Kai Wanke
Sofia Vale

OCS Life Sciences

Work Less – Do More

An Automated Approach to Define-XML Validation

As Define-XML contains many repetitive items, manual validation is not only a tedious and time-consuming task, but also prone to errors. However, this repetitive structure allows for many of its contents to be validated automatically.

It is our recommendation to aim at automating the validation on top of the development for a number of reasons, including: 1) Define-XML should ideally be created before the SDTM or ADaM data is available; 2) manual enrichments may have been made which (inadvertently or not) affect the metadata; and 3) the data may be updated but a new Define-XML cannot be automatically generated as it would undo manual enrichments.

This whitepaper will describe how a significant part of the validation of Define-XML can be automated using SAS scripts, independent of the software or method used to generate the Define-XML, to reduce the workload of the programmer while guaranteeing a high degree of accuracy.

INTRODUCTION

Define-XML is a form of metadata that describes the content of tabular dataset structures (typically SDTM or ADaM when used according to CDISC standards). In short, it provides the reviewer of a clinical study with valuable information regarding the number and content of datasets submitted, the characteristics and origin of all variables used in these datasets (up to the value-level for selected variables) and dictionaries and codelists used in the study. Its purpose is to help the reviewer track the data used in the generation of the datasets, understand the information they contain and, in this way, minimize the time they need to familiarize themselves with the content of a study.

The Define-XML is part of the submission package of clinical studies that is required by certain regulatory bodies such as the United States Food and Drug Administration (FDA) and the Japanese Pharmaceuticals and Medical Devices Agency (PMDA). Because of this, it is essential that the metadata presented in it is complete, accurate and validated.

CHALLENGES OF DEFINE-XML

Several characteristics of clinical studies can make the creation of Define-XMLs challenging: clinical studies usually contain a large number of datasets and variables, which need to be described to a high level of detail, covering associated type, length, format (i.e. ISO 8601 for date/time variables), codelist, origin and other aspects; also, some parameters, like origin, may be difficult to define in a consistent way when done by hand. this makes the description of a single variable a very time-consuming task. Finally, everyday work will probably require more than one programmer to work on the creation and / or validation of a Define-XML, making it hard to maintain consistency among the involved personnel. These reasons make the work often tedious and time-consuming, creating the need to streamline and optimize this process.

PROPOSED SOLUTION: AUTOMATE DEFINE-XML

Fortunately, the repetitive and predictable structure of Define-XML makes it an ideal target for automatization. Indeed, some tools like Pinnacle21® (P21) are available to support the generation and validation of Define-XML files. The free version, P21 Community, uses SDTM or ADaM datasets to create an excel template for the Define-XML which can then be modified by the user and uploaded back into the software to create an xml file based on specifications in the template. This tool already greatly improves the work with Define-XML, but it is unable to create a complete Define-XML template, that needs to be finished by the programmer. The filled-in template is then validated before the final Define-XML is created.

REASONS TO AUTOMATE VALIDATION

We chose to highlight the automatic validation of Define-XML for several reasons: First, a Define-XML may be created very early during the study or the process of data conversion and experience shows that clinical data and/or its presentation in SDTM / ADaM datasets can change even after the Define-XML has been prepared. Manual changes are here often more efficient than to recreate the entire Define-XML.

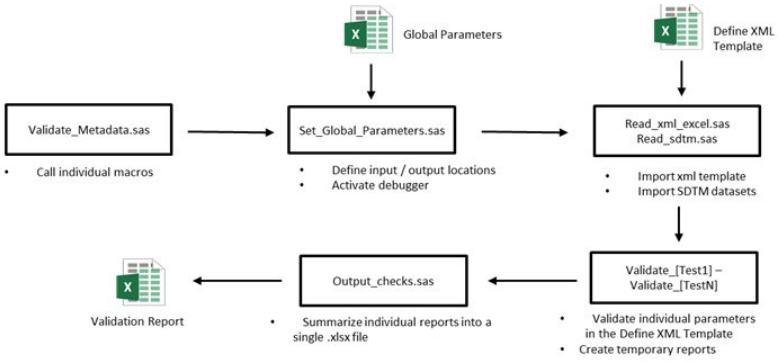
Moreover, there are cases where the data present in the study datasets is inconsistent, causing unexpected results during the automatic generation of the Define-XML. The programmer may overlook these situations, leading to errors in the metadata. Lastly, certain parameters, such as methods or comments, may be study-specific and therefore not predictable by an automated Define-XML creation program, so manual input cannot be fully avoided.

