PhUSE 2009

Paper DH04

# Validate it Again?! Get SAS to Do It!

Carol Matthews, United BioSource Corporation, Pennsylvania, USA
Ginger Lewis, United BioSource Corporation, Pennsylvania, USA
Olga Belotserkovsky, United BioSource Corporation, Pennsylvania, USA

## ABSTRACT

In today's pharmaceutical research environment, clinical project teams often want to see statistical output well before database lock and also want to see it frequently throughout the course of the study. Changing data values can affect the results of validated programs in subtle, yet significant ways. As a result, it is critical to revalidate programs after each data update to prevent errors in output. With this in mind, it is more important than ever for programmers to make their SAS® code do as much of the validation work as possible. This paper will discuss several simple techniques that will put Base SAS to work looking for data issues that could adversely affect your program's result.

## INTRODUCTION

When discussing SAS program validation, many think of it in two circumstances – either at the beginning of a project or at the end. At the beginning of a project, programs are validated and then may be run several times later but since "validation" was already done, many believe that it does not need to be repeated on new data unless an obvious issue arises (programs don't run or your client notices problems in the output). Alternately, programs may be run on interim data and not be validated until the end of the project after final data is received. Both of these scenarios are unrealistic in today's pharmaceutical research environment. While validation at the end of a project is more conservative if output is being submitted to a regulatory authority, it is rare that this is the only time that programs will be run and output reviewed. Project teams are often anxious to see preliminary output well before the database is locked, especially in clinical trials that take more than six months to complete enrollment.

It is important to realize that when programs are validated, they are only considered valid in relation to the data that they are run against. If programs are run against new or updated data files, they must be reviewed again to ensure that the results are still valid. If you assume the new data has the same structure and potential variable values as the original data, the chance is very low that a validated program will yield an incorrect result when run against the new data. However, if the data structure or variable values have changed, it is possible that the program will not perform as planned will produce invalid results. Consequently, it is critical to check that any new data received still meets any assumptions made regarding structure and critical variable values.

While data is the main focus in any clinical trial, programs that create summary reports based on that data also need to be revalidated against new data prior to delivery to a client (even if that client is internal to your company). In addition to the primary programmer, many companies perform independent validation of derived data sets (any permanent data set that is created as the result of running a program) and summary tables and graphs. If this independent validation is a manual process, the time needed to validate after each data update can be time and cost prohibitive. However, if output that is reviewed is incorrect and decisions are made based on invalid information, much more time may be lost. The key is to make the validation process as efficient as possible.

In light of the strong possibility that code will be run multiple times against data that will change over time, it is more important than ever that SAS programs be built to automate much of the validation process rather than depending on manual comparisons. Throughout the process of receiving data to generating summary tables, graphs and data listings, there are three major points at which SAS can be made to facilitate the efficient and effective validation of code. Upon data import, SAS can check to ensure that the structure and the values of key variables in the new data received are the same as the previous data. Once critical aspects of the new data are confirmed, techniques can be used when programming derived data sets to further ensure that the data contains logical values. Finally, techniques usually reserved for validation of data sets can be used to significantly reduce the time required for independent validation programmers to validate summary tables and graphs.

## VALIDATE THE NEW AGAINST THE OLD

Each time data is updated, there is a possibility that the data structure has changed or that new, unexpected values in the data will create problems in programs validated against older data. For example, new unscheduled visits might not be decoded properly in derived data sets or summarized correctly on tables that present "by visit" results. You

might also find that the database structures change as the result of changes in extraction programs that pull data from an electronic data capture (EDC) system. Base SAS can be used to compare newly imported data to original data and quickly identify differences in key variable values or changes in the data structure that might impact existing programs generating derived data sets or tables. Macros can be used to make these checking programs useful across studies and one example is described below.

