets held for
      trading (2)', 'IFRS: Financial assets designated at fair value through
      profit or loss (4)' or 'IFRS: Non-trading financial assets mandatorily
      at fair value through profit or loss (41)' are not subject to
      impairment");

RL2:

      when ACCNTNG_CLSSFCTN in ("6", "8", "14")

      then not isnull(ACCMLTD_IMPRMNT)

      errorcode("For instruments classified as 'IFRS: Financial assets at
```

```
      amortised cost (6)', 'IFRS: Financial assets at fair value through
      other comprehensive income (8)', 'IFRS: Cash balances at central banks
      and other demand deposits (14)' the 'Accumulated impairment' should
      not be null");

}
```

The ruleset can then be used with the check operator:

```
VALIDATION_RESULT := check (DATASET, DR_ACCMLTD_IMPRMNT1);
```

Transformation parser

We developed a parser for VTL in order 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.