Paper 012-2013

A Metadata Driven Programming Technique using SAS®

Xiyun Cheryl Wang, Statistics Canada, Ottawa, Ontario, Canada

ABSTRACT

In a typical statistical SAS® system, validations on user inputs and imputation on missing values using default values are essential to ensure core system processes finish successfully without errors. This paper introduces a metadata driven programming technique using SAS® for validations and imputations in SAS macros. It first explains how to define validation rules as metadata for various types of inputs and how to store default values for missing values as metadata using SAS syntax; then it illustrates how to utilize the defined metadata to automate validation and imputation processes against user inputs. This metadata driven programming technique using SAS® eases our system development by promoting efficiency, reusability, easy maintainability, and good coding consistency.

1. INTRODUCTION

Metadata driven programming technique contains two key concepts. First, what are the metadata and how to define them. Second, how to utilize the defined metadata to drive system implementation. Next, I will explain how metadata driven programming technique using SAS is brought into our SAS systems development.

G-Sam (Generalized Sampling System), a new statistical SAS system I am responsible for, has a very unique situation; some SAS macros receives more than 20 different user inputs which need to be validated and imputed before proceeds to the core system processes. User inputs can be library names, dataset names, string, or floating point numbers. During prototyping, I quickly find out that by using IF THEN ELSE IF THEN statements, the block of SAS code for validation and imputation becomes excessive. Furthermore changes on the excessive blocks of IF THEN ELSE IF THEN statements become fragile since often unenclosed if statements are found; modifying existing blocks of IF THEN ELSE IF THEN statements is always at risk to introduce new bugs when validation and imputation rules need to be changed. It gets hard to accurately spot the full business rules for validation and imputation within the excessive IF THEN ELSE IF THEN statements blocks. My investigation and research to find another alternative to replace IF THEN ELSE IF THEN blocks ends up a new solution: use metadata driven programming technique for validations and imputations. Validation and imputation rules can be defined in centralized metadata file and then the defined metadata drives system implementation of validation and imputation processes. The already released G-Sam system proves this technique efficient with high reusability, easy maintainability, and good coding consistency.

2. THREE LEVELS OF VALIDATIONS AND IMPUTATIONS

In G-Sam, the end products are a collection of pre-compiled SAS macros stored in SAS macro catalogs. Typically when a SAS macro is to be built into G-Sam, I insist three levels of validations and imputations done in the SAS macro, described below:

- **At macro parameters level:** First, macro parameters are validated against validation rules. If some macro parameters are not valid, then error messages are printed into SAS log and the SAS macro exits with errors; otherwise, missing optional macro parameters are set to default values. Then the macro can proceed to next level of validations and imputations.
- **At input datasets file structure level:** At this level, input datasets must have correct file structures. If not, then error messages are printed into SAS log and the SAS macro terminates with errors; otherwise, add optional variables into input datasets with correct data types if they are not in input datasets. Then the macro can proceed to next level of validations and imputations.
- **At data level:** At this level, the data inside input datasets are validated to be correct. If not, then error messages are printed into SAS log and the SAS macro stops with errors; otherwise, impute default values for missing values in input datasets. Then the macro can proceed to further system processing.

When user inputs passes these three levels of validations and imputations, the quality of user inputs are ensured and thus further core system processing will encounter less problems.

The three levels of validations and imputations are required for many SAS macros. As mentioned in section 1, by using IF THEN ELSE IF THEN statements, not only the blocks of code are excessive and hard to be maintained, but also system implementation of validation and imputation processes becomes repetitive, robotic and time-consuming. On the contrary, meta-data driven programming technique using SAS becomes a nature fit. First SAS allows code generate through SAS macro facilities. This allows validation and imputation rules in SAS syntax format to be stored in SAS macro variables and then be plugged into SAS data steps or SAS macros to generate code for validations

and imputations. Second, by utilizing SAS macro facility and SAS data steps, it is easy to build generic SAS macros for three levels of validations and imputations; then these generic SAS macros can be invoked in any SAS macro to conduct its three levels of validations and imputations. So meta-data driven programming using SAS becomes a nature fit for validations and imputations in statistical SAS systems. It saves systems development efforts and promotes reusability, easy maintainability, and coding consistency.

3. METADATA DRIVEN PROGRAMMING FOR THREE LEVELS OF VALIDATIONS AND IMPUTATIONS

Here, examples are used to explain how to define the three levels of validations and imputations rules as metadata and furthermore how the defined metadata drives the implementation of validations and imputations processes.

Assume the following SAS macro `%process1` is defined according to specifications. It receives a collection of user inputs, runs its processing steps, and finally saves system results to output files.

```
%macro process1(inDataLib=WORK,
                inGlobalParametersFile=,
                inFrameFile=,
                inAllocationFile=,
                inUpdateFlag=,
                inRotationRate=,
                outDatalib=WORK,
                outSampleFile=);
   /*processing-steps are here*/
%mend;
```

3.1 AT MACRO PARAMETERS LEVEL

1) Define metadata on validation and imputation rules for macro parameters of SAS macro `%process1`

Here are validation and imputation rules for macro parameters of SAS macro `%process1`:

- `inDatalib` specifies the SAS library where all input files are stored and has to be valid; if `missing`, `inDatalib` is set to SAS "`WORK`" library;
- `outDatalib` specifies the SAS library where output files are stored and has to be valid; if `missing`, `outDatalib` is set to SAS "`WORK`" library;
- `inGlobalParametersFile`, `inFrameFile`, `inAllocationFile` specify input datasets names; these datasets must exist in the input data library;
- `inUpdateFlag` specifies a flag; it has to be "`YES`" or "`NO`" or `missing`. If `missing`, it is set to "`NO`";
- `inRotationRate` specifies a floating number; it has to be between `0.0` and `1.0`;
- `outSampleFile` specifies an output dataset name;

The above rules can be easily translated into metadata in Table 1 and saved into a Microsoft Excel file. Note some columns are written in SAS syntax format for code generation within validation processes.