#### COLLECT DATA ABOUT DATA SETS IN LIBRARIES

In many cases, receiving an updated clinical trial database involves many SAS data sets. Rather than review one data set at a time, it is more efficient for SAS to read all of the data sets in a library and review each automatically. PROC SQL automatically retrieves information from SAS data libraries into special read-only tables that can be used in other SAS procedures and DATA steps. You can collect all of the data set names in a library into a macro string and then use that string to generate frequencies or to compare data sets without having to list them individually. The SQLOBS macro automatically created by PROC SQL returns the number of data sets found in a library and can be used as the upper limit for DO loops to cycle through all of the data sets in that library.

```
%global coreobsnewlib coredsnnew;

%macro getlibs(libnam);
   ** GET LIST OF DATASET NAMES INTO MACRO VARIABLES FROM LIBRARY  **;
   title1 "CORE DATASET MEMBERS IN &LIBNAM";
   proc sql;
      select distinct memname into :coredsn separated by "*"
      from dictionary.members
      where upcase(libname) = upcase("&libnam") and upcase(memtype) = "DATA";
   quit;
   %let coreobs&libnam=&sqlobs;
   %if &libnam=newlib %then %let coredsnnew=&coredsn;

   ** PUT LIST OF DATASET NAMES AND VARIABLE LISTS INTO MACRO STRINGS **;
   %do i = 1 %to &&coreobs&libnam;

      %let dsnam=%scan(&coredsn,&i,"*");

      %if %length(&dsnam)>3 %then
         %do;
            %if %substr(&dsnam,1,4)=SUPP %then
               %do;
                  %let &&dsnam=idvar idvarval qnam qlabel;
               %end;
            %else
               %do;
                  %let &&dsnam=&dval;
               %end;
      %end;
      %else
         %do;
            %let &&dsnam=&dval;
         %end;
      %varlists

      **----- COUNT NUMBER OF VARIABLES IN LIST  -----**;
      %let count=1;

      %let word=%quote(%nrquote(%scan(&&&dsnam,&count,%str( ))));

      %do %while(&word^=);
          %let var&count=%unquote(&word);
          %let count=%eval(&count+1);
          %let word=%quote(%nrquote(%scan(&&&dsnam,&count,%str( ))));
      %end;
```

```
%let cnt = %eval(&count-1);

%put >>>>> DATACHECK: NAME OF DATASET BEING PROCESSED &dsnam ;
%put >>>>> DATACHECK: VARLIST TO PROCESS &&&dsnam ;
%put >>>>> DATACHECK: NUMBER OF VARIABLES TO PROCESS = %eval(&count-1);
```

**CHECK THE VALUES OF KEY VARIABLES WITH PROC FREQ**

Once the number of data sets and variables are saved into macro variables, you can cycle through code to perform various checks against each data set in the library. In this example, we use PROC FREQ to compare the data value differences in old data as compared to new.

```
*****************************************************************;
**----- GET FREQS FROM CURRENT DATA OF IDENTIFIED VARIABLES ----**;
*****************************************************************;

%do j = 1 %to &cnt;
    %let tabvar=%scan(&&&dsnam,&j,%str( ));
    proc freq data=&libnam..&dsnam  noprint;
       tables &tabvar / list out=&dsnam&j;
    run;

    ** DETERMINE IF VARIABLE IS CHARACTER OR NUMERIC FOR RENAME **;
    ** WHEN SETTING ALL FREQUENCIES TOGETHER                    **;
    data _null_;
       set &dsnam&j;
       call symput("vartype"||"&j",vtype(&tabvar));
       call symput("varnam"||"&j","&tabvar");
    run ;
%end;
```

**CHECK THE STRUCTURE OF THE DATA**

In the same loop that checks variable values in each data set, PROC CONTENTS can be run for each data set with the OUT option which will store the results in data sets to perform PROC COMPAREs later.

```
*****************************************************************;
**----- GET PROC CONTENTS TO USE FOR COMPARE OF STRUCTURES ----***;
*****************************************************************;

proc contents data=&libnam..&dsnam
   out=&libnam&dsnam (keep=memlabel label length name type) noprint ;
run ;

proc sort data=&libnam&dsnam out=&libnam&dsnam ;
   by name ;
run ;

**----- CREATE DATASET WITH BLANK VARIABLES TO SET WITH OTHER FREQUENCIES --**
**----- CREATES TESTVAR LENGTH                                          -----**;
data blank;
   length testvar $100;
   testvarn=.;
   testvar=' ';
run ;

**----- SET ALL FREQUENCIES TOGETHER -----**;
data &dsnam (drop=count testvarn percent);
  ** RENAME NUMERIC AND CHARACTER SEPARATELY **;
```

```
      set
         blank
         %do j = 1 %to &cnt;
            &dsnam&j (rename=(
               %let tabvar=%scan(&&&dsnam,&j,%str( ));
               %if &&vartype&j=C %then &tabvar = testvar;
                  %else &tabvar=testvarn;
                  ) in=&dsnam&j)
         %end;
         ;

      length varname dsname $10 ;

      if testvarn^=. then testvar=left((put(testvarn,best8.)));
                     else testvar=left(testvar);
      if testvar^=' ' ;

      **----- SAVE DATASET AND VARIABLE NAMES AS VALUES FOR REPORTING -----**;
      dsname="&dsnam";
      %do j = 1 %to &cnt;
         if &dsnam&j=1 then varname=upcase("&&varnam&j");
      %end;
    run ;

  %end;

  data &libnam.frq;
     set
       %do i = 1 %to &&coreobs&libnam;
          %let dsnam=%scan(&coredsn,&i,"*");
          &dsnam
       %end;
       ;
  run ;

  proc sort data=&syslast nodupkey out=&syslast;
     by dsname varname testvar ;
  run ;

%mend getlibs;
```