VALIDATION MACROS

The Define-XML validation macros presented in part in this paper are designed to address these cases and to guide the validation of the Define-XML. The macros identify potential inconsistencies in the metadata and summarize them in an excel file that is easily understandable by the validator. The validation macros can be used during any stage of the data conversion and make it possible to specifically change small parts of the Define-XML without recreating it as a whole. The application of automated Define-XML creation and validation programs is an ideal way to reduce the amount of time and effort spent in the creation of clinical metadata and at the same time guarantee the highest degree of completion and consistency.

GENERAL WORKFLOW

The macro to validate the Define-XML is split into a group of sub-macros that work in a modular way.



There is a set of macros dedicated to define the input parameters (i.e. name and location of the Define-XML Template, location of a .xfdf file containing CRF annotation and page numbers and name and location of the to-be-produced output) and to import the necessary files into SAS-datasets. The macro **Set_Global_Parameters** uses a user-generated excel file that provides all the necessary information. It also controls the activity of an optional debugger which determines whether temporary datasets are being deleted or kept for further inspection by the user. The parameters defined by this macro are then used as input parameters for the **Read_xml_excel** and **Read_sdtm** macros which import the specified input files into SAS®. These SAS-datasets are then further used by a set of different macros that validate different aspects of the Define-XML template by comparing it to the actual data in the SDTM datasets. Each macro creates a temporary report dataset which will later be used by the **Output_check** macro that combines them into one final report that is then exported as an excel file.

The test macros act in a modular way and are independent from each other, meaning that they can be easily modified or that new checks can be introduced by the user if a specific trail requires this. This approach makes the validation macro highly adaptive and easy to maintain when new versions of SDTM or Define-XML are released.

The 'heart' of the Define-XML macro is represented by a set of macros that compare the characteristics of datasets, variables, values and codelists described in the Define-XML template with the actual data presented in the SDTM datasets. The macros check if the metadata in the Define-XML Template is complete and consistent with the data in the original datasets.

Possible findings are then linked back to their location in the Define-XML template and summarized in a temporary SAS-dataset. This section highlights a number of key-checks to demonstrate the use of the individual macros in the context of the complete validation macro.

EXAMPLE: CHECK VARIABLE TYPE

The macro **validate_vartypelen** checks the type and length of a variable presented in the metadata with its counterpart in the SDTM dataset. This section focuses on its method to validate the type of a variable. In our studies, we use the four main types as presented in Define-XML: 'text', 'integer', 'float' 'and datetime types.

First, the macro creates a SAS-dataset containing every variable used in a study (i.e. all SDTMs domain are being examined at once). The macro makes use of PROC SQL dictionary tables to identify the variables needed, divides them into character and numeric variables and finally combines them all into one dataset.

```
CODE:

proc sql noprint;
  create table work.temp_sdtm01 as
  select memname as dataset
  from dictionary.tables
  where lowcase(libname) = "sdtm";
quit;

/* Put all SDTM datasets with numeric variables in a
   macro variable to be used in the next data step.*/

proc sql noprint;
  select distinct quote(trim(memname))
  into :numvar_datasets separated by ","
  from dictionary.columns
  where lowcase(libname) = "sdtm"
    and type = "num";
quit;

/* Get all values for each variable in each dataset
   (character or numeric) and then combine all datasets
   created in one. Numeric values will be transformed
   into characters to allow the combination of character
   and numeric values in one common variable. */

data _null_;
  set work.temp_sdtm01;

  call execute(cat(
   "data work.temp_char_", strip(dataset), ";
      set sdtm.", strip(dataset),";
      array _char{*} _character_; length _dataset _variable _value _var_type $ 200;
      retain _var_type 'char';
      _dataset = '", strip(dataset), "'; ", "do i=1 to dim(_char);
      _variable=vname(_char{i});
      _value = _char{i}; output; end; ",
     "keep _dataset _variable _value _var_type;
    run;",

   "proc sort data = work.temp_char_", strip(dataset), " nodupkey;
      by _dataset _variable _value;
    run;"));

  if strip(dataset) in (&numvar_datasets) then
        call execute(cat(
     "data work.temp_num_", strip(dataset), ";
        set sdtm.", strip(dataset),";
        array _num{*} _numeric_;
        length _dataset _variable _value _var_type $ 200;
        retain _var_type 'num';
        _dataset = '", strip(dataset), "'; ",

         "do i=1 to dim(_num); variable=vname(_num{i});
                  _value = strip(put(_num{i},best.));
           output;
        end; ",

         "keep _dataset _variable _var_type _value;
     run;",
    "proc sort data = work.temp_num_", strip(dataset), " nodupkey;
       by _dataset _variable _value;
     run;"));
run;

data work.temp_all01;
  set work.temp_char_:
      work.temp_num_:;
run;
```