object_type	object_name	member_name	member_type	IO_type	invalid_condition	errMsg	default	required
MACRO	PROCESS1	inDataLib	LIBRARY	INPUT	(%upcase(&inDataLib) NE WORK and %upcase(&inDataLib) NE WORK1)	inDatalib <&inDataLib> is invalid and should be either WORK or WORK1 or missing	WORK	NO
MACRO	PROCESS1	inGlobalParametersFile	DATASET	INPUT				YES
MACRO	PROCESS1	inFrameFile	DATASET	INPUT				YES
MACRO	PROCESS1	inAllocationFile	DATASET	INPUT				YES
MACRO	PROCESS1	inUpdateFlag	CHARACTER	INPUT	NOT(%upcase(&inUpdateFlag) = YES or %upcase(&inUpdateFlag) = NO)	inUpdateFlag=<&inUpdateFlag> is not valid%nrstr(;) it should be either YES or NO or missing	NO	NO
MACRO	PROCESS1	inRotationRate	NUMERIC	INPUT	%sysevalf(&inRotationRate<0.0) or %sysevalf(&inRotationRate > 1.0)	inRotationRate =<&inRotationRate> is not valid%nrstr(;) inRotationRate should be >= 0.0 and <=1.0		YES
MACRO	PROCESS1	outDatalib	LIBRARY	OUTPUT			WORK	NO
MACRO	PROCESS1	outSampleFile	DATASET	OUTPUT				YES

Table 1: Metadata on validation and imputation rules for macro parameters of SAS macro `%process1`

For SAS macro `%process1`, the metadata in Table 1 is interpreted in this way:

- `object_type`: the type is "`MACRO`";
- `object_name`: the object name is "`PROCESS1`";
- `member_type`: the type of information supplied by macro parameters and is used in validation process;
  - If `member_type` is "`LIBRARY`", then the library provided by the corresponding macro parameter must exist and its name must not be longer than 8 characters;
  - If `member_type` is "`DATASET`", then if the corresponding macro parameter specifies "`INPUT`" dataset name,

then the dataset must exist and its name cannot be longer than 32 characters;

- member_name: the list of members are macro parameters of macro %process1;
  - inDataLib and outDataLib: specify input and output library names;
  - inGlobalParametersFile, inFrameFile, inAllocationFile, outSampleFile: specify dataset names ;
  - inUpdateFlag: specifies a character string (CHARACTER type);
  - inRotationRate: specifies a floating point number (NUMERIC type);
- IO_type: specifies whether a macro parameter provides information for input or for output. This metadata is used in validation process. For example, if a macro parameter specifies an input dataset names, then the dataset must exist; but if it specifies an output dataset name, then no need to check whether the dataset exists or not;
- invalid_condition: specifies under what condition, a macro parameter is invalid; this metadata is used to generate code and must be in SAS syntax format; take inDataLib as an example, (%updase(&inDataLib) NE WORK and %updase(&inDataLib) NE WORK1) means if inDataLib is not equal to "WORK" or "WORK1", then it is invalid;
- errMsg: describes what error messages to be printed into SAS log when invalid_condition is true
- default: specifies what is the default value for an optional macro parameter when it is missing
- required: specifies whether a macro parameter is mandatory or optional; this metadata is used for both validations and imputations processes. For example:
  - "inDataLib" is optional. During validation, it can be missing; during imputation, if missing, it is set to WORK;
  - For all mandatory macro parameters, if missing, they are invalid and SAS macro stops;

2) Utilize metadata for validations and imputations on macro parameters of SAS macro %process1

Here are the major steps for validations and imputations processes on macro parameters of %process1.

- Step1: import metadata in the Excel file storing Table 1 into a SAS dataset work.macroParams_metaData

```
proc import datafile="L:\SAS Global Forum\programs\macroParams_metadata.xlsx"
     out=work.macroParams_metaData dbms=xlsx replace;
     sheet=macroParams_metadata;
     getnames=Yes;
run;
```

- Step 2: build generic SAS macros for validations and imputations on macro parameters driven by metadata defined in work.macroParams_metadata.
  First the generic SAS macro %macroParamsValidation(inMacroName=) loops through each macro parameter defined in metadata for &inMacroName and invokes another generic SAS macro %genericMacroParamValidation() to conduct validations and imputations on each macro parameter.

```
%macro macroParamsValidation(inMacroName=);
    %local j consCnt;
    /*save the list of macro parameters from metadata file into macro variables*/
    data _null_;
        retain constrCount 0;
        set work.macroParams_metaData end=_eof;
        where upcase(object_name) =upcase("&inMacroName");
        n=strip(_n_);
        call symputx("paramInvalid_condition"||n,strip(invalid_condition));
        call symputx("paramErrMsg"||n,strip(errMsg));
        call symputx("paramName"||n,strip(member_name));
        call symputx("paramType"||n,strip(member_type));
        call symputx("paramDefault"||n,strip(default));
        call symputx("paramReqFlag"||n,strip(required));
        call symputx("IO_type"||n,strip(IO_type));

        if _eof then
            call symputx("consCnt",n);
    run;
    /* loop through all macro parameters to validate them*/
    %do j=1 %to %cmpres(&consCnt);
        /*reset the missing value for numeric type of parameters*/
        %if %upcase(&&paramType&j) = NUMERIC and %sysfunc(cats(&,&&paramName&j))= %then
            %let &&paramName&j=.;

        %genericMacroParamValidation(inMacroName=%upcase(&inMacroName),
            paramType=%upcase(&&paramType&j),
            paramName=&&paramName&j,
            paramValue=%sysfunc(cats(&,&&paramName&j)),
            paramReqFlag=%upcase(&&paramReqFlag&j),
            paramDefault=%upcase(&&paramDefault&j),
            paramInvalid_condition=&&paramInvalid_condition&j,
            paramErrMsg=&&paramErrMsg&j,
            paramIOType=&&IO_type&j);
    %end;
%mend;
```

Generic SAS macro `%genericMacroParamValidation()` performs the following validations and imputations:

- For mandatory parameters: they must not be missing, library and input datasets provided by them must have rigth name length and exist, and `invalid_condition` is not true; otherwise, macro stops with errors;
- For optional parameters: if missing, impute to default values. If not, library and input datasets provided by them must have proper name length and exist, and `invalid_condition` is not true, otherwise, macro stops with errors;

```
 /*generic macro to validate macro input parameters*/
%macro genericMacroParamValidation(inMacroName=, paramName=, paramType=, paramValue=,
          paramReqFlag=, paramDefault=, paramInvalid_condition=, paramErrMsg=,  paramIOType=);

    %if &paramName = %then %return;
    %if (%sysfunc(countc(%upcase(%cmpres(&paramType)),LIBRARY DATASET CHARACTER NUMERIC, iv)) ne 0) %then %do;
          %put ERROR: For macro <&inMacroName>, param type <&paramType> for param <&paramName> should be: LIBRARY DATASET CHARACTER NUMERIC;
          %goto error;
    %end;

    %if &paramValue= %then %do;
       %if &paramReqFlag = YES %then %do;
          %put ERROR: For macro <&inMacroName>, required macro parameter of <&paramType> type <&paramName> cannot be empty;
          %goto error;
       %end;
       %else %do; /*param is optional, only DATASET type of param will not be set to defaul values*/
          %if &paramType = DATASET %then
             %put WARNING: For macro <&inMacroName>, macro parameter of <&paramType> type <&paramName> is empty;
          %else %do; /*except DATASET type of param, all other params are set to default values*/
             %let &paramName = &paramDefault;
             %put WARNING: For macro <&inMacroName>, macro parameter of <&paramType> type <&paramName> is empty and set to default as <&paramDefault>;
          %end;
       %end;
    %end; /*end %if &paramValue= */
    %else %do; /* &paramValue not empty*/
       %if &paramType = LIBRARY %then %do;
          %if %eval(%length(&paramValue.)>8) %then %do;
             %put ERROR: For macro <&inMacroName>, SAS library <&paramValue> provided by param <&paramName> is longer than 8 characters;
             %goto error;
          %end;
          %if (%sysfunc(libref(&paramValue.))) %then %do;
             %put ERROR: For macro<&inMacroName>, SAS library <&paramValue> provided by param <&paramName> does not exist;
             %goto error;
          %end;
          %genericConstraintsCode();
          %if &errorFlag=YES %then  %goto error;
       %end; /*end %if &paramType = LIBRARY ...*/
       %else %if &paramType = DATASET %then %do;
            %if %eval(%length(&paramValue.)>32) %then %do;
               %put ERROR: For macro <&inMacroName>, SAS dataset name <&paramValue> provided by param <&paramName> is longer than 32 characters;
               %goto error;
            %end;
            /*only verify input dataset*/
            %if &paramIOType = INPUT and (%sysfunc(exist(&paramValue.,data)) = 0) %then %do;
               %put ERROR: For macro <&inMacroName>, SAS Dataset <&paramValue> provided by param <&paramName> does not exist;
               %goto error;
            %end;
            %genericConstraintsCode();
            %if &errorFlag=YES %then %goto error;
        %end;/*end %if &paramType = DATASET ...*/
        %else %if &paramType = CHARACTER OR  &paramType = NUMERIC %then %do
            %genericConstraintsCode();
            %if &errorFlag=YES %then %goto error;
        %end;/*end %if &paramType = CHARACTER ...*/
    %end;

    %return;
%error: %let errorFlag=YES;
%mend genericMacroParamValidation;

%macro genericConstraintsCode();
    %if &paramInvalid_condition NE %then %do;
        %if &paramInvalid_condition %then %do;
            %put ERROR: For macro <&inMacroName>, &paramErrMsg;
            %let errorFlag=YES;
        %end;
    %end;
%mend;
```

- Step 3: Utilize the above generic SAS macros to validate and impute macro parameters of macro `%process1`. SAS macro `%process1` invokes %`macroParamsValidation` to conduct validations and imputations on its macro parameters. Testing case 1 is to test a collection of valid and invalid macro parameter for `%process1`.

```
/*define SAS macro process1*/
%macro process1(inDataLib=, inGlobalParametersFile=, inFrameFile=, inAllocationFile=,
                inUpdateFlag=, inRotationRate=, outDatalib=, outSampleFile= );
     %local macroName; %let macroName=PROCESS1;
     %local errorFlag; %let errorFlag=NO;
     %put; %put NOTES: --> &macroName starts;

     %macroParamsValidation(inMacroName=&macroName);
     %if &errorFlag=YES %then %goto error;

     /*more processing steps here*/
     %put; %put NOTES: --> &macroName finishes successfully;
     %return;
%error: %put ERROR: -->&macroName stops due to list of errors;
%mend process1;

/* testing code block */
%put; %put NOTES: ****** Testing case1: some macro parameters are not valid******;
libname inLib "L:";
data allocationFile;
     ID=1;size=10;
run;
%process1(inDataLib=inLib,
     inGlobalParametersFile=a12345678910111213141516171819202122 2,
     inFrameFile=,
     inAllocationFile=allocationFile,
     inUpdateFlag=,
     inRotationRate=1.5,
     outDatalib=,
     outSampleFile=outSample
     );
```

- Step 4: review SAS log

```
NOTES: ****** Testing case1: some parameters are not valid******

NOTES: --> PROCESS1 starts
ERROR: For macro <PROCESS1>, inDatalib  <inLib> is invalid and  should be either WORK or WORK1
ERROR: For macro <PROCESS1>, SAS dataset name <a12345678910111213141516171819202122 2> provided by
param <inGlobalParametersFile> is longer than 32 characters
ERROR: For macro <PROCESS1>, required macro parameter of <DATASET> type <inFrameFile> cannot be empty
WARNING: For macro <PROCESS1>, macro parameter of <CHARACTER> type <inUpdateFlag> is empty and set to default as <NO>
ERROR: For macro <PROCESS1>, inRotationRate =<1.5> is not valid;  inRotationRate should be >= 0.0 and <=1.0
WARNING: For macro <PROCESS1>, macro parameter of <LIBRARY> type <outDatalib> is empty and set to default as <WORK>
ERROR: -->PROCESS1 stops due to list of errors
```

In testing case1, a group of errors found on macro parameters and macro `%process1` stops due to errors; at same time, optional macro parameters are set to default values and warning message are printed out. Here the validation and imputation are done together for each parameter to avoid repetitive code since the code for validations and imputations would be pretty similar.

3.2 AT FILE STRUCTURE LEVEL

After all macro parameters of SAS macro %`process1` are properly supplied, now I move to file structure level validations and imputations. Here I take input dataset `&inGlobalParametersFile` as an example.

1) Define metadata on file structure of `&inGlobalParametersFile`

Based on specifications, metadata for validations and imputations on input dataset `&inGlobalParametersFile` is defined in Table 2 and saved into a Microsoft Excel file.

macroName	object_type	object_name	member_name	member_type	invalid_condition	errMsg	default	required
PROCESS1	DATASET	&inGlobalParametersFile	selMeth	CHAR	NOT(selMeth in ("SRS", "BERNOULLI") or missing(selMeth))	Selection method should be either MISSING, or "SRS" or "BERNOULLI"		YES
PROCESS1	DATASET	&inGlobalParametersFile	start	NUM	NOT(0<=Start<=1 or missing(start))	Starting point should be either MISSING or between 0 and 1	0	NO
PROCESS1	DATASET	&inGlobalParametersFile	rotationRate	NUM				YES
PROCESS1	DATASET	&inGlobalParametersFile	direction	CHAR	NOT(direction in ("LEFT", "RIGHT") or missing(direction))	Direction should be either MISSING, or "LEFT" or "RIGHT"	RIGHT	NO
PROCESS1	DATASET	&inGlobalParametersFile	seed	NUM			%sysfunc(time())	NO

Table 2: Metadata for validations and imputations on input dataset `&inGlobaParametersFile`

Note, in Table 2, I naturally combined metadata for validations and imputations both on file structures and on data in input dataset `&inGlobalParametersFile` together. Only these columns are needed for file structure validations and imputations: `macroName`, `object_type`, `object_name`, `member_name`, `member_type`, `required`.