**REVIEW THE RESULTS**

Once the information for both the old and new data are collected into data sets with the %getlibs macro, a variety of reports can be created. In this example, %varlists sets the list of variables to check in the specified data sets. Alternatively, a default list of variables can be used if no list is specified.

```
%let dval=studyid ; ** SET THIS VALUE TO DEFAULT VAR COMMON TO ALL DATASETS **;

%macro varlists /** SAMPLE VARIABLE LIST **/;
   %let ae = aecat aescat aeser aeacn aerel;
   %let cm = cmgrpid cmscat visit visitnum ;
   %let dm = sex race country ;
   %let qs = qsgrpid qstestcd qscat visit;
%mend;
%varlists

%getlibs(newlib)

%getlibs(oldlib)
```

```
**----- COMPARE DATA IN OLD SEND TO NEW AND KEEP DIFFERENCES -----**;
data diffdata;
   merge newlibfrq (in=ina)
         oldlibfrq (in=inb);
   by dsname varname testvar;
   length flag $30;

   if ina^=inb;
   if ina then flag="Current Data only (NEWLIB)";
   else if inb then flag = "Old Data only (OLDLIB)";
run ;
```

A simple PROC PRINT of the data set created above produces the report below which identifies data values in the old data that are not in the new or vice versa.

| Dataset | Unique to Which Library? | Variable Name | Value |
|---|---|---|---|
| SUPPCM | Current Data only (NEWLIB) | IDVARVAL | 01-01 |
| | | IDVARVAL | 02-01 |
| | | IDVARVAL | 02-03 |
| | | IDVARVAL | 03-03 |
| | | IDVARVAL | CMGRPID |
| | Old Data only (OLDLIB) | IDVAR | CMGRPD |
| | | IDVARVAL | 01-1 |
| | | IDVARVAL | 02-1 |
| | | IDVARVAL | 02-3 |
| | | IDVARVAL | 03-1 |

Another loop of code can perform PROC COMPARE on the results of the PROC CONTENTS statements above to compare the structures of the files in the old and new libraries.

```
**----- PROC COMPARE DATA STRUCTURES AND OUTPUT DIFFERENCES -----**;
%macro cmpare;
       %put _all_;

       %do i = 1 %to &coreobsnewlib;
          %let dsnam=%scan(&coredsnnew,&i,"*");
          proc compare base=newlib&dsnam compare=oldlib&dsnam;
             id name;
             title “COMPARE RESULTS FOR &DSNAM” ;
          run ;
       %end;
%mend;
%cmpare
```

The PROC COMPARE results will identify changes in data structures that might require changes in programs using the data set. In this example excerpt, the length of the subject identifier has changed significantly for one of the data sets in the library.

COMPARE RESULTS FOR LABS

| NAME | Variable Length Base LENGTH | Compare LENGTH | Diff. | % Diff |
|---|---|---|---|---|
| SUBJID | 20.0000 | 11.0000 | -9.0000 | -45.0000 |

## VALIDATE THE DERIVED DATA SETS

Whether creating CDISC SDTM data sets from “raw” data or creating analysis data sets to generate summary tables, the data sets you create with a SAS program need to be validated. All calculations and decodes need to be verified every time new data is manipulated, as new data can fall out of code that had worked perfectly fine with older data. It is an inexperienced programmer who believes that just because their program runs without errors in the log, the program result is correct. Rather than pour over logs and pages of output every time new data is received, there are a number of techniques that can enable SAS to perform the time-consuming part of validation for you.

