PharmaSUG 2013 - Paper TF24

Defensive Programming and Error-handling: The Path Less Travelled

Tracy Sherman, InVentiv Health Clinical, Cary, NC

Angela Ringelberg, InVentiv Health Clinical, Cary, NC

ABSTRACT

Defensive programming detects logic errors by determining whether the code represents a proper implementation of the specifications. Thus, defensive programming is a prospective approach that anticipates logic errors, even those not yet discovered. Once a logic error is identified, messages are written to the SAS log and, possibly, additional diagnostics commences. The process emulates a thought experiment that reduces logic errors. This paper explains defensive programming and error-handling techniques based on the implementation of CDISC ADaM (analysis) data sets, which can be easily extended to other programming tasks.

1.0 INTRODUCTION

In the pharmaceutical industry a clinical trial determines the safety and efficacy of a treatment. Thus, it is imperative to ensure integrity of the data and analysis. Similar to Good Clinical Practice (GCP), an inherent part of any clinical trial, SAS professionals must employ good programming practices (Nelson and Zhou 2011), such as defensive programming and error-handling techniques.

Defensive programming is a method of writing robust SAS programs that respond to unexpected circumstances caused by the algorithm or the data. The issue is not about syntax errors, which do not threaten the integrity of the analysis. Besides, the SAS System identifies such errors easily. Instead the goal concerns the proper implementation of the specifications that includes an anticipation of possible flaws in the algorithm and a mechanism of handling such errors. Moreover, this technique provides feedback for the programmer / developer.

This paper provides several scenarios showing defensive programming and error handling used in creating the ADaM data sets: ADSL, ADAE, ADTTE, ADCM, and ADLB. The error handling output includes warning messages found in the SAS log and actual records written to Excel.

2.0 ADSL (SUBJECT-LEVEL ANALYSIS DATASET)

2.1 Subjects excluded from analysis sets

Consider the situation where you have been asked why there are so many subjects that have been excluded from the Per Protocol analysis set. You respond: “I’ll check into that and get back to you by tomorrow.” A better response would be to scroll through the SAS log for the program that created the ADSL data set and locate the user-defined log message that identifies such anticipated problems. The message is generated easily, as shown in the Data step below. Notice that the PUT statement splits the word “WARNING” into two words to prevent the word from being detected in the log file.

```
data pprot;
   set adsl;
   if pprotfl ^in ('','Y') and ppreas ne ''
      then put 'WAR' 'NING: ' subjid= 'is not in PP because ' ppreas= ;
run;
```

This code will produce the following warning message in the log file:

```
WARNING: SUBJID=01042001 is not in PP because PPREAS=Missing Tumor Assessment
WARNING: SUBJID=03002002 is not in PP because PPREAS=Erlotinib Compliance,Missing Tumor Assessment
```

Output 2.1 User-defined log message for subjects not in the Per Protocol analysis set

So in this case, error-handling feeds back to potential derivation errors and/or misinterpretation of data set specifications. If the Per Protocol analysis set flag, PPROTFL, is derived in the SDTM DM domain, then you will need to feed this back to the SDTM programmer for the study.

2.2 Baseline derivation

Generally, baseline readings are defined as the last non-missing value before the first dose of the study medication. For example, consider the baseline values for height and weight, HEIGHTBL and WEIGHTBL, respectively, which are commonly added to the ADSL data set. The derivation for these baseline readings seem straight-forward; yet, these variables often contain missing values. To prevent these logic errors, the following Data step identifies those subjects who have a null value for the baseline reading.

```
data vital;
  set vs;
    by subjid vstestcd vsdt;
 if vstestcd=’Height’ and vsstresn ne . and vsdt<fdosdt
    then do;
       if last.subjid;
       HEIGHTBL=vsstresn;
     end;

 if HEIGHTBL = .
    then put ‘WAR’ ‘NING: HEIGHTBL is missing. Check baseline derivation ‘
       subjid= vsstresn= vsdt= fdosdt= saffl=;
run;
```

```
WARNING: HEIGHTBL is missing. Check baseline derivation ‘ SUBJID=U00022741304
vsstresn=65 vsdt=09Mar2013 fdosdt= saffl=N
```

Output 2.2 Error-handling message sent to the log file for checking logic error in the baseline derivation

It turns out that the subjects having the aforementioned issue did not receive any study medication and were not part of the Safety population; however, these subjects are found in the Randomized analysis data set. Thus, to produce a correct demography table, the Data step must be modified, as follows:

```
data vital;
 set vs;
 by subjid vstestcd vsdt;
 if vstestcd=’Height’ and vsstresn ne . and ((vsdt<fdosdt) or fdosdt=.) then do;
    if last.subjid;
    HEIGHTBL=vsstresn;
 end;
run;
```

2.3 Additional ADSL logic checks and error-handling

Depending on the study, there are numerous opportunities to apply defensive coding and error-handling techniques. More importantly, especially for the ADSL data set, if these logic errors are not identified and resolved, they could propagate throughout the other data sets. Table 2.1 lists several examples for checking for possible issues, including: missing or incorrect values for treatment groups, duplicate records, incorrect enrollment assignment, and even alerting the programmer of the blinded status of the trial.