For dataset &inGlobalParametersFile, rotationRate and seed variables are optional; other variables are mandatory; and each variable has its own data type indicated in member_type column.

2) Utilize metadata for validations and imputations on file structure of dataset &inGlobaParametersFile

Here are the major steps for validations and imputations on file structure of dataset &inGlobaParametersFile.

- Step 1: import metadata in a Excel file storing table 2 into work.DSN_metadata.

```
proc import datafile="L:\SAS Global Forum\programs\fileStructMetadata.xlsx"
     out=work.DSN_metaData dbms=xlsx replace;
     sheet=fileStructMetadata;
     getnames=Yes;
run;
```

- Step 2: build generic SAS macros %genericFileStructValidation(inDSN=,inMacroName=) to validate and impute file structure of dataset &inDSN according to its defined metadata. First the list of variable names in &inDSN are stored in a macro variable &inDSN._varList; next a data step loops through each variable defined in metadata for &inDSN and uses SAS index() function to check whether a required variable is in &inDSN._varList. If yes, then check the variable type is correct or not. if not in &inDSN._varList, and if the variable is optional, add this variable into &inDSN with proper data type and print out warning messages. (Note: code example continues to next page).

```
/*meta-data driven file structure validations and imputations*/
%macro genericFileStructValidation(inDSN=,inMacroName=);
    %local NumObs1 where_clause &indsn._reqVarList &indsn._reqVarList1;

    /*get required variable names and variable types from metadata*/
    data work.&inDSN._reqStruct;
        set work.DSN_metaData
            (keep=macroName object_type object_name member_name member_type required)
             end=_eof;
        where upcase(strip(macroName)) = upcase(strip("&inMacroName")) and
              upcase(strip(resolve(object_name))) = upcase(strip("&inDSN")) and
              upcase(strip(Object_type)) = "DATASET";

        rename member_name=varname  member_type=vartype;
        if _eof then call symputx("NumObs1",_n_);
    run;

    %if %eval(&NumObs1 =0) %then %return;

    /* Storing variable names from metadata into macro vars*/
    proc sql noprint;
        select quote(upcase(strip(varName))),upcase(strip(varName))
          into :&indsn._reqVarList separated by " ",:&indsn._reqVarList1 separated by " "
          from work.&indsn._reqStruct;

          select upcase(strip(varName)) into :&indsn._reqVarList1 separated by " "
          from work.&indsn._reqStruct where upcase(strip(required))="YES";
    quit;

    /*For data step where clause*/
    %let where_clause = (%sysfunc(cats(&,&indsn._reqVarList)));

    /*get the list of variables presented in input dataset*/
    proc contents data=work.&inDSN.
        out=work.&indsn._struct(keep=NAME  TYPE)  noprint;
    run;

    data work.&indsn._struct(KEEP=varname varType);
        set work.&indsn._struct;
        where upcase(strip(name)) in &where_clause;

        if type=2 then varType="CHAR";
        else varType="NUM";
        rename name=varname;
    run;

    /* check whether all required variables and variables types are supplied*/
    %local &indsn._varTypeList &indsn._varList;
    proc sql  noprint;
        select cats(upcase(strip(varName)),"-",upcase(strip(varType)), upcase(strip(varName))
          into :&indsn._varTypeList separated by " ", :&indsn._varList separated by " "
          from work.&indsn._struct;
    quit;

    /*if no mandatory variables presented in input file*/
    %if %sysfunc(cats(&, &indsn._varList)) = %then %do;
        %put ERROR: In dataset <&inDSN>, mandatory variables <" %unquote(%sysfunc(cats(&,&indsn._reqVarList1))) "> are not presented;
        %goto error;
    %end;
```

```
    /*some mandatory variables are presented in input file*/
    data _null_;
        length varList  varTypeList varAddedList $500;
        retain varList  varTypeList varAddedList "";
        retain errorCount 0;
        set work.&inDSN._reqStruct end=_eof;

        if _n_=1 then do;
                varTypeList=upcase(strip(symget("&indsn._varTypeList")));
                varList=upcase(strip(symget("&indsn._varList")));
        end;

        if upcase(strip(required)) = "YES" then do; /*mandatory variable*/
            if index(varList,upcase(strip(varName)))=0 then do;
                put "ERROR: In <&inDSN> dataset, variable <" varName "> is required and not presented";
                errorCount=errorCount+1;
            end;
            else do; /*mandatory variable is in input file, then check its type*/
                if index(varTypeList, cats(upcase(strip(varName)),"-",upcase(strip(varType))))=0 then do;
                    put "ERROR: In <&inDSN> dataset, variable <"  varName "> should be of data type of <" varType ">";
                    errorCount=errorCount+1;
                end;
            end;
        end;
        else do; /*optional variable*/
            /*prepare statements to add optional variables if they are not presented in input file*/
            if index(varList, upcase(strip(varName)))=0 then do;
                varAddedList =  catx(" ,",varAddedList,varName);
                varAddedList =  catx(" ",varAddedList,varType);
                put "WARNING: In <&inDSN> dataset, variable <" varName "> is not presented and added as <" varType "> type";
            end;
            else do; /*optional variable is in input file, then check its type*/
                if index(varTypeList, cats(upcase(strip(varName)),"-",upcase(strip(varType))))=0 then do;
                    put "ERROR: In <&inDSN> dataset, variable <" varName "> should be of data type of <" varType ">";
                    errorCount=errorCount+1;
                end;
            end;
        end;
        if _eof then do;
                call symputx("varAddedList",varAddedList);
                call symputx("errorCount",errorCount);
        end;
    run;

    /*add optional variables*/
    %if &varAddedList NE %then %do;
        proc sql noprint;
            alter table work.&indsn add &varAddedList;
        quit;
    %end;
    %if %eval(&errorCount>0) %then %goto error;
    %return;
    %error: %let errorFlag=YES;
%mend;
```