### SAS SYSTEM OPTION – DSOPTIONS

Perhaps the most useful SAS System Option for validation purposes is dsoptions="note2err". This option requires programmers to have clean logs (no notes or warnings) and then alerts them if data changes cause code not to work correctly. This option forces what are typically NOTEs in the log to be reflected as fatal ERRORs instead. For example, “NOTE: Character values have been converted to numeric values…” will show as “ERROR: Character value found where numeric value needed at line xx column yy” and SAS will stop processing just as if this issue was a fatal error. With this option it will be obvious where issues are within programs that ran cleanly against previous data but now generate notes when run against new data.

Sample log excerpt:

```
12         options dsoptions="note2err" ;
13         data dem ;
14            set dem ;
15            sexn = sexcd*1 ;
ERROR: Character value found where numeric value needed at line 15 column 11.
ERROR: Character value found where numeric value needed at line 15 column 11.
ERROR: Character value found where numeric value needed at line 15 column 11.
16         run ;

NOTE: The SAS System stopped processing this step because of errors.
NOTE: SAS set option OBS=0 and will continue to check statements.
```

Rather than review lines of the log and try to decide which notes are truly “ok” and which indicate a change in the data which could cause problems in the final product of the program, the program should start with no notes (fix the example above by using the PUT function) and the dsoptions=”note2err” option should be set. For future program runs, any updates in the data that result in something unexpected will stop the program from running and the programmer will know that something important has changed.

### CHECK FOR UNEXPECTED VALUES

There are many cases where data is collected as a code (1,2 or M,F) and needs to be decoded or stored as the full meaning of the value. A typical method to do this is to apply a user-defined format with PROC FORMAT and the PUT or INPUT function. When creating formats to decode data, it is simple to add one more value to alert the programmer in the event that an unexpected value occurs in the data. In addition, continuous variables often have reasonable limits (you wouldn’t expect a heart rate less than 50 or more than 100 beats per minute). Again, formats can help to check that the data values fall within expected limits. If a consistent system for checking with formats is developed, then a macro can easily check the data and produce warnings when unexpected data is encountered. Consider the following example.

PROC PRINT of test data set:

| Obs | charvar | idvar | catv | testv | testv2 |
|---|---|---|---|---|---|
| 1 | | . | . | . | . |
| 2 | sample text | 1 | 2 | 7.5 | 32 |
| 3 | sample text | 2 | 1 | 1.8 | 90 |
| 4 | sample text | 3 | 2 | 3.6 | 22 |
| 5 | more text | 4 | 1 | 7.9 | 39 |
| 6 | more text | 5 | 2 | 1.2 | 18 |
| 7 | more text | 6 | 1 | 7.8 | 43 |
| 8 | more text | 7 | 2 | 9.7 | 26 |
| 9 | last text | 8 | 1 | 7.1 | 55 |
| 10 | last text | 9 | 1 | . | 55 |

For each of the variables in the test data set, a format is created to verify the acceptable values. Some are simply expected not to be missing (if your record ID variable is missing, chances are there is a problem). Others are meant to be decoded (1 becomes “Male” and 2 becomes “Female”) and while missing values are acceptable, values other than expected are not. Still other variables are expected to be within a reasonable range. The PROC FORMAT code below shows how the formats can be defined to check different types of variables. Notice that the name of the format is the same as the name of the variable it is testing, except in the case of the TESTV2 variable (this ends in “x” by convention).

```
proc format ;
   /* CHECK A RANGE OF VALUES, DIFFERENTIATE MISSING */
   value testv
      2 - 8 = 'ok'
          . = '*missing*'
      other = 'INVALID'
            ;
   /* CHECK A RANGE OF VALUES, MISSING IS OK */
   value testv2x
      20 - 100 = 'ok'
             . = 'ok'
         other = 'INVALID'
               ;
   /* ONLY MISSING VALUES ARE INVALID */
   value idvar
          . = 'INVALID'
      other = 'ok'
            ;
    /* DECODE EXPECTED VALUES, OTHERS ARE INVALID */
   value catv
          1 = 'Category 1'
          2 = 'Category 2'
      other = 'INVALID'
            ;
   /* CHECK CHARACTER VARIABLE FOR MISSING AND INVALID VALUES */
   value $charvar
              " " = "MISSING"
     "sample text" = "ok"
       "more text" = "ok"
             other = "INVALID"
            ;
run ;
```

Knowing that the convention for creating formats is 1) to name them the same as the variable, unless the variable ends in a number in which case an “x” is added as the last character, 2) invalid values are set to the value “INVALID” and 3) missing values are set to “MISSING” it is possible to create a macro to report the data issues both in the log and as output. In this example, the macro call would look as follows:

```
%mchkval(inset=test,vlist=idvar catv testv testv2 charvar,misserr=Y)
```

