Guidelines for the homogenization of VTL 1.1 syntax

1 Introduction

This document collects some remarks and guidelines in terms of syntax and style of VTL 1.1 operators. The document is organized so that in each section a specific problem is described along with indications for the solution. Although the issues and the solutions are described w.r.t. specific cases, various examples are provided and the solutions are thought to be discussed so as to be adopted throughout the whole document in all the cases to which they apply, with the goal to improve the overall coherence.

2 Heterogeneous styles for operators

As far as the operators are concerned, VTL 1.1 has the following syntactic styles: 1. *functional*; 2. *SQL DML-like*, 3. *SQL DDL-like*; 4. *PL/SQL-like style* 5. *SQL/functional hybrid*, 6. *infix*, 7. *postfix*.

The functional syntax is mainly adopted in VTL-ML; it is minimalistic as the operator name is simply followed by the list of values for the various parameters enclosed by round parentheses.
SQL-like style is characterized by the presence of reserved words prefixing and suffixing parameter values. This style is adopted in both VTL-DL and VTL-ML. For example in `defineDataset` we have the reserved words `IDENTIFIER`, `MEASURE` that define the role of specific components, recalling the type declaration in SQL, which follows the names of the components. The same in analytic functions (and many more) where we find reserved words such as `partition by` or `preceding`.
PL/SQL-like style is adopted in some declarations (e.g. rulesets or functions), and is mainly denoted by the use of `end` to limit the declaration block, instead of parentheses.
Hybrid style refers to the adoption of a functional notation together with special reserved words giving specific meaning. Examples are the analytic functions or the recently proposed `join` operator, where we have the keyword `on` to denote the join components. Operator `check` is also such an example, where `imbalance` and `errorlevel` act as SQL-like reserved words, yet even abuse the functional notation.
Infix and postfix syntaxes are used in arithmetic operators and clauses.

The language should try to minimize the number of heterogeneous styles for the operators, while guaranteeing an overall high readability and usability.

2.1 Proposal

Since the language consists of two kinds of operators, *definition* (VTL-DL) and *manipulation* (VTL-ML), it appears that a distinction in style between the two parts is important and acceptable. Instead, within the single parts, the differences should be minimized. The suggestion is the following:

- For VTL-DL use a SQL DDL-like style, i.e., maintain the current syntax and avoid a PL/SQL-like syntax (blocks should be limited by parentheses). This part of the language is thought to be somehow non-functional, thus the DDL style appears acceptable and the proposals in the following sections do not apply.
- For VTL-ML use as much as possible the functional notation. Avoid any kind of SQL-like or hybrid style. Maintain infix and postfix styles. Refer to the next sections for more details.

Postfix notation (clauses) plays an important role in the language as it covers operators that to some extent perform *structural manipulation*: `calc` can add new components; `keep` and `drop` can add/remove components and so on. For this reason the postfix notation must be preserved. As a

simplification, we suggest that `calc` keyword in the clause is removed and made implicit. Accordingly, an example of invocation would be:

$ds_1[v_4 := v_3 + v_1]$

In some sense, the implicit `calc`, is just the assignment of new values for the components, which justifies the adoption of the assignment symbol. We also confirm the validity of the adoption of a more compact syntax for multiple applications. Specifically, the following two examples are equivalent:

$ds_1[v_4 := v_3 + v_1, v_5 := v_3 + v_2]$

$ds_1[v_4 := v_3 + v_1][v_5 := v_3 + v_2]$.

3 Operators: CamelCase vs underscore_style, UPPER vs lowercase

The names of some operators adopt a CamelCase style, i.e., the practice is writing a compound word by separating the words with capital letters. An example is `defineDataset`. Other operators have compound names that are separated by underscore. There is a variety of examples, including time series operators (e.g., `flow_to_stock`), statistical and analytic functions (e.g., `first_value`, `last_value`), Boolean operators (e.g., `match_characters`).

The described heterogeneity hampers the look&feel of the language and gives an overall idea of a custom and not refined approach.

3.1 Proposal

As the underscore_style appears a more adopted convention in functional languages, whereas the CamelCase one is more in the domain of object-oriented languages, the suggestion is to adopt the former for all VTL operators. We also suggest that the names of the operators are always *lowercase*.

4 Heterogeneous styles for parameters in functional operators

The language adopts heterogeneous syntactic approaches for the parameters of functional operators.

Positional parameters: where the formal parameters can be individuated by the position in the parameter list of the operator. An example is `power`(ds, exponent), where the first parameter is the dataset and the second is the exponent.

