Working Paper.

UNITED NATIONS
ECONOMIC COMMISSION FOR EUROPE

CONFERENCE OF EUROPEAN STATISTICIANS

Work Session on Statistical Data Editing
(The Hague, Netherlands, 24-26 April 2017)

Technical aspects of VTL to SQL translation

Prepared by Regional Statistical Office in Olsztyn, Poland

I. Introduction

A. Background

1. VTL (Validation and Transformation Language) is a new international standard language for defining validation and transformation rules on statistical data. The language is designed for business users without technical skills and is independent of the statistical domain. It is also independent of content and typologies of data, phase of the statistical process or technical infrastructure. VTL has an active role for data processing, therefore it should be interpreted and executed in data processing system.

2. Translation from VTL to typical data processing languages is not a trivial task and requires the creation of specialized software tool. Regional Statistical Office in Olsztyn (Poland) approaches to develop such source-to-source compiler, called VtlProcessingLib, which will be used in Polish data processing system SPDS to validate data and aggregates. Implementing VTL is also one of the topics of ESSnet ValiDat Integration project in context of executing validation rules prior to data transmission to Eurostat.

3. This document describes the technical issues related to the creation of this kind of translator.

B. Different versions of VTL

4. At the beginning of 2015 a specification of VTL version 1.0 was publically released by SDMX Technical Working Group. VTL 1.0 has some design flaws and its usefulness for data validation purposes was questioned (Gelsema, 2015).

5. In October 2016 a draft for public review of VTL version 1.1 was released. The updated specification introduce a completely new definition language (VTL-DL) while manipulation language (VTL-ML) has been modified. This version allows creating reusable artefacts and rules with the ability to define modules, procedures and functions. All language operators are divided into “core” and “standard library”. This involves a number of complications in the design of the translator relative to the 1.0 VTL version. Definition language (VTL-DL) needs to be implemented. The ability of the language to define reusable components makes it necessary to implement mechanisms for their linkage to the translated code. It also complicates the development of the intermediate representation.

II. Project overview

A. Product description

6. The VtlProcessingLib project’s scope is to create a programming library that allows for translating source code of VTL to target code of different programming languages. An executable command

line program and simple UI application for testing and presentation purposes will be also developed.

7. The initial functionality of the library will be as follows:

   (a) Generation of output code of target programming language (initially T-SQL)

   (b) Optimization and type checking of input VTL compilation unit[1]

   (c) Generation of compiled VTL modules from their source codes to be persistently stored for later reuse

   (d) Analysis of input VTL program or module for list of compile-time errors

   (e) (Optional) Generation of target language (initially T-SQL) code for detecting runtime errors (e.g. runtime type checking, …)

8. It will feature a modular architecture that allows extending functionality by adding new:

   (a) Supported VTL versions and dialects

   (b) Individual standard library operators, functions or procedures

   (c) Supported target languages (e.g. SAS, R, C#)

   (d) Components for optimizing and transforming VTL compilation units

9. The library would allow for wide range of possible usage. It could be used to build more extensive solutions, for example:

   (a) Stand-alone applications managing and running VTL programs

   (b) Development environments for defining VTL rules

   (c) Shared services for validating and running VTL programs

B. Implementation background

10. A computer program that performs such translation of source code of one high-level language to target code of a different high-level language is called a translator or source-to-source compiler. The top-level architecture of VtlProcessingLib is based on a classical approach for writing compilers. A typical multi-pass compiler architecture consists of three stages [Figure 1].

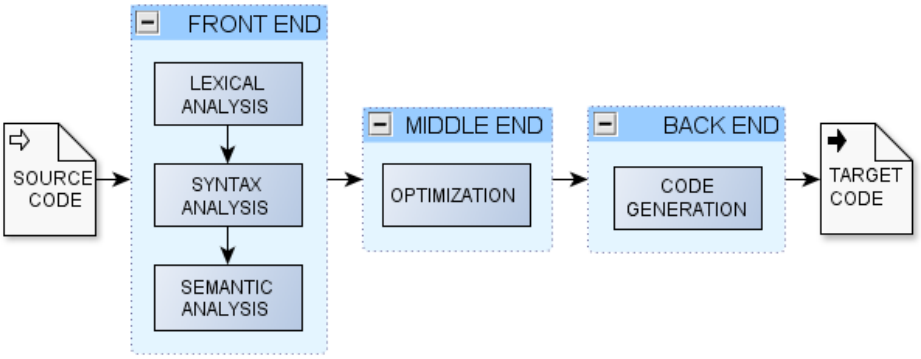

Figure 1 Typical compiler stages

   (a) FRONT END – verifies syntax and semantics according to a specific source language. It includes lexical analysis, syntax analysis and semantic analysis stages. Eventually generates an intermediate

[1] Compilation unit – a unit of code stored in a single file or transmitted as a message. Programs and modules are two types of compilation units in VTL. (SDMX Technical Working Group, VTL Task Force, 2016)

representation (abbreviated as IR) of the source code for processing by the middle-end. This IR is usually a lower level representation of the program with respect to the source code.