In the macro call above, there are three parameters being passed. INSET is the name of the data set being checked, VLIST is the list of variables to check, and MISSERR is set to Y so missing values are considered “invalid.” The resulting output is below.

```
MCHKVAL: INVALID VALUES IN KEY VARIABLES

                    Invalid
       Variable      Value      Invalid
Obs      Name        Type        Value

 1     idvar       INVALID          .
 2     catv        INVALID          .
 3     testv       *missing*        .
```

```
4     testv       INVALID        1.2
5     testv       INVALID        1.8
6     testv       INVALID        9.7
7     testv2      INVALID         18
8     charvar     INVALID      last text
9     charvar     MISSING
```

In addition to the printed list of invalid values above, the macro produces a warning message in the log “WARNING: INVALID VALUES FOR KEY VARIABLES IN test DATASET”. With the message in the log, even if you only check the log for “Warning” and “Error” will see that there are problems with the data. It is important to remember that this is only a message in the log, it does not stop program execution and there is no error message at the bottom of the program – you would need to search the log for the word “warning”. The source code for this checking macro is provided below:

```
%macro mchkval(inset=,vlist=,misserr=N) ;

   /* PARSE VARIABLE LIST INTO SEPARATE MACRO VARS */
   %let count=1;
   %let word=%quote(%nrquote(%scan(&vlist,&count,%str( ))));
   %do %while(&word^=);
      %let var&count=%unquote(&word);
      %let count=%eval(&count+1);
      %let word=%quote(%nrquote(%scan(&vlist,&count,%str( ))));
   %end;

   %let cnt = %eval(&count-1);
   %put >>>>> NUMBER OF VARIABLES TO PROCESS = %eval(&count-1);

   /* LOOP ONCE FOR EACH VARIABLE PASSED IN... */
   %do i = 1 %to &cnt ;

      /* ASSUME FORMAT NAME IS SAME NAME AS VARNAME UNLESS LAST LETTER
         IS A NUMBER - THEN REPLACE WITH X SINCE INVALID FOR USER-DEF
         FORMATS TO END IN A NUMBER */
      data _null_ ;
         if anydigit(substr(reverse("&&var&i"),1,1))>0 then
            call symput("fmt",compress("&&var&i")||"x") ;
         else call symput("fmt", "&&var&i") ;
      run ;
      %put >>> FORMAT FOR &&VAR&I = &fmt ;

      /* DETERMINE IF VAR BEING PROCESSED IS NUM OR CHAR */
      proc contents data=&inset (keep=&&var&i) out=_cc (keep=type) noprint ;
      run ;

      data _null_ ;
         set _cc ;
         call symput("vtyp", put(type,1.)) ;
      run ;

      /* PUT VALUES FOR VAR INTO STANDARD VARIABLES */
      data _zz (keep=_varnam _varval _varvalf) ;
         set &inset ;
         length _varnam _varvalf $25 _varval $50 ;
         _varnam = "&&var&i" ;
         _varvalf= put(&&var&i, &fmt..) ;
         /** IF VARIABLE TYPE IS NUMERIC, MAKE CHARACTER **/
         %if &vtyp=1 %then %str( _varval = put(&&var&i,best12.); );
         %else %str( _varval = &&var&i ; );
      run ;
```

```
      /* GET FREQ OF EACH VARIABLE */
      proc freq data=_zz ;
         table _varnam*_varvalf*_varval / out=o_&&var&i (drop=percent) noprint ;
      run ;

      %if &i=&cnt %then
      %do ;
         /* ON LAST LOOP, SET ALL FREQS TOGETHER AND ONLY KEEP INVALID VALUES */
         data _yy ;
            set %do j = 1 %to &cnt ;
                   o_&&var&j (in=in&j
                              where=(upcase(_varvalf) in('INVALID'
                                     %if &misserr^=N %then
                                         %str(' ' 'MISSING' '*MISSING*') ;
                               )))
               %end ; ;
             %if &misserr^=N %then
             %do k = 1 %to &cnt ;
                 if in&k and &&var&k eq ' ' then &&var&k = '*missing*' ;
             %end ;
             label _varnam  = "Variable Name"
                   _varvalf = "Invalid Value Type"
                   _varval  = "Invalid Value"
                   ;
         run ;

         /* PRINT VARS WITH INVALID VALUES */
         proc print data=_yy label ;
            var _varnam _varvalf _varval ;
            title "MCHKVAL: INVALID VALUES IN KEY VARIABLES" ;
         run ;

         /* IF CHECKING DATASET HAS OBS THEN PUT WARNING IN LOG */
         data _null_ ;
            if 0 then set _yy nobs=numobs ;
            call symput("nyy", left(put(numobs,5.))) ;
            stop ;
         run ;
         %if &nyy ne 0 %then
             %put WARNING: INVALID VALUES FOR KEY VARIABLES IN &INSET DATASET ;
      %end ;
   %end ;

%mend mchkval ;
```