- Step 3: Utilize the above generic SAS macros to validate and impute file structure of `&inGlobalParametersFile`. Macro `%process1` simply invokes `%genericFileStructValidation`. Testing case1 is to test valid and invalid variables and their data types in input dataset `globalParameters`.

```
%macro process1(inDataLib=, inGlobalParametersFile=, inFrameFile=, inAllocationFile=,
             inUpdateFlag=, inRotationRate=, outDatalib=, outSampleFile= );
    %local macroName; %let macroName=PROCESS1;
    %local errorFlag; %let errorFlag=NO;

    %put; %put NOTES: --> &macroName starts;
    %macroParamsValidation(inMacroName=&macroName);
    %if &errorFlag=YES %then %goto error;

    %genericFileStructValidation(inDSN=&inGlobalParametersFile,inMacroName=&macroName);
    %if &errorFlag=YES %then %goto error;
    /*more processing steps here*/
    %put; %put NOTES: --> &macroName finishes successfully;
    %return;
    %error: %put ERROR: -->&macroName stops due to list of errors;
%mend process1;

%put; %put NOTES: ******Test case1: globalParameters has wrong file structure***;
/*only file structure of dataset globalParameters will be validated */
data globalParameters;
    selMeth=1;           /*error: Wrong data type*/
    direction1="LEFT";  /*optional variable <direction> is miss-spelt as <direction 1>*/
    start=1.1;           /*this is okay*/
    rotationRate1=0.2;  /*error: variable <rotationRate> is miss-spelt as <rotationRate1>*/
    usage="";            /*dummy variable is okay to be in the file*/
run;
data infile; /*dummy file for other input data files*/
    selMeth=1;
run;
%process1(inDataLib=WORK,
    inGlobalParametersFile=globalParameters,
    inFrameFile=infile,
    inAllocationFile=infile,
    inUpdateFlag=YES,
    inRotationRate=0.0,
    outDatalib=WORK,
    outSampleFile=outfile);
```

- Step 4: review SAS log

```
NOTES: ******Test case1: globalParameters has wrong file structure***

NOTES: --> PROCESS1 starts
ERROR: In <globalParameters> dataset, variable <SELMETH > is required and not presented
ERROR: In <globalParameters> dataset, variable <ROTATIONRATE > is required and not presented
WARNING: In <globalParameters> dataset, variable <DIRECTION > is not presented and added as <CHAR
> type
WARNING: In <globalParameters> dataset, variable <SEED > is not presented and added as <NUM > type
ERROR: -->PROCESS1 stops due to list of errors
```

The log shows that in `globalParameters` dataset, mandatory variables `selMeth` and `rotationRate` are not presented, and optional variables `direction` and `seed` are added with proper data type. Again, here validations and imputations on file structure are conducted at the same time since the two processes are similar.

3.3 AT DATA LEVEL

After ensure all input dataset have the right file structures for macro %`process1`, I move to data level validations and imputation inside each input dataset. I'll take input dataset `&inGlobalParametersFile` as an example.

1) Define Metadata on validations and imputations for data inside `&inGlobalParametersFile`

In Table 2, only these columns are used for data validations and imputations: `macorName`, `object_type`, `object_name`, `member_name`, `member_type`, `default`, `required`. The metadata is interpreted in this way:

- Variable `selMeth` is required and should be either "`SRS`" or "`BERNOULLI`";
- Variable `start` is optional and should be between `0` and `1` or `missing`; if `missing`, it is set to default value `0`;
- Variable `rotationRate` is required and should be between `0` an `1`;
- Variable `direction` is optional and should be either "`LEFT`" or "`RIGHT`" or `missing`; if `missing`, it is set to default value "`RIGHT`";
- Variable `seed` is optional and can be missing; if missing, it is set to default value `%sysfunc(time())*1000;`

2) Utilize metadata for validations and imputations on data inside dataset `&inGlobaParametersFile`

Here are the major steps of validations and imputations on data inside dataset `&inGlobalParametersFile.`

- Step 1: import metadata in the Excel file storing Table 2 into a SAS dataset, shown in step 1 in section 3.2 -> 2;
- Step 2: build generic SAS macros to validate and impute data inside an input dataset `&inDSN`;

```
%macro genericDSNSetDefault(inMacroName=,inDSN=);
    %local j defExist;
    %let inDSN=%cmpres(&inDSN);
    %let defExist =NO;

    /* get the list of default values*/
    data work.&inDSN._metaData_def;
        set work.DSN_metaData end=_eof;
        where upcase(strip(macroName)) = upcase("&inMacroName") and
              upcase(strip(resolve(object_name))) = upcase("&inDSN") and
              upcase(strip(Object_type)) = "DATASET" and
             (not missing(strip(default))) ;

        if (_eof and _n_>0) then
            call symputx("defExist","YES");
    run;
    %if &defExist = NO %then %do;
        %put NOTES: No default values to be applied to <&inDSN>;
        %return;
    %end;

    /* save the list of default values and other info into macro variables*/
    data _null_;
        retain constrCount 0;
        set work.&inDSN._metaData_def end=_eof;
        n=strip(_n_);
        call symputx("defStr"||n,resolve(strip(default)));
        call symputx("varName"||n,strip(member_name));
        if _eof then call symputx("defCnt",n);
    run;

    /* generate the code to set default for missing values*/
    data work.&inDSN(drop=defCount);
        retain defCount 0;
        set work.&inDSN end=_eof;

        %do j=1 %to %cmpres(&defCnt);
            retain defCount&j 0;
            if missing(&&varName&j) then do;
                &&varName&j..=symget("defStr&j");
                defCount&j=defCount&j+1;
                defCount = 1;
            end;
            if _eof and defCount&j > 0 then do;
                put "WARNING: In <&inDSN> dataset, default set to missing values on variable <&&varName&j>";
                drop defCount&j;
            end;
        %end;
```

The above generic macro `%genericDSNSetDefault` is to set default values for missing data values in `&inDSN`. It stores default values for variables in `&inDSN` into a sequence of macro variables; then a data step loops through each observation in `&inDSN`, and within the loop, code is generated by using the sequence of macro variables to set default values for missing data values for each variable.