The following data steps assign the SDTM type of each variable. Some derivations presented here are based on our study-defined conventions and may be subject to change in other circumstances.

```
CODE:

data work.temp_type01;
  set work.temp_all01;

  if missing(_value) then _type = 0;
  else if _var_type = "char" then _type = 4;
  else if find(_value,".") and length(strip(_value)) > 1 then _type = 2;
  else _type = 1;

  if _type = 4 then do;
    if substr(variable,length(variable)-2,3)= 'DTC' then do;
      temp_value = translate(_value,"xxxxxxxxxx","0123456789");
      if find(temp_value,"xxxx-xx-xxTxx:xx") then _type = 3.9;
      else if find(temp_value,"xxTxx") then _type = 3.8;
      else if find(temp_value,"xxxx-xx-xx") then _type = 3.7;
      else if find(temp_value,"xxxx") or find(temp_value,"--xx")
        then _type = 3.6;
      else if find(temp_value,"Txx:xx") then _type = 3.5;
      else if find(temp_value,"Txx") then _type = 3.4;
    end;
    else if length(_value) > 2 and findc(strip(_value),"PTYMWDHS","div") = 0
      and find(_value,"P") = 1 and anyalpha(substr(_value,length(_value)))
      then _type = 3.3;
  end;
  drop temp_value;
run;

proc sort data = work.temp_type01;
  by _dataset _variable _type;
run;

data work.temp_type02;
  set work.temp_type01;
  by _dataset _variable _type;
  if last._variable then output;
run;

data work.temp_type03;
  set work.temp_type02;
  length type_c $ 200;
  select(_type);
    when(0)   type_c = "missing";
    when(1)   type_c = "integer";
    when(2)   type_c = "float";
    when(3.3) type_c = "durationDatetime";
    when(3.4) type_c = "partialTime";
    when(3.5) type_c = "Time";
    when(3.6) type_c = "partialDate";
    when(3.7) type_c = "date";
    when(3.8) type_c = "partialDatetime";
    when(3.9) type_c = "datetime";
    when(4)   type_c = "text";

    otherwise put "WAR" "NING: Unknown type found.";
  end;
run;
```

Next, the macro creates a new dataset containing the variable and type information generated here with the metadata presented in the Define-XML Template. It then compares the two to find inconsistencies, which are then used to generate warnings in the report dataset.

```
CODE:

proc sql;
  create table work.temp_comp_vartype (drop = order) as
  select a.order
        ,a.dataset
        ,a.variable
        ,a.data_type as xml_type
        ,b.type_c as act_type
  from work.xml_variables(where = (^missing(variable))) as a
  left join work.temp_type03 as b
    on a.dataset = b._dataset
   and a.variable = b._variable
  order by dataset, order;
run;

data work.check_vartype;
  set work.temp_comp_vartype;
  length type id result compare $ 200;

  type = "Variable type";
  id = cat("VAR_TYPE_",strip(put(_n_,z3.)));

  if act_type = "missing" then
    compare = cat("No values found for variable ", strip(variable),
                  " on dataset ", strip(dataset),
                  ". Variable type has to be checked manually.");
  else if xml_type ^= act_type then
    compare = cat("The variable ", strip(variable), " on dataset ",
                  strip(dataset), " is defined as ", strip(xml_type),
                  " when the expected type is ", strip(act_type), ".");
  if missing(compare) then result = "Right";
  else if find(compare, "manually") then
       result = "Check (" || strip(id) || ")";
  else result = "Wrong (" || strip(id) || ")";
run;
```

EXAMPLE: CHECK TYPES OF VALUE-LEVEL METADATA

Our Define-XML validation macro also considers value-level metadata. Generally, we create this type of metadata for variables with more than one origin (e.g. LBORRES which usually uses CRFs and electronic Data Transfers - eDTs), different codelists (e.g. TSPARM) or different types of data (e.g. –TRESC variables that contain text or numeric data). The creation and validation of value-level metadata is often time consuming and challenging, since every possible value has to be taken into account. Thus, the Define-XML validation macro combines the information in the Where-Clause tab of the excel template with the value-level metadata to determine the different groups of values that should be present in the variables. It then uses this information to break down the data into different SAS-datasets that are then combined and compared to the value-level metadata in the Define-XML template. These checks are done using the same logic as the ones used for variable-level, so only the code for the initial breakdown will be presented here.