**SET DATA TRAPS**

In addition to the values in existing variables, logic is often applied to derive new variables. A variety of techniques can be used to apply logic to create these new variables. It is important to include “traps” in your code for data that do not fit the logic or data assumptions used in creating new variables. While adding this code make take a little more time initially, it can save time over the life of the project as SAS can now check that data meets assumptions rather than the person responsible for running the program on new data.

In the course of creating derived data sets, data sets are often combined and assumptions are often made about how the data will merge, for example, when merging demographic data with physical exam data it is expected that every record in physical exam will have a matching record in the demographics data set. While the current data may merge correctly, the data you receive next may not. Using the IN= option with each data set being merged and then outputting records that do not meet assumptions is critical to the validation process.

In both of the cases above, placing the data that does not meet expectations into a separate data set allows for the use of a macro that can produce both a data listing and a warning message in the log. Consider the code example below:

```
data demo problem check ;
   merge demo (in=indemo)
         trts (in=intrts) ;
   by screenno ;
   if indemo and not intrts then output problem ;
   ** COMBINE RACE DECODE WITH SPECIFY **;
   length race $50 ;
   if racecd lt 9 then race = put(racecd,racecd.) ;
   else if racecd eq 9 then
      do ;
         if raceoth ne '' then race = raceoth ;
                         else race = "Other, not specified" ;
      end ;
   ** MAKE SURE SPECIFY FIELD NOT FILLED FOR PRE-SPECIFIED RACES **;
   if racecd lt 9 and raceoth ne '' then output check ;
   if indemo and intrts then output demo ;
run ;

%mwarn(inset=problem,why=DATA IN DEMO BUT NO MATCHING TRT INFO)

%mwarn(inset=check,why=RACE SPECIFIED FOR RACE NOT OTHER)
```

In this example, two items are being checked – that the result of the merge is what was expected and that the data being manipulated to create the new variable RACE does not contain any surprises. While it is simple to check these items, it can be time consuming to check it over and over with each new data update. This macro, in addition to the simple PROC PRINT, also produces a warning and error message in the log that can easily be found when searching the log for issues. A partial excerpt from the log that results from executing the code above is shown below.

```
NOTE: There were 10 observations read from the data set WORK.DEMO.
NOTE: There were 10 observations read from the data set WORK.TRTS.
NOTE: The data set WORK.DEMO has 9 observations and 10 variables.
NOTE: The data set WORK.PROBLEM has 1 observations and 10 variables.
NOTE: The data set WORK.CHECK has 1 observations and 10 variables.

85         %mwarn(inset=problem,why=DATA IN DEMO BUT NO MATCHING TRT INFO)

NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

NOTE: There were 1 observations read from the data set WORK.PROBLEM.
NOTE: The PROCEDURE PRINT printed page 3.
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


WARNING: DATA IN DEMO BUT NO MATCHING TRT INFO
ERROR: Execution terminated by an ABORT statement at line 85 column 37.
ERROR: Execution terminated by an ABORT statement at line 85 column 37.
ERROR: Execution terminated by an ABORT statement at line 85 column 37.
_ERROR_=1 _N_=1
NOTE: The SAS System stopped processing this step because of errors.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

The ultimate result of this macro will depend on your operating environment. If this macro is run in a batch environment, OBS will be set to 0, SAS will continue checking syntax and the bottom of the log will also show that there are errors ("ERROR: Errors printed on page 4."). In the interactive environment, the program will continue to execute as normal (OBS will *not* be set to 0 and there will *not* be an error message at the bottom of the log). These error messages are accomplished via a DATA _NULL_ with an ABORT statement and the behavior of this statement is dependent on the environment in which you execute it.

In addition to the log messages, a simple print is produced which contains the problematic records.