(b) MIDDLE END – performs transformations and optimizations on intermediate representation of the source code. This allows for transformations which are independent to both source and target languages syntax. Middle-end produces modified/optimized IR for further processing.

(c) BACK END – takes the output from the middle-end, then generates the target language code. Some more optimizations specific to target language might be done at this stage.

C. Development tools

11. The solution is being developed under Microsoft .NET platform and will be distributed as a package of .Net assemblies. This would allow for developing further software on top of the VTL translator.

12. In order to make implementation easier and more maintainable, some existing proven-to-work tools were used. Lexical and syntax analysis is realized entirely with some code automatically generated by parser generator called ANTLR[2]. Base code for Semantic analysis is also generated with ANTLR tool. This was a quite natural choice as VTL specification is supplied together with formal description of language syntax in eBNF[3] notation. Such file can be passed directly to ANTLR for parser code generation, optionally with some minor tweaks.

13. Some of the middle-end's IR optimizations may be realized with specific graph transformation tools. GrGen.NET[4] toolset was proposed to be used as a core engine for IR representation and transformation. The decision on using GrGen was not made yet at the time of writing this paper.

14. The target code generation stage is based on template engine tool called StringTemplate[5]. The usage of such a tool enables the separation of application logic, input data structure (IR) and target code formatting. The tool is well suited especially for generating source code of programming languages. It takes set of template files and attribute objects to generate structured text output (target code).

III. Technical architecture

A. Translation process overview

15. Compiling the VTL source code (program or module) proceeds in a sequence of specific phases. The result of each phase becomes the input for the next phase. VtlProcessingLib is intended to translate from many different dialects of VTL to many different target languages. In order to avoid developing separate translator for every such combination, all translation phases are grouped into three separate parts: front-end, middle-end and back-end. Different front-end implementations are needed for supporting different dialects of VTL (versions 1.0 and 1.1). This stage always produces a common form of intermediate representation called ICU for Intermediate Compilation Unit, which is in turn transformed and optimized by common middle-end. ICU object is then optimized and transformed several times. Compilation unit dependencies are included and optimizations are performed. The back-end of the translator slightly differs from the classical approach as there is only one such component. The target language specific part is supplied to the back-end as replaceable modules.

---

[2] http://www.antlr.org/

[3] Extended BNF - notation for specifying the syntax of a linear sequence of symbols. It has applications in the definition of programming and other languages, as well as in other formal definitions. (ISO/IEC 14977, 1996)

[4] http://www.info.uni-karlsruhe.de/software/grgen/

[5] http://www.stringtemplate.org/

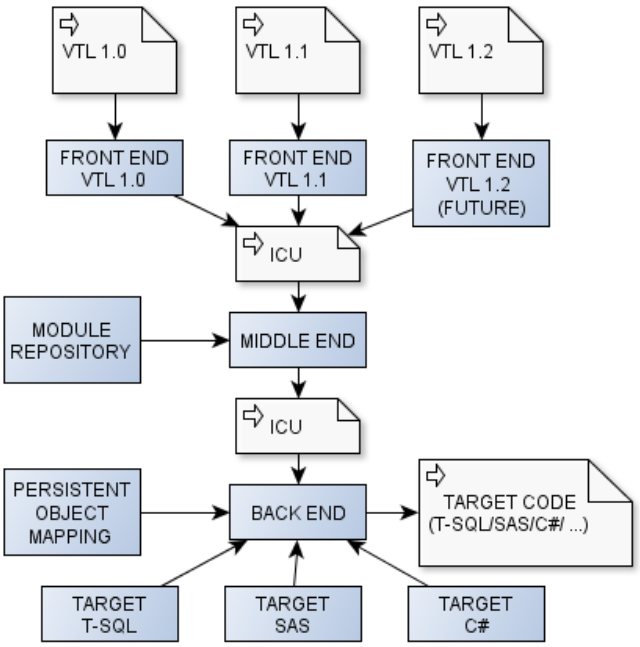

Figure 2 Modular top-level architecture of VTL translator

B. Key design choices

16. VTL compilation unit (program or module) is internally represented by ICU structure. Initially it has all information contained in compilation unit source code and it is gradually enriched and optimized by subsequent phases.