CODE:

```
/*Get the where-clauses to subset the datasets
  with value-level metadata.*/
data work.temp_where01;
  set work.xml_whereclauses;
  length clause1 $ 200;
  if comparator in ("EQ", "NE") then
    clause1 = strip(variable) || " " || strip(comparator) || " '" ||
           strip(value) || "'";
  else if comparator in ("IN", "NOTIN") then do;
    value = tranwrd(value,", ", "', '");
    clause1 = strip(variable) || " " || strip(comparator) || " ('" ||
           strip(value) || "')";
  end;
run;

proc sort data = work.temp_where01;
  by id variable;
run;

data work.temp_where02;
  set work.temp_where01;
  by id variable;
  length clause $ 200;
  retain clause;
  if first.id then clause = clause1;
  else clause = strip(clause) || " AND " || strip(clause1);
  if last.id then output;
run;

proc sql;
  create table work.temp_val01 as
  select a.order
        ,a.dataset
        ,a.variable
        ,a.data_type
        ,a.length
        ,a.where_clause
        ,b.clause
  from work.xml_valuelevel as a
  left join work.temp_where02 as b
  on a.where_clause = b.id;
quit;

/*Obtain a dataset with all values with value-level metadata.*/
data _null_;
  set work.temp_val01;
  if missing(clause) then
    put "WAR" "NING: Missing where-clause for variable " variable
        " on dataset " dataset ", order number " order ".";
  else do;
    call execute(cat(
      "data work.temp_vlm_", strip(dataset), strip(order), ";
       set sdtm.", strip(dataset), ";
       length dataset variable where_clause value $ 200; ",
      "dataset = '", strip(dataset), "';
       variable = '", strip(variable), "';
       where_clause = '", strip(where_clause), "';
       value = ", strip(variable), ";
       where ", strip(clause), ";
       keep dataset variable where_clause value; run;"));

    call execute(cat(
      "proc sql;
         create table work.temp_vlm_d_",strip(dataset),strip(order)," as
         select distinct *
         from work.temp_vlm_",strip(dataset),strip(order),";
       quit;"));
  end;
run;
```

GENERATE OUTPUT

Each different macro creates a SAS-dataset with its findings that can be related to one of the levels of the Define-XML. The macro **output_check** combines the datasets containing information regarding one of its levels in a single dataset. See below an example for the dataset describing variable-level information. Note that the dataset "check_varlen" has not been created in the previous examples and is displayed here only to demonstrate how different outputs are being combined during the generation of the output file.

```
CODE:

/*Combine all variable checks*/
proc sql;
  create table work.table_variables as
  select c.order as order
        ,a.dataset
        ,a.variable
        ,a.result as type
        ,b.result as length
  from work.check_vartype as a
  left join work.check_varlen as b
    on a._dataset = b._dataset
    and a._variable = b._variable
  left join work.xml_variables as c
    on a.dataset = c.dataset
    and a.variable = c.variable
  order by dataset, c.order;
quit;
```

Eventually, the same macro exports the datasets in a single excel file containing a tab per findings related to each level of the Define-XML metadata. The macro makes use of the ODS System to create the output. The macro also controls the colour of cells in the final excel sheet in order to highlight its findings for the user. Below you can see how the variable-level information about type is exported into the output excel sheet.

```
CODE:

%let wrong_color = lightpink;
%let check_color = gold;

proc report data = work.table_variables;
  columns dataset variable type;
  define dataset / display "Dataset";
  define variable / display "Variable";
  define type / display "Type";

  compute type;
    if find(type,"Check") then do;
      call define(_col_, "style", "style=[background=&check_color]");
    end;
    else if find(type,"Wrong") then do;
      call define(_col_, "style", "style=[background=&wrong_color]");
    end;
  endcomp
run;
```

CONCLUSION

This paper presents one approach to automate the validation of Define-XML metadata. The validation is based on an excel-template, which is then used to create a Define-XML using P21. The validation macros compare the metadata described in the template with the SDTM datasets of its associated study to find missing or inconsistent information. The identified issues are then summarized in an excel file that can easily be used by the programmer to find and solve potential issues in the template.

The code presented in this paper demonstrates the general workflow of our strategy to automate Define-XML validation and provide a starting point for programmers to add their own checks, which is made easy by the modular framework in which our macros operate.

The here presented macros represent only a small fraction of the checks used in the full program, which is able to examine dataset -, variable - and value – level information, as well as the codelists and dictionaries used in the study. Specifically, it evaluates information regarding variable / value type, length, format, codelist, origin and more.

Taken together, the automated validation of Define-XML has proven to be an efficient way to optimize the generation of metadata in clinical studies. Implementing this strategy has great potential to greatly increase speed and accuracy of Define-XML production by simultaneously reducing the workload of the programmers involved.

ACKNOWLEDGMENTS

The authors would like to thank our colleagues at OCS Life Sciences, especially Jules van der Zalm and Lieke Gijsbers, for their valuable input and support during the preparation of this paper.

RECOMMENDED READING

Generating Define.xml from Pinnacle 21 Community - Pinky Anandani Dutta, Inclin, Inc

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Kai Wanke

OCS Life Sciences

Address: Ruwekampweg 2G

City / Postcode: 's-Hertogenbosch / 5222 AT

Work Phone: +31 (0)73 523 6000

Email: sasquestions@ocs-consulting.com

Web: https://www.linkedin.com/in/kaiwanke/