```
MWARN: DATA IN DEMO BUT NO MATCHING TRT INFO

Obs  screenno   inv_no   patid   brthdt    racecd   raceoth   trtgrpn  trtgrp  race

 1   4012         4      146    10/01/1971    1                .
```

The source code for the %mwarn macro is simple and presented below:

```
%macro mwarn(inset=problem,why=) ;

   /* CHECK IF ANY OBS IN PROBLEM DATA SET */
   data _null_ ;
      if 0 then set &inset nobs=numobs ;
      call symput("nnnn", left(put(numobs,5.))) ;
      stop ;
   run ;

   /* IF PROBLEMS EXIST… */
   %if &nnnn ne 0 %then
      %do ;
         /* PRINT PROBLEM RECORDS */
         proc print data=&inset ;
            title "MWARN: &why" ;
         run ;
         title;
         /* PUT WARNING AND ERROR MESSAGE IN LOG */
         %put WARNING: &why ;
         data _null_ ;
            error "WARNING: &why" ;
            abort ;
         run ;
      %end ;

%mend mwarn ;
```

## VALIDATE THE SUMMARY TABLES

In addition to permanently stored derived data sets, clinical programmers are often responsible for generating summary tables. When generating summary tables, one common method of validation is for two programmers (or one programmer and one statistician) to independently program the output. One programmer is responsible for creating the "production" output (nicely formatted tables) while the other is responsible for generating the summary statistics for validation purposes (output directly from the SAS PROC is usually sufficient). Once both programmers have generated summary statistics, the programmer responsible for validation then compares the program results manually (e.g. prints the nicely formatted table and checks the numbers against the validation output). If programs are only run once on final data, as may be the case with short studies, this process can be efficient. However, with clinical trials that take a year or more to complete, programs are often run several times on different versions of the database. In these cases, manually checking summary tables more than twice is tedious and very inefficient.

To help improve and speed up the table validation process, it is necessary to automate as much of the independent validation process as possible. While the automated process takes a little more time to implement up front, it can quickly ensure that the results of summary tables are always 100% validated each time the program is run. The larger the table, the more time is saved as compared to manually checking the results. Programmers still need to

manually review the final tables to ensure that data is presented properly, but the actual content is validated by SAS. One method to automate independent table validation is described here.

**PRODUCTION PROGRAMS PRODUCE DATA SETS**

The key step in automating the independent validation process is storing the data set immediately prior to the report-writing section of code (PROC REPORT, DATA _NULL_, etc.) as a permanent data set. The table production program needs to create a permanent SAS data set in a designated validation folder, which in this example will be called CHECKTAB. This data set should only contain the essential results that will be displayed in the table; no titles, footnotes, headers, or other supportive information should be included as the validation programmer will assume that all variables in the table validation data set need to be validated. Wherever possible, "pure" values should be included in the validation data set rather than "reporting" values so the validation programmer does not need to spend more time than necessary duplicating the data format. For example, when reporting counts and percentages, include the numeric variables CNT1 and PCT1 with values of 4 and 66.7 respectively, rather than CAT1 with the combined value "4 (66.7%)". There are some circumstances where combined character values should be included in the validation data set. One case where it may be advantageous to include the reporting variable is for confidence intervals that get displayed as "(xx.xx, yy.yy)". Including this variable will allow the validation programmer to automatically confirm that the variables are strung together correctly (i.e. low, high instead of high, low) rather than checking manually. A portion of the production program would look something like the code below.

```
… code creating summary statistics …

data final;
   set all;
   array cat {3} cat_1 - cat_3 ;
   array cnt {3} cnt_1 - cnt_3 ;
   array pct {3} $8 pct_1 - pct_3 ;
   do i = 1 to 3 ;
      cnt{i} = input(scan(cat{i}, 1, ' '), 8.) ;
      pct{i} = scan(cat{i}, 2, ' ') ;
      if cat{i} = '' then cat{i} = '0' ;
      if cnt{i} = .  then cnt{i} = 0 ;
   end ;
   stat = "n (%)" ;
run ;

proc print data = final(where = (&printme))  ;
   title "&progname: FINAL DATA" ;
run ;

**----- OUTPUT FINAL DATASET TO VALIDATION FOLDER -----**;

proc sql ;
   create table checktab.Table_1 (drop = cat_1 - cat_3) as
   select *
   from all
   order by prntord, statord;
quit ;

proc sort data = final;
   by page prntord statord ;
run ;

**------------------------------------------------------------------------------**;
**  CREATE TABLE                                                                **;
**------------------------------------------------------------------------------**;