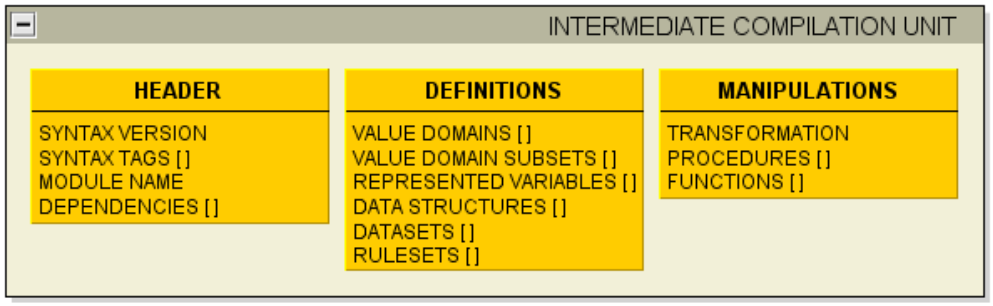

Figure 3 Intermediate representation contents

17. Intermediate Compilation Unit contents can be divided into three groups:

(a) HEADER – It contains all the information about compilation unit itself. That consists of a version of VTL syntax, a module name and a list of dependencies. Some more compilation unit metadata may be added at later stages of development.

(b) DEFINITIONS – This is a collection of persistent artefacts of VTL information model defined in compilation unit source code with VTL model definition language. It is primarily used by VTL "get" operator to acquire input data structures.

(c) MANIPULATIONS – This is information about data manipulations performed by compilation unit. It contains a collection of graph structures, each of which corresponds to a single VLT function, procedure or transformation. These graphs are the objects for optimizations and transformations mostly done in the middle-end phase of translation.

18. This differs from the classical approach for compiler intermediate representations in some areas. All the information about a particular node in transformation tree is available locally instead of depending on global symbol tables. That means each transformation contains all information about itself, such as its operation type, result data structure and even type inference logic. This is a more object-oriented approach for translator design. After passing all the ICU-transforming phases the transformation graph structure contains all the information needed for executing transformations

on actual datasets. The other parts of ICU are not needed anymore and might not be passed to the back-end at all.

19. The important design choice is that all functions and procedures occurrences in the translated code are inlined [6]with their definitions. It greatly simplifies the entire process of translation and allows to perform cross-module optimizations, which is a great benefit. Processing of a single compilation unit needs to load only its direct dependencies, not the entire dependency tree. Also the target code generation stage may be much simpler.

C. Implementation details

20. The following diagram [Figure 4] illustrates the architecture of VtlProcessingLib translator library.

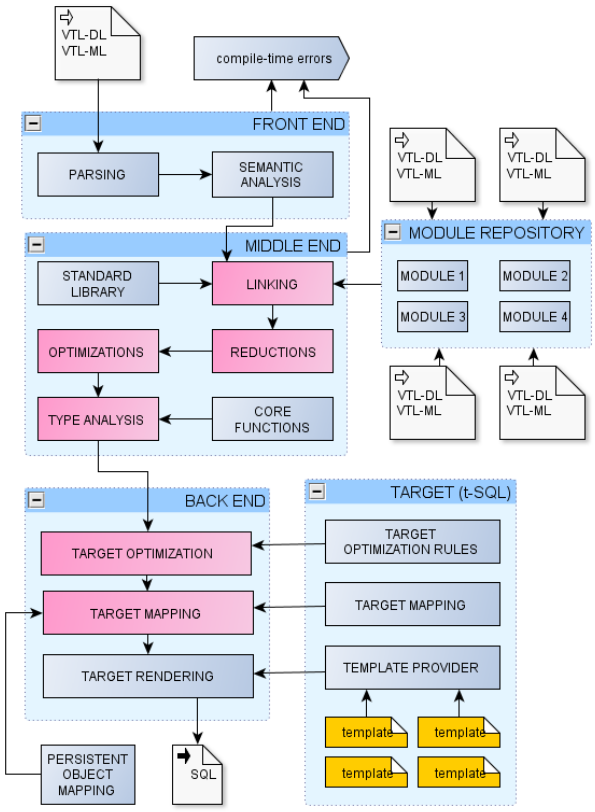

Figure 4 VTL translation stages

21. The translation process consists of several stages:

(a) PARSING – This is the starting point of translation. Parsing is the process of analysing a source code of input VTL program. It internally consists of two phases: lexical and syntactic analysis. The result of this stage is a CST[7] tree structure that represents the syntactic structure of input VTL source code. It is done by VTL lexer and parser generated automatically by ANTLR tool.