Named parameters: where the syntax foresees the indication of the formal parameter name before the actual parameter and the equality symbol is used. Moreover, the name of the formal parameter is optional or mandatory without any clear rationale.
An example is ds_1[`rename` k `as` k' `role` = `MEASURE`], where the role of the renamed component is denoted by the formal parameter name `role`. Another example is `timeshif`$(ds, timeId, \texttt{unit} = A, ...)$, where the formal parameter name `unit` introduces the parameter denoting the unit of measure. Another example is `eval`, where there are explicit named parameters for the script, the parameters for such script and the dataset name.

Functional style parameters: where the syntax suggests the adoption of a function to define the parameter, even if no function is actually implied. An example is the specification of `imbalance`, `errorcode`, `errorlevel` in the `check` operator, e.g.: `check`$(ds, \texttt{imbalance}(5))$. Here the imbalance resembles a function application on the actual parameter 5, whereas we only assign the value 5 to the imbalance component. Another example is the `GET` operator, where `keep`, `filter` and others have a

functional style. Another example is `union` where the `dedup` parameter looks like a function invocation.

Reserved words parameters: where the syntax introduces parameters with some reserved words of the operator. This case is different from the *named parameters* one since the equality symbol is not used and the keyword, always mandatory, is not necessarily the name of the parameter. An example is `rename` cmp_1 `to` cmp_2, where the keyword `to` introduces the second operand of the renaming. Another important example is represented by the *analytic functions*, where there is massive use of keyword parameters, e.g.: `partition by`, `order by`, `between`, `precedng`, `following`. Clauses are another example, where `role` is used to introduce an untyped value (see Section 5) denoting the role of a component. The "definition" parts of the language also adopt this notation for parameters, e.g. declaration of functions, rulesets, datasets, domains, etc.

Indeed, the described styles make the language look like a patchwork of different sub-languages and differences should be mitigated before the official release. We should try to balance the adoption of the various styles and adhere to a general rationale. Positional parameters are not sufficient and must be supported by named parameters in order to handle optional parameters. For example, let us consider the `check`. It has two optional parameters, namely *imbalance* and *errorlevel*. Conversely, the first parameter, *dataset*, is mandatory. If we simply adopted the functional style, the following expression would be ambiguous: `check`$(ds_1, 1)$, since the constant 1 could be bound to either imbalance or errorlevel. This motivates the fact positional parameters are not enough and must be supported by named parameters to disambiguate cases.

4.1 Proposal

A possible rationale to uniform the language is exemplified by the following prototypical invocation.

`operator_name`$(p_1, \ldots, p_m, name_1 := p_{m+1}, \ldots, name_{m+k} := p_{m+k})$

$p_1, \ldots, p_m$ are symbols that represent the values for the m mandatory parameters of the operator, while $p_{m+1}, \ldots, p_{m+k}$ are symbols that represent the values for the k optional parameters. The rationale is: 1. the mandatory parameters precede the optional parameters and are individuated positionally; 2. the optional parameters follow the mandatory parameters and are individuated by their names and the assignment symbol := is used for this purpose. In the invocations, mandatory parameters must be specified in the correct order and need not be denoted by their name, though this can be anyway done for the sake of readability. Optional parameters always follow the mandatory ones and must be denoted by their name to avoid ambiguities.

For example:

`check`$(ds_1, $`imbalance`$:= 1, $`errorlevel`$:= 2)$
`check`$($`dataset`$:= ds_1, $`imbalance`$:= 1, $`errorlevel`$:= 2)$

would be correct and unambiguous invocations of the `check`. Observe that since the optional parameters are denoted by their name, the order is not relevant, e.g., the example above is equivalent to

`check`$(ds_1, $`errorlevel`$:= 2, $`imbalance`$:= 1)$.

In this example, more complex cases where the `check` accepts a complex expression to calculate new measures, would be covered as well:

`check`$(ds_1, $`imbalance`$:= (V_1 + V_2)/V3, $`errorlevel`$:= 2)$

In the manual, operators could be described by the following prototypical signatures:

`operator_name`($par\_name_1, \ldots, par\_name_m\{, par\_name_{m+1}, \ldots, par\_name_{m+k}\}$)

where $par\_name_1, \ldots, par\_name_m$ are the names of the mandatory parameters (to be bound position-wise, while the indication of the name in the invocation is optional) and
$par\_name_{m+1}, \ldots, par\_name_{m+k}$ are the names of the optional parameters (to be bound name-wise independently of the order).

Possible dependencies in the optionality of parameters, i.e., a parameter must (not) appear if others do (not), should be detailed in the textual explanations (avoiding nested curly brackets in the meta-syntax) as well as the default values (avoiding hardly readable default values in the meta-syntax). With respect to the `check`, we would have:[1]

`check`(`dataset`, {, `threshold`, `imbalance`, `errorcode`, `errorlevel`)}.