**----- RTF SETUP -----**;
options nodate nonumber orientation=landscape nobyline;
ods listing close ;
ods escapechar='^' ;
ods rtf style=TStyleRTF file="C:\Table_1.rtf" ;
```

```
**----- REPORT TITLES -----**;
title1 justify = right 'Page ^{pageof}' ;
title2 "Table 1" ;
title3 “Summary of Adverse Events” ;

**----- REPORT DEFINITION -----**;
proc report data=final missing nowindows  split='|' ;
   by page ;
   column page prntord statord  statname stat cat_1 cat_2 cat_3 ;
   .
   .
   .
run ;

**----- CLOSE RTF AND RESET TITLES/FOOTNOTES -----**;
ods rtf close ;
```

This program produces permanent SAS data set named Table_1 that looks like this:

| PRNTORD | STATNAME | CNT_1 | CNT_2 | CNT_3 | PCT_1 | PCT_2 | PCT_3 | STAT |
|---|---|---|---|---|---|---|---|---|
| 1 | Total Number of Individual TEAEs | 66 | 77 | 143 | | | | n (%) |
| 2 | Subjects with at Least One TEAE | 3 | 3 | 6 | (100.0) | (100.0) | (100.0) | n (%) |
| 3 | Subjects with at Least One Serious TEAE | 3 | 3 | 6 | (100.0) | (100.0) | (100.0) | n (%) |
| 4 | Resulted in Death | 2 | 2 | 4 | (66.7) | (66.7) | (66.7) | n (%) |

**VALIDATION PROGRAMS COMPARE DATA**

The validation programmer still independently creates a SAS program to generate the summary statistics for the table being validated. Those summary statistics are now placed into a data set that matches the structure of the corresponding CHECKTAB data set and then PROC COMPARE is used to validate the results. It is important to note that this code will only validate values within the body of the table (the time-consuming part of validation). The validation programmer will still need to check manually for cosmetic (any misspellings in headings, footnotes) and layout issues (did the data get presented on the page correctly or are values truncated or running together?). It is critical that the results from the PROC COMPARE be absolutely clean and free from differences. Any differences must be resolved prior to declaring the output “validated.” This way, if the table is run with new data, any problems that result will be immediately obvious if the PROC COMPARE shows any discrepancy. Continuing the example started above, the final lines of the validation program would be similar to the code below.

```
proc compare base= checktab.Table_1 compare=valtab listall ;
   id prntord statname ;
run ;
```

Problems where the validation output does not match the production output would show in the results of the PROC COMPARE, as in the following partial output:

```
                    Value Comparison Results for Variables

_____________________________________________________________________________

                                ||       Base    Compare
   PRNTORD   STATNAME           ||      CNT_1      CNT_1      Diff.     % Diff
   _______   ________________   ||  _________  _________  _________  _________
                                ||
         4   Resulted in Death   ||     2.0000     6.0000     4.0000   200.0000
```

Prior to being declared “validated” all PROC COMPARE output should show that the number of observations in each data set are equal and declare that “No unequal values were found.”

```
                         Variables Summary

                    Number of Variables in Common: 9.
                    Number of ID Variables: 2.
```

```
                    Observation Summary


        Number of Observations in Common: 4.
        Total Number of Observations Read from CHECKTAB.Table_1: 4.
        Total Number of Observations Read from WORK.FINTAB: 4.

        Number of Observations with Some Compared Variables Unequal: 0.
        Number of Observations with All Compared Variables Equal: 4.

        NOTE: No unequal values were found. All values compared are exactly equal.
```

## CONCLUSION

While validation can often seem burdensome, it is the only way to ensure that mistakes in program output are minimal. With the proper preparation, many SAS programs can perform the bulk of the validation task automatically. When manual review of program output is minimized, accuracy and efficiency can both be achieved. It is important that the manual component not be forgotten, but rather applied strategically to ensure that all SAS program output is correct and makes sense within the context of the project. The methods described above have been used in practical situations and have been found to increase efficiency and reduce error and we sincerely hope that they can do the same for you.

## CONTACT INFORMATION

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

Carol Matthews
Ginger Lewis
Olga Belotserkovsky
United BioSource Corporation
17 Blacksmith Road
Newtown, PA 18940
Work Phone: 215.321.9876
Email: carol.matthews@unitedbiosource.com
ginger.lewis@unitedbiosource.com
olga.belotserkovsky@unitedbiosource.com
Web: www.unitedbiosource.com