The following generic macro `%genericDataValidation()` is to validate the data inside `&inDSN`. Note the macro parameter `keyID` is a string containing the key columns forming the unique key for `&inDSN`. This `keyID` parameter is used to identify at which observation, invalid data is found. This macro first stores the list of validation rules into a sequence of macro variables; then in a data step on `&inDSN`, it generates code by using the sequence of macro variables to validate data inside `&inDSN` and save invalid data in an output dataset.

```
%macro genericDataValidation(inMacroName=,inDSN=,keyID=);
    %local i j invalid_conditionExist keyIDVar;
    %let inDSN=%cmpres(&inDSN);
    %let invalid_conditionExist =NO;
    %let keyID = %upcase(%cmpres(&keyID));

    /*compose the values from the key variables of the input file*/
    %if &keyID= %then %let keyID=_n_;
    %do i=1 %to %sysfunc(countw(&KeyID));
        %let keyIDVar=%cmpres(%scan(&keyID,&i));
        %if %eval(&i=1) %then %let keyIDVarList=%sysfunc(cats("&keyIDVar=",||,&keyIDVar));
        %else
           %let keyIDVarList=%sysfunc(cats("&keyIDVar=",||,&keyIDVar,||," AND ",||,&keyIDVarList));
    %end;

    /* get the list of validation rules*/
    data work.&inDSN._metaData_cons;
        set work.DSN_metaData end=_eof;
        where upcase(strip(macroName)) = upcase("&inMacroName") and
              upcase(strip(resolve(object_name))) = upcase("&inDSN") and
              upcase(strip(Object_type)) = "DATASET" and
              (not missing(strip(invalid_condition)));
        if (_eof and _n_>0) then
            call symputx("invalid_conditionExist","YES");
    run;
    %if &invalid_conditionExist = NO %then %do;
        %put NOTES: No validation rules to be applied to <&inDSN>;
        %return;
    %end;

    /* save the list of rules and other info into macro variables */
    data _null_;
        retain invalid_conditionCount 0;
        set work.&inDSN._metaData_cons end=_eof;
        n=strip(_n_);
        call symputx("invalid_condition"||n,strip(invalid_condition));
        call symputx("errMsg"||n,strip(errMsg));
        call symputx("varName"||n,strip(member_name));
        if _eof then call symputx("consCnt",n);
    run;

    /* generate code to validate the data against rules */
    data work.&inDSN._invalid(keep=errMsg varWithError keyID);
        retain errorCount 0;
        length errMsg $500 varWithError $40;
        set work.&inDSN end=_eof;

        %do j=1 %to %cmpres(&consCnt);
            retain errCount&j 0;
            if &&invalid_condition&j then do;
                varWithError = "%bquote(&&varName&j)";
                errMsg="%bquote(&&errMsg&j)";
                keyID= &keyIDVarList;
                errCount&j=errCount&j+1;
                errorCount = errorCount+errCount&j;
                output;
            end;
            if _eof and errCount&j > 0 then
                put "ERROR: In <&inDSN> dataset, invalid data found on variable <&&varName&j>";
        %end;
        if _eof and errorCount >0 then do;
            put "WARNING: Please check <work.&inDSN._invalid> for invalid data";
            call symputx("errorFlag","YES");
        end;
    run;
%mend;
```

- Step 3: Utilize the above generic SAS macros to validate and impute data of `&inGlobalParametersFile` in SAS macro `%process1`. Macro `%process1` simply invokes `%genericDataValidation` to validate the data inside `&inGlobalParametersFile` and then invokes `%genericDSNSetDefault` to impute default values for missing data values in `&inGlobalParametersFile`.

  Testing case1 is to test invalid data in input dataset `globalParameters` dataset. Testing case 2 is to test the imputation on missing data values in `globalParameters` dataset.

```
%macro process1(inDataLib=, inGlobalParametersFile=, inFrameFile=, inAllocationFile=,
                inUpdateFlag=, inRotationRate=, outDatalib=, outSampleFile= );
    %local macroName; %let macroName=PROCESS1;
    %local errorFlag; %let errorFlag=NO;

    %put; %put NOTES: --> &macroName starts;

    %macroParamsValidation(inMacroName=&macroName);
    %if &errorFlag=YES %then %goto error;

    %genericFileStructValidation(inDSN=&inGlobalParametersFile,inMacroName=&macroName);
    %if &errorFlag=YES %then %goto error;

    %genericDataValidation(inDSN=&inGlobalParametersFile,keyID=,inMacroName=&macroName);
    %if &errorFlag=YES %then %goto error;

    %genericDSNSetDefault(inDSN=&inGlobalParametersFile,inMacroName=&macroName);
    %if &errorFlag=YES %then %goto error;
    /*more processing steps here*/
    %put NOTES: --> &macroName finishes successfully;
    %return;
    %error: %put ERROR: -->&macroName stops due to list of errors;
%mend process1;

%put; %put NOTES: ****** Test case1: validate data in globalParameters file ***;
data globalParameters;
    length selMeth $10 direction $10;
    selMeth="SRS"; direction=""; start=10; rotationRate=0.5; output;
    selMeth=""; direction="LEFT"; start=0.2; output;
run;
data infile;
    selMeth="LEFT";
run;
%process1(inDataLib=WORK,
    inGlobalParametersFile=globalParameters,
    inFrameFile=infile,
    inAllocationFile=infile,
    inUpdateFlag=YES,
    inRotationRate=0.0,
    outDatalib=WORK,
    outSampleFile=outfile
    );

%put; %put NOTES: ****** Test case2: setting defaults in globalParameters file***;
data globalParameters;
    length selMeth $10 direction $10;
    selMeth="SRS"; direction=""; start=.; rotationRate=0.2; output;
    selMeth="SRS"; direction=""; start=0.2; output;
run;
%process1(inDataLib=WORK,
    inGlobalParametersFile=globalParameters,
    inFrameFile=infile,
    inAllocationFile=infile,
    inUpdateFlag=YES,
    inRotationRate=0.0,
    outDatalib=WORK,
    outSampleFile=outfile
    );
```

- Step 4: review SAS log, invalid data, and data imputed

  Here is the SAS log for testing case 1: there are invalid data found on variable `start` and the invalid data is saved into `work.globalParameters_invalid`.

```
NOTES: ****** Test case1: validate data in globalParameters file ***

NOTES: --> PROCESS1 starts
WARNING: In <globalParameters> dataset, variable <seed > is not presented and added as <NUM > type
ERROR: In <globalParameters> dataset, invalid data found on variable <start>
WARNING: Please check <work.globalParameters_invalid> for invalid data
ERROR: -->PROCESS1 stops due to list of errors
```

By examining `work.globalParameters_invalid`, I find that the first record in `work.globalParameters` input file has wrong value for variable `start` as 10, which should be between 0 and 1.