(b) SEMANIC ANALYSIS – This is a process of transformation of CST from previous stage to the form of Intermediate Compilation Unit (ICU) structure. It is done by a component which implements a visitor design pattern. That means it walks through the parse tree to build ICU object in the process. The base mechanism of such visitor is also automatically generated by ANTLR.

(c) LINKING – If the compilation unit uses some external modules, their code must be merged. This process involves loading needed modules from a module repository and renaming variables,

[6] Function inlining - Defining a member function's implementation within the class where it was also declared. This is usually reserved for small functions since the inline function must be re-compiled for every instance of the class. (The Free On-line Dictionary of Computing, 2017)

[7] CST – Concrete syntax tree or Parse tree. It is an ordered, rooted tree that represents the syntactic structure of a string according to some context-free grammar. (Raymond, 1996)

functions and procedures to avoid naming conflicts. All loaded modules are then joined into the main ICU structure and passed to the next translation stage.

(d) REDUCTIONS – Functions and procedures (also these from the VTL standard library) used in the compilation unit are inlined at this stage. All occurrences of functions and procedures in the transformation graph are simply replaced by theirs bodies. Any usage of "syntactic sugar"[8] is treated in a similar way and replaced with simpler equivalents. This produces a reduced form of ICU which uses only a limited set of VTL core constructs, so further transformation will be much simpler.

(e) OPTIMIZATIONS – This involves a graph pattern matching algorithms, which replaces some specific patterns found in the transformation graph by its optimized equivalents. Some more specific optimisations might be also performed at this stage, like dead or unreachable code removal. It works on already reduced ICU graph structure, which uses only core instructions. Every single optimisation must preserve correctness, thus produces an equivalent but optimized form of ICU.

(f) TYPE ANALYSIS – At this stage information about strict types of all transformation's results is inferred based on structures of persistent datasets stored in ICU's definitions part. Type inference logic is defined by core functions, which describe the behaviour of VTL core language constructs. This is done as the last step of the common middle-end ICU transformations because VTL procedures and functions might not be strictly typed and need to be inlined before analysis of data typing. Some compile-time type errors can also be found at this stage.

(g) TARGET OPTIMIZATION – Some more optimizations may be performed at this stage. This is needed to exploit some specific target language optimization possibilities. For example some operations might be grouped together and rendered as one operation of the target language. Optimization rules are defined separately for every target language in a separate module.

(h) TARGET MAPPING – Every target programming language may have different constraints on identifier naming. All temporary dataset and structure component names are mapped to the form which is acceptable by the target. Such mapper is also defined in a separate module, different for every target language.

(i) TARGET RENDERING – This takes the final form of ICU and produces an executable code of target programming language. A template engine is used for this task. Renderer itself contains only template for general structure of the output code. The collection of needed templates for every VTL core construct are supplied by the target module.

22. Stages 3 to 8 alter ICU object in place, so may be chained as a sequence of ICU transformations. Each of the transformations returns equivalent form of its input ICU object. This allows for easily adding new translation stages or reorganizing their order in the future.

D. Potential issues

23. Aggressive inlining strategy described above might have some serious performance flaws due to replication of code. It results in bigger file sizes of compiled modules. This is acceptable because functions and procedures aren't strictly typed and most of function occurrences would be unique anyway. Every module has its entire dependency tree "compiled in" it. That means the compilation process needs to load only these modules which are directly referenced by a compilation unit. However, if there is a change in one module, all dependant modules need to be recompiled. In a pessimistic scenario this might lead to recompiling hundreds of modules after a single code change. It results in the need for automated dependency tracking, which will not be covered by the project to develop VtlProcessingLib. A modules version management will be also needed. Some additional external tools are needed for these tasks.

[8] Syntactic sugar – [coined by Peter Landin] Features added to a language or other formalism to make it `sweeter' for humans, but which do not affect the expressiveness of the formalism. (Raymond, 1996)

IV. Work progress

24. Before attempting to implement a full-featured VTL 1.1 translator, a prototype software was made at Regional Statistical Office in Olsztyn (Poland). It is based on VTL 1.0 language specification. Its overall architecture is similar to the one described above but much simpler. It uses the same tools for parsing (ANTLR) and code generation (StringTemplate). It performs only a few necessary transformations of its intermediate representation, which is also greatly reduced. One of the most important simplifications is treating scalar values as datasets, which is observable in the example below. The tool is capable of translating simple VTL 1.0 programs. Creation of this software helped to identify potential issues on real translator. It could be further developed to become a product ready for use with any VTL 1.0 source code.

