VDM Mapping Manifests
Structure and Guidelines

Michael Kamfonas

Overview

The VDM© Framework is an open toolkit[1] that facilitates the development of database and shell scripts, but also program code, based on well-defined patterns. Templates are used to define the patterns, and manifests to provide the replacement values to convert these templates to actual scripts. The specific manifests discussed here capture mappings from Cobol copybook names and positions to table columns.

The purpose of these manifests is two-fold:

A. As human-readable documents, they provide detail mapping and transformation specifications.
B. As machine-readable documents, they assist in the development of correct and validated parsers that can process stage files and populate database tables.

The focus of this document is to describe the composition of these manifests and to provide instructions for constructing them.

The Manifest Structure

The first few rows of the manifest look like this:

```
BEGIN_DIRECTIVES
DISCRIMINANT_EXTERNAL_VARIABLE          VDM_PARM_DISCRIMINANT
XML_CONTENT XERROR XDATA
LOAD_LOOKUPFILE
XML_ROOT_TAG XDATA
END_DIRECTIVES
```

[0-9]	CBFMT1 MRCH_HKEY	NETWORK_ID -	C C 1 1 20 20	
[0-9]	CBFMT1 EFFC_FROM_DATE	- -	- - - - - -	DATA_DATE ""
[0-9]	CBFMT1 AUTH_RSPN_IDNT	- -	- - - - - -	
[0-9]	CBFMT1 -	INVOICE_NO invoicenr	C 10 2 21 30 10	
[0-9]	CBFMT1 TRAN_TYPE_KEY	TXN_TYPE -	C 20 3 31 50 20	code_lookup["TRAN_TYPE::" TXN_TYPE]
[0-9]	CBFMT1 TRAN_MODE_KEY	TXN_MODE -	C 2 4 51 52 2	code_lookup["TRAN_MODE::" TXN_MODE]
[0-9]	CBFMT1 -	CARD_TYPE -	C 1 5 53 53 1	
[0-9]	CBFMT1 TRAN_DATE	TXN_DATE -	C 8 6 54 61 8	substr(TXN_DATE,1,4) "-" substr(TXN_DATE,5,2) "-" ...
...				

```
#
```

The first few lines may contain directives. A typical directive is to indicate which positions in the incoming file determine the record type of multi-record files. If the file has only one record format, we use the key-word DISCRIMINANT_CONSTANT to indicate this, and also provide a name for the format, in this case DARS.

The rest of the lines can be divided into four sections:

[1] The methodology and core framework is Intellectual property of InfoKarta Inc.

1. The first token on the left is the “discriminant” that determines which record format the line describes. There are three methods supported:
    a. Multi-format files where the source file contains many record types and a single parser is generated to handle all formats and load them into one target. This option is useful for XML and mixed targets with flexible formats are the norm. It may also be used for relational structures where different record formats have different mappings for the same columns or where some of the columns are populated for every format. The discriminant is a field on the incoming records, that determine which one of many formats the record being read represents. That value is matched to the first column of the manifest to determine which rules (lines of the manifest) apply for the given format.
    b. Single-format files with one target are the simplest for relational targets. Since the source file has only one format, there is no need for a discriminant. Since the discriminant value is used in various places in the generated artifacts, we like to use a constant value for the discriminant in those cases, usually the copybook name, or another literal associated with this type of source file.
    c. This is a generalization of the single format – single target manifest, where a single manifest describes many format variations associated with one or more similar source files, and intended to be loaded to one target or a family of very similar targets. This option generates one parser for each format. It is very convenient to use this form for parsers that handle slight variations of source data, as it happens when loading historical data where small enhancements of copybooks cause new versions over time. Under this option, multiple parsers, each associated with a different discriminant value (typically the copybook name) are generated from a single manifest. The copybook name is matched to the discriminant to determine which subset of the manifest applies for each parser.
2. The first box overlay in the picture above blocks four columns. The dash is a placeholder that denotes “no value”. Here is what these four columns represent:
    a. A Format/Record Type designation. Use the name of the format the field belongs to here. The first value after a change of discriminant is currently used in XML parsers as the TYPE of the top element in XML. It is the first value of this column or every change of discriminant value that is maintained. A dash “–“ can be used in all but the first line for each discriminant value. We consider best practice having the same value for each record type.
    b. The column name of the target table goes in the second column of this box. If there is no column for this field then use a dash “-“
    c. The next column contains the field name as defined in the COBOL copybook after the dashes in the name are converted to underscores. The parser uses these names to create internal variables. Nothing says that the names have to be exactly the same as the copybook, but since we want to make this a readable document, it is good to use names that a human can relate to the source definition. Flexibility exists, however, to remove redundant or unnecessary prefixes and keep the names relatively concise. In

cases where the original value and an “edited” value needs to be retained, a suffix can be added to the COBOL name to differentiate the variables.