GLOBALPARAMETERS_INVALID

Filter and Sort | Query Builder | Data | Describe | Graph | Analyze | Export | Send To

	errMsg	varWithError	keyID
1	Starting point should be either MISSING or between 0 and 1 inclusive	start	_n_= 1

Here is the SAS log for testing case 2: First, the input file passes data validations, then missing values are set to default for variables `seed, start, and direction`.

```
NOTES: ****** Test case2: setting defaults in globalParameters file***

NOTES: --> PROCESS1 starts
WARNING: In <globalParameters> dataset, variable <seed > is not presented and added as <NUM > type
WARNING: In <globalParameters> dataset, default set to missing values on variable <start>
WARNING: In <globalParameters> dataset, default set to missing values on variable <direction>
WARNING: In <globalParameters> dataset, default set to missing values on variable <seed>
NOTES: --> PROCESS1 finishes successfully
```

By comparing the data in `globalParameters` dataset before and after imputation, I can verify that the imputation is done properly. Note, default value for `seed` is generated by `time()` function which changes depending on when the generic macro `%genericDSNSetDefault` is invoked.

Before imputation

	selMeth	direction	start	rotationRate
1	SRS		.	0.2
2	SRS		0.2	0.2

After imputation:

	selMeth	direction	start	rotationRate	seed
1	SRS	RIGHT	0	0.2	16525.114
2	SRS	RIGHT	0.2	0.2	16525.114

3.4 PORTABILITY AND REUSABILITY

Generic SAS macros described in 3.1, 3.2 and 3.3 for three levels of validations and imputations using metadata are portable and reusable. I'll take another example to explain this. Here a new macro `%process2` to be built:

```
%macro process2(inDataLib=,instartingValuesFile,
                outDatalib=,outAllocationFile=);
    /*processing steps here*/
%mend;
```

To conduct the three levels of validations and imputation for `%process2`, the only work to be done is to define its metadata properly and append them into the same Excel files where metadata for macro `%process1` are stored; then re-import the Excel files. Then macro `%process2` simply invokes these generic SAS macros `%macroParamsValidation`, `%genericFileStructValidation`, `%genericDataValidation`, `%genericDSNSetDefault` by feeding them with proper information. No other system coding efforts are required. This greatly shortens development time for validations and imputations processes in macro `%process2`.

Next, major steps of conducting three levels of validations and imputations for macro `%process2` are explained to prove it is very easy to implement it.

- Step1: Define metadata for three levels of validations and imputations for SAS macro `%process2`.

  The metadata for its macro parameters in Table 3 explains that `inDatalib` and `outDataLib` are optional macro parameters specifying library names; `inStaringValuesFile` and `outAllocationFiles` are mandatory macro parameters specifying dataset names.

object_type	object_name	member_name	member_type	IO_type	invalid_condition	errMsg	default	required
MACRO	PROCESS2	inDataLib	LIBRARY	INPUT			WORK	NO
MACRO	PROCESS2	inStartingValuesFile	DATASET	INPUT				YES
MACRO	PROCESS2	outDatalib	LIBRARY	OUTPUT			WORK	NO
MACRO	PROCESS2	outAllocationFile	DATASET	OUTPUT				YES

Table 3: Metadata for validations and imputations on macro parameters of macro `%process2`

The metadata for one of the input dataset `&inStartingValueFile` in Table 4 includes both metadata for file structure and for the data. Variable `StratID` is mandatory and is numeric data type and must be integer. Variable `size` is optional and is numeric data type, and its default values is 1.

macroName	object_type	object_name	member_name	member_type	invalid_condition	errMsg	default	required
PROCESS2	DATASET	&inStartingValuesFile	stratID	NUM	NOT(int(stratID)=stratID)	<stratID> has to be an integer		YES
PROCESS2	DATASET	&inStartingValuesFile	size	NUM	(size>1000 or size<1) and not(missing(size))	<size> has to be between 1 and 1000 or missing	1	NO

Table 4: Metadata for validations and imputations on input dataset `&inStartingValulesFile`

- Step2: re-import metadata Excel files into SAS datasets (Table 3 is appended to the same Excel file storing metadata in Table 1 and Table 4 is appended to the same Excel file storing metadata in Table 2)
- Step3**:** reuse generic SAS macros to conduct three levels of validations and imputations for `%process2`**.**

Below is the code example. Note that in macro `%process2`, first `%macroParamsValidation()` validates and imputes macro parameters of `%process2`; next `%genericFileStructValidation()` validates and imputes file structure of `&inStartingValuesFile`; next `%genericDataValidation()` validates the data `inside &inStartingValuesFile`, and at the end `%genericDSNSetDefault()` imputes default values for missing values in `&inStartingValuesFile`. Four testing cases is built to test validation and imputation processes.

```
%macro process2(inDataLib=,instartingValuesFile=,
                outDatalib=,outAllocationFile= );
    %local macroName; %let macroName=PROCESS2;
    %local errorFlag; %let errorFlag=NO;

    %put; %put NOTES: --> &macroName starts;

    %macroParamsValidation(inMacroName=&macroName);
    %if &errorFlag=YES %then %goto error;

    %genericFileStructValidation(inDSN=&instartingValuesFile,inMacroName=&macroName);
    %if &errorFlag=YES %then %goto error;

    %genericDataValidation(inDSN=&instartingValuesFile,keyID=stratID,inMacroName=&macroName);
    %if &errorFlag=YES %then %goto error;

    %genericDSNSetDefault(inDSN=&instartingValuesFile,inMacroName=&macroName);
    %if &errorFlag=YES %then %goto error;
    /*more processing steps here*/
    %put NOTES: --> &macroName finishes successfully;
    %return;
    %error: %put ERROR: -->&macroName stops due to list of errors;
%mend process2;

%put; %put NOTES: ****** Test case1: validate macro params of process2 ***;
%process2(inDataLib=abcdefghijk,
          instartingValuesFile=abcdefghijklmnopqrstuvwxyz123456790,
          outDatalib=,
          outAllocationFile=allocFile);

%put; %put NOTES: ****** Test case2: validate file structure of inStartingValuesFile ***;
data startingValuesFile;
    stratID1=10; size=50; output; /*mis-spell stratID to stratID1*/
    stratID1=20; size=30; output;
run;

%process2(inDataLib=WORK,
          instartingValuesFile=startingValuesFile,
          outDatalib=,
          outAllocationFile=allocFile);

%put; %put NOTES: ****** Test case3: data inside inStartingValuesFile ***;
data startingValuesFile;
    stratID=10; size=5000; output; /*Error: size is not valid*/
    stratID=20; size=.; output;
    stratID=3.5; size=.; output;   /*Error: stratID is not an integer*/
run;

%process2(inDataLib=WORK,
          instartingValuesFile=startingValuesFile,
          outDatalib=,
          outAllocationFile=allocFile);

%put;
%put NOTES: ****** Test case4: impute default values for missing values in inStartingValuesFile ***;
data startingValuesFile;
    stratID=10; size=1000; output;
    stratID=20; size=.; output;   /*Warning: size is set to default value 1*/
run;

%process2(inDataLib=WORK,
          instartingValuesFile=startingValuesFile,
          outDatalib=,
          outAllocationFile=allocFile);
```