As a further simplification, we also suggest the adoption of a special convention for Boolean parameters with the goal of improving readability: whenever the value `TRUE` is intended for a Boolean parameter passed by name, it is sufficient to pass the parameter name. Consider, for example the following invocation (simplified with what proposed in Section 5).

`alter_dataset`(ds_1, `keep_all` = TRUE).

It can more simply be written as:

`alter_dataset`(ds_1, `keep_all`)

and the `TRUE` value for `keep_all` is implied.

5 Unidentified Untyped Objects

Often operators include syntactic objects (seemingly strings without quotes or constants or optional reserved words), that specify properties or options. These objects are not coherent with the VTL typing system and thus inflate the set of reserved words. There are many examples. Let us refer to the `check` operator and, specifically, to the following syntactic objects: `not valid`, `valid`, `all`, `measures`, `conditions`. They are neither strings, as there are not quotes, nor parameter names (see Section 4). They are pure syntactic elements specifying options for the operator. Another interesting example is in the `hierarchy` operator, where the objects `sum` and `prod` are two possible options for the `aggregation` parameter. Several cases are present in analytic functions: `asc`, `desc`. We also mention the reserved words denoting roles in clauses, namnely `role` in `calc`.

The main reason for the presence of such syntactical objects is the following: specifying a custom behavior for the operator (or for some parts thereof), based on a standard list of pre-defined behaviors. Observe that these syntactic objects are not named parameters nor keyword parameters (see Section 4) and their presence does not directly depend on the adoption of the SQL-like or hybrid style.

5.1 Proposal

The suggestion is replacing such options with optional Boolean, numeric or string parameters (for the style of optional parameters refer to Section 4). Optional Boolean parameters should replace optional syntactic objects, which currently alter the behavior of the operator depending on their presence.

For example, referring to `check`, let us consider the `valid`, `not valid`, `all` syntactic objects that allow to configure which data points should be returned by the operator. In this case, we would introduce a new optional string parameter `return_only`, which can take three different string values: "valid", "not valid" and "all". The same could be done to handle `measures` and `conditions`, by

[1] Some parameters, or better, options, of the `check` operator are omitted because the style problems are dealt with in Section 5.

introducing a new string parameter `output_components`, which can take two possible string values: "measures" and "conditions". A prototypical invocation of the check than would be:

`check`(ds_1, `imbalance` := 1, `errorlevel` := 2, `return_only` := "*valid*", `output_components` := "*measures*").

Observe that the invocation is coherent with the proposal in Section 4, as the optional parameters follow the mandatory ones and their order is irrelevant. Another example is the `alterDataset` operator, which we will call `alter_dataset`, according to the proposal in Section 3. The syntactical object `all` specifies that all the components must be kept in the result dataset. This can be easily replaced by a Boolean optional parameter `keep_all`, with a prototypical invocation like the following:

`alter_dataset`(ds_1, `keep_all` := TRUE).

As a final remark, observe that the default behavior of such optional parameters should be conveniently chosen so as to minimize the need to resort to them. Such behavior should be explained in the text in order to keep the meta-syntax as light as possible.

6 Varargs (functions with variable number of arguments)

There are cases in which functional operators take as input a varying number of parameters of the same type. The actual number of parameters is determined only at runtime. On the other hand, there are operators that handle multiple parameters as lists, keeping the number of formal parameter fixed and well-defined and therefore determined at compile time. An example of vararg operator is `union`, which takes as input a varying number of datasets. An example of operator that handles multiple operators with lists is `funcDep` (which according to what proposed in Section 3 should be called `func_dep`).

The presence of varargs makes the syntax complex and even very hard (if not impossible) to parse. Moreover, the adoption of different styles in the language must be avoided.

6.1 Proposal

The suggestion is to forbid varargs and always use *lists* of values. This has the advantage of having well-defined and fixed-length signatures for the operators, easier to read and parse. The variability is hidden in the lists, which can of course be of variable length. As an example, consider the following invocation of the `union` with the proposed solution:

`union`($[ds_1, ds_2, ds_3]$)

where the square brackets denote the list (one single parameter) of datasets.

Observe that replacing varargs with lists affects mostly clauses as apparent from the following examples:

ds_1[`keep` $[comp_1, comp_2, comp_3]$]

ds_1[`drop` $[comp_4]$]

where the components to be kept/dropped in the projection are specified as a single list parameter.

7 Uppercase vs Lowercase

With reference to the problem discussed in Section 5, we observe that the unidentified syntactical objects (which we propose to replace with values of optional parameters) are sometime uppercase

(like for `MEASURE`, `IDENTIFIER`) and sometimes lowercase (like for `all` or `valid`).

7.1 Proposal

We propose to complement the suggestion in Section 5 by always adopting lowercase values for this kind of optional parameters as this appears to be the standard in functional languages.