d. The last column of this box is only valid for XML element generation, and that is the place where the xml-tag is defined. If there is a value in this column, it implies that we want to generate an XML element for this field. To generate an element <invoicenr> that contained the INVOICE_ID value, we would simply use the value “invoicenr” as shown in the fifth position for that row. It is OK to include both an SQL columns and an XML element if we want to populate the value to both a regular column and an XML one.

3. The next group of five columns (column 6 through 11) help identify the field type and positional information on the record buffer:
   a. Column 6 is informational and identifies whether the COBOL type is Character (C) or Numeric (N) – NS stands for zoned numeric. There is one exception to this:

      This column is used to identify the beginning and ending of groups of fields, as records in COBOL or nested elements in XML. When the work “GROUP” appears in this column, an XML group element can be generated if a tag exists on column 6.

   b. Column 7 is also informational and in our case it contains the size of the COBOL type. It may also be used for the SQL type or always left as a dash.
   c. Column 8 is the sequence number of the field in the COBOL record buffer and is also informational. This means that changing that value does not change the parser generated from this manifest. If not interested in tracking the order of the columns you can use a dash as a default.
   d. Columns 9 and 10 contain the beginning and ending position that this field is located in the record buffer. These are critical fields and have to be defined correctly. It is OK to use dashes if the line represents an action that does not require reading the record buffer, for example in order to apply a transformation on a previously read field. Currently the implementation is not aware of multi-byte characters, so characters and bytes are equivalent. In future implementations we plan to support UTF-8 second level, and enable measuring positional information by characters or bytes.
   e. Column 11 is the record length – currently the parser derives the length from the beginning and ending location, however this may change. Make sure that the correct length is in this column. A test script should be able to validate that the length matches the difference between the beginning and ending position.
4. The last box is not columns but any arbitrary string that can be as long as 500 characters. It contains an awk expression that transforms the field or checks its validity. For more complex transformations a function can be defined and the call placed here. Typical transformations we incorporate here are:
   a. Concatenations to form dates/times etc.

b. Conditional evaluations – e.g. we can test a field name and if it matches a certain pattern we can chose to replace the value with a default otherwise return the value: (FIELDNAME ~ /0000/)? “N/A” : FIELDNAME
c. Lookups of codes etc. as:
   code_lookup(“CODE_CTGR::” , MATCH_CD, xTag, xValue, default)
   more on this later
d. Environment or global variables that are passed into the script at run time. Some of these are:
   i. DATA_DATE
   ii. RUN_GROUP
   iii. SOURCE_SYSTEM_CODE
   iv. SORC_FILE_ID
e. Special functions such as:
   i. fix_dec(FIELDNAME,PRECISION, <excertion-handling-parameters>)
   ii. user-specified functions (e.g. algorithm for masking private data or checksum digit calculation for validating account numbers)
f. Simple arithmetic functions, typically used to adjust the decimal position of numbers; for example to convert a number AMOUNT to a decimal we divide by 100: AMOUNT/100

Well-Formed Manifests

The first thing to remember is that the syntax of these manifests is, at least for now, not machine-checked. This means that an ill-formed manifest may be used undetected and code can be generated from it. This makes it critical to keep the manifests simple, neat and readable. Mistakes will be detected down the line, so it is extremely unlikely that wrong code will be moved into production, but it is best to have as few of these errors as possible, and identify them as early as possible. This is mostly a productivity concern, not a correctness one.

Here is a process and a few rules we use to make a well-formed manifest:

1. Begin with the spreadsheet used for the copybook that defines the fields and positional information. Copy the spreadsheet into a Unix Text File
2. Format the text file with tabs and dashes to take the form described above. Use one line for each copybook line. The positional information is very critical and must be correct.
3. Eliminate GROUP and REDEFINES lines that are not useful. Remember we go by the positional information on the buffer
4. By default place the Table Column Name in the appropriate column in front of the field that it is mapped to. Add any simple transformation expression needed, such as a lookup or a test. In most cases this is enough to map the field to the column