- **Step 4**: Review SAS logs

  In testing case 1: a list of macro parameters errors are found and macro `%process2` stops with errors;

```
NOTES: ****** Test case1: validate macro params of process2 ***

NOTES: --> PROCESS2 starts
ERROR: For macro <PROCESS2>, SAS library <abcdefghijk> provided by param <inDataLib> is longer than 8
characters
ERROR: For macro <PROCESS2>, SAS dataset name <abcdefghijklmnopqrstuvwxyz123456790> provided by param
<inStartingValuesFile> is longer than 32 characters
WARNING: For macro <PROCESS2>, macro parameter of <LIBRARY> type <outDatalib> is empty and set to
default as <WORK>
ERROR: -->PROCESS2 stops due to list of errors
```

  In testing case 2: all macro parameters are provided properly, but errors are found in file structure of input dataset `&inStartingValuesFile`; `%process2` stops with errors;

```
NOTES: ****** Test case2: validate file structure of inStartingValuesFile ***

NOTES: --> PROCESS2 starts
WARNING: For macro <PROCESS2>, macro parameter of <LIBRARY> type <outDatalib> is empty and set to
default as <WORK>
ERROR: In <startingValuesFile> dataset, variable <stratID > is required and not presented
ERROR: -->PROCESS2 stops due to list of errors
```

  In testing case 3: all macro parameters and input datasets file structure are provided properly, but errors are found in data in input dataset `&inStartingValuesFile.` `%process2` stops with errors;

```
NOTES: ****** Test case3: data inside inStartingValuesFile ***

NOTES: --> PROCESS2 starts
WARNING: For macro <PROCESS2>, macro parameter of <LIBRARY> type <outDatalib> is empty and set to
default as <WORK>
ERROR: In <startingValuesFile> dataset, invalid data found on variable <stratID>
ERROR: In <startingValuesFile> dataset, invalid data found on variable <size>
WARNING: Please check <work.startingValuesFile_invalid> for invalid data
ERROR: -->PROCESS2 stops due to list of errors
```

  In testing case 4: No validation error is encountered in the three levels of validations and `%process2` passes all validations and imputations successfully;

```
NOTES: ****** Test case4: impute default values for missing values in inStartingValuesFile ***

NOTES: --> PROCESS2 starts
WARNING: For macro <PROCESS2>, macro parameter of <LIBRARY> type <outDatalib> is empty and set to
default as <WORK>
WARNING: In <startingValuesFile> dataset, default set to missing values on variable <size>
NOTES: --> PROCESS2 finishes successfully
```

  The three levels of validation and imputations for the new macro `%process2` becomes very simple by reusing the existing generic SAS macros shown in 3.1, 3.2, and 3.3. This greatly eases our system development.

4. BENEFITS

G-Sam system is released for user testing already, in which metadata driven programming technique is used for three levels validations and imputation in SAS macros. Based on my experience, here are key benefits gained:

- **Efficiency in system development**: the lines of code for three levels of validations and imputations are shortened significantly. Also efficiency is gained by reusing generic SAS macros for three levels of validation and imputations.
- **Reusability**: one aspect is that generic metadata driven SAS macros for validations and imputations can be ported to any other SAS systems. Another aspect is that developers' knowledge of meta-data driven programming can be reutilized in other SAS® systems development once they understand it well.
- **Easy maintainability**: when validation and imputation rules are changed, required changes on system side are only in metadata files and no system processing code needs to be modified. This eliminates the risk of introducing new bugs into existing system and makes it easy to maintain the processes of validations and imputations. Since validation and imputation rules are centralized metadata files, it is easy for users and system developers to examine the full business rules of the system globally.
- **Consistency**: Cross the whole systems, validations and imputations are done in a consistent way. It makes easy for new resource to understand the processes of validations and imputations.

- **Modularization**: The metadata on validations and imputations are defined in template Excel files, then the files are imported into SAS datasets for utilization. A metadata maintainer can be trained to be central expertise for defining the metadata using SAS syntax without needing knowing systems processes. Then he can also take care of the generic meta-data driven SAS macros and insert the block of template code for validations and imputations to create a pre-template SAS macro for any new SAS macros to be built. Lastly, other developers can focus on building the main system processes by using the pre-template SAS macros. Modularized system development can save development efforts also and ensure the systems to be easily maintained..

5. LIMITATIONS AND CHALLENGES

- Since metadata is imported into a dataset. I need to interoperate two datasets: the metadata dataset and the user input datasets. In our implementation, the values in one of the dataset will be saved into a list (macro variables separated by a delimiter or a sequence of linked macro variables). Then utilize this list within a data step of another dataset.
- To ease system coding, constraints and defaults have to be written in SAS® format. based on my experience, it is better have one person to take care of defining metadata to ensure its correctness and consistency. a junior SAS® programmer can be easily trained to do this and a senior developer or project leader should always review the metadata before they formally used by the team.
- There are always exceptions when validation and setting default are not straightforward and metadata cannot be easily abstracted. For example, if a variable's default value is based on data from another input dataset, then the default value cannot be register into the metadata. Setting default for this variable has to be done outside the metadata driven imputation block. It is essential to analyze systems specifications carefully in order to utilize meta-data driven validations and imputation at maximum to gain its benefits.

6. CONCLUSION

This paper demonstrates how metadata driven programming technique is used for three levels of validations and imputations when building SAS macros. Through examples, step by step explanation is given to help readers fully understand this technique. Then, the paper illustrates how this programming technique can ensure strong reusability, easy maintainability, and good coding consistency, thus shortens system development efforts. Lastly, challenges and benefits are summarized.

ACKNOWLEDGMENTS

Special thanks to my section chief Yves Deguire for his strong support on my experiment of using meta-data driven programming in G-Sam project.

CONTACT INFORMATION

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

Name : Xiyun (Cheryl) Wang
Enterprise : Statistics Canada (http://statcan.gc.ca)
Work Phone: (613) 951-0843
E-mail : cherylxiyun.wang@statcan.gc.ca