25. Some simple tests of this VTL 1.0 translator were performed. For this sample VTL code:

```
W1:=false
A:=2-1
W2:=true
B:=1+1
C:=9/3
X:=if W1 then A elseif W2 then B else C
```

26. prototype translator produced following T-SQL output:

```
-- VTL: W1 := false
SELECT CAST('false' as bit) AS value1F INTO #w11
-- VTL: const1 := 2
SELECT 2 AS value1F INTO #const100
-- VTL: const2 := 1
SELECT 1 AS value1F INTO #const200
-- VTL: A := 2--1
SELECT
ds_1.value1F - ds_2.value1F AS value1F
INTO #a1
FROM #const100 AS ds_1
CROSS JOIN #const200 AS ds_2
-- VTL: W2 := true
SELECT CAST('true' as bit) AS value1F INTO #w21
-- VTL: const4 := 1
SELECT 1 AS value1F INTO #const400
-- VTL: const5 := 1
SELECT 1 AS value1F INTO #const500
-- VTL: B := 1+1
SELECT
ISNULL(ds_1.value1F, 0) + ISNULL(ds_2.value1F, 0) AS value1F
INTO #b1
FROM #const400 AS ds_1
CROSS JOIN #const500 AS ds_2
-- VTL: const6 := 9
SELECT 9 AS value1F INTO #const600
-- VTL: const7 := 3
SELECT 3 AS value1F INTO #const700
-- VTL: C := 9/3
SELECT
ISNULL(ds_1.value1F, 0) / ISNULL(ds_2.value1F, 0) AS value1F
INTO #c1
FROM #const600 AS ds_1
CROSS JOIN #const700 AS ds_2
-- VTL: X := if W1 then A elseif W2 then B else C
SELECT
CASE
WHEN ds_cond_1.value1F = 1 THEN ds_1.value1F
WHEN ds_cond_2.value1F = 1 THEN ds_2.value1F
ELSE ds_3.value1F
END AS value1F
INTO #x1
FROM #w11 AS ds_cond_1
CROSS JOIN #a1 AS ds_1
CROSS JOIN #w21 AS ds_cond_2
CROSS JOIN #b1 AS ds_2
CROSS JOIN #c1 AS ds_3
```

27. This T-SQL code can be successfully executed on SQL Server instance and gives a valid result in temporary table #x1.

28. Any work on VTL 1.0 translator is suspended for now, but a new project for VTL 1.1 was launched. Currently the efforts concentrate on developing overall software architecture independent of the language specification's details. Further progress is blocked by the lack of a VTL 1.1 specification release.

29. The example translation shows that resulting SQL code may be hardly human-readable even in simple cases.

V. Conclusions

30. Regional Statistical Office in Olsztyn (Poland) approaches to develop source-to-source compiler which allows the translation of VTL into the various data processing languages. Creating such translator is not technically simple task. Architecture of the solution must be independent of the specifics of the target language. A major issue is also the optimization of generated code.

31. At the time of writing this paper the VTL 1.1 specification was not yet released. The design of VtlProcessingLib may change when final version of the standard will become available.

References

Gelsema, T. (2015). *A study of VTL.*

ISO/IEC 14977. (1996). *Extended BNF*.

Marlow, S., & Peyton-Jones, S. (n.d.). *The Glasgow Haskell Compiler*. Retrieved from The Architecture of Open Source Applications: http://aosabook.org/en/ghc.html

Raymond, E. S. (1996). *The New Hacker's Dictionary - 3rd Edition.* London: MIT Press.

SDMX Technical Working Group, VTL Task Force. (2015, February). *Validation & Transformation Language Part 1 - General Description Version 1.0.* Retrieved from The official site for the SDMX community: http://sdmx.org/wp-content/uploads/VTL1_2015_part1_general_final.pdf

SDMX Technical Working Group, VTL Task Force. (2016, October). *VTL – version 1.1 (Validation & Transformation Language) Part 1 - General Description (DRAFT FOR PUBLIC REVIEW).* Retrieved from The official site for the SDMX community: https://sdmx.org/wp-content/uploads/VTL-1-1-review-User-Manual-20161017-final.pdf

The Free On-line Dictionary of Computing. (2017, February). *Inline expansion.* Retrieved from http://encyclopedia2.thefreedictionary.com/Inline+expansion

The GrGen.NET User Manual. (2016, July). Retrieved from IPD Goos: http://www.info.uni-karlsruhe.de/software/grgen/GrGenNET-Manual.pdf