Logic check	Defensive code/Error-handling log message
Each subject should have a treatment arm assigned except those subjects that were screen failures	`if treatmentarm = ' ' and ARMCD not in ('SCRNFAIL') then put 'WARN' 'ING: Missing treatment assignment '` `usubjid= armcd= arm= TreatmentArm= Randno=;`

Logic check	**Defensive code/Error-handling log message**

DM domain should contain unique subjects only	`data adsl;`
	`  set dm;`
	`  by usubjid;`
	`  if first.usubjid and not last.usubjid then put`
	`  'WAR' 'NING: PLEASE CHECK THERE ARE DUPLICATES IN THE DM`
	`DATASET' USUBJID=;`
	`run;`

Study should only enroll Stage IIIB and IV cancer patients	`data mh;`
	`  merge mh1(in=mh) suppmh1 ;`
	`  by usubjid mhseq;`
	`  if mh;`
	`  if stagese='IIIB' then stagesen=0;`
	`    else if stagese='IV' then stagesen=1;`
	`      else put 'WAR' 'NING: PLEASE CHECK WHY WE HAVE STAGE`
	`OTHER THAN IIIB AND IV' usubjid=stagese=;`
	
	`  if first.usubjid and not last.usubjid then put 'WAR' 'NING:`
	`CHECK AS THERE ARE TWO PRIMARY CANCER STAGE HISTORY'`
	`USUBJID=;`
	`   run;`

Check and alert programmers of the blinded status of the trial. STATUS ne 'FINAL'; is set in your autoexec file.	`%let war=WAR;`
	`%let ning=NING;`
	
	`%if STATUS ne 'FINAL' then %put &war.&ning.: dummy variable`
	`for planned treatment, TRTP;`

Table 2.1 Additional ADSL logic checks and error-handling

3.0 ADAE (ADVERSE EVENTS ANALYSIS DATASET)

3.1 Date conversions with format modifier "??"

Taking this simple adverse event date conversion example, you often see the format modifier '??' added in an input statement to suppress printing _ERROR_ messages and the input lines when SAS encounters invalid data values.

```
data one
 set two;
 aesdt = input(aesdtc, ?? date9.);
run;
```

Unfortunately, by adding this format modifier, we are hiding potential data issues such as missing or impartial dates in the adverse event data. In the example below, we can add error-handling code to ensure feedback of the errant dates are printed to the log so these can be properly handled and/or fed back to the data management group for data cleaning.

```
data one
 set two;
 aesdt = input(aesdtc, ?? date9.);
 if length(aesdtc)<10 then put 'WAR''NING: check AESDTC ' subjid= aesdtc= aesdt=;
run;
```

3.2 Treatment-emergent Adverse Events

Hypothetically, in the statistical analysis plan for your study, treatment-emergent adverse events (AE) are defined as any event that occurs after first dose and less than 30 days after the last dose of study drug. The protocol and the data management plan state that AEs will not be captured after 30 days after the last drug has been administered. This appears to be a straight forward derivation but in practice it can be quite challenging because of missing/partial dates, dirty data and incorrect assumptions. To overcome these issues, defensive coding and some error-handling can combine to directly fix the issue or to identify and log the information required for the programmer or data manager to assess the issue. In the example below, code is added to check for AEs that occurred greater than 30 days after last dose (TRTENDT) and whether there are missing or partial dates.

```
data adae;
 set ae;
 if aesdtc ne '' and length(aesdtc)<7 then put 'WARN' 'ING: AESDTC is partial '
subjid= aesdtc= aesdt= aeterm= aeseq=;
 if aesdt>trtendt+30 then put 'WARN' 'ING: AESDT >30 days after study drug admin '
usubjid= aesdtc= trtendt= aeterm= aeseq=;
run;
```

You add the code above, check the log file and find that AEs were recorded more than 30 days after the last dose of study drug:

```
WARNING: AESDT >30 days after study drug admin usubjid=U00022741304
aesdt=10JUL2012 trtendt=03MAY2012 aeterm=DECUBITUS ON COCCYX aeseq=3
```

Output 3.1 Error-handling message sent to the log file

As a result, you need to update the specifications for treatment-emergent AEs to prevent this AE from being captured as treatment-emergent and further relay the information back to data management and clinical so they can prevent AEs from being recorded more than 30 days after last study drug administration.

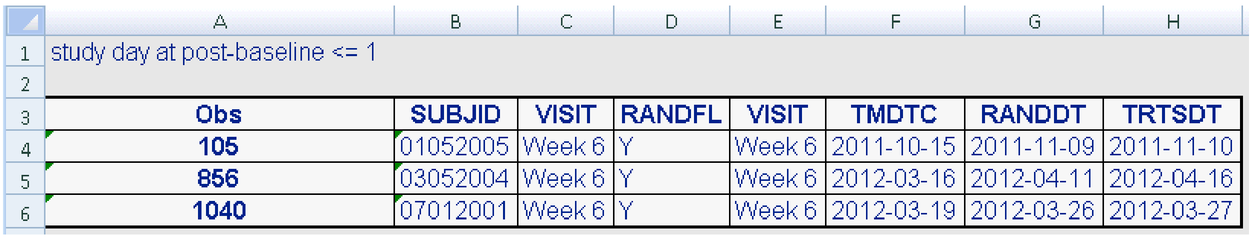
4.0 ADTTE (ANALYSIS DATASET TIME TO EVENT)

4.1 User generated data issue spreadsheet

Assume for a given clinical study that all tumor assessments are done post-baseline. However, upon implementing some defensive code, it becomes evident that there are issues with the data. In this case, the PRINT procedure along with a WHERE clause checked for any records prior to first dose, as shown below.

```
ods html close;
ods html file="tm_prob.xls" HEADTEXT= "<STYLE> TD {MSO-NUMBER-FORMAT:\@}</STYLE>";

proc print data=adtte (where=(ady<=1 and visitnum>0));
       var usubjid visit randfl visit tmdtc randdt trtsdt;
       title 'study day at post-baseline  <= 1';
run;

ods html close;
ods listing;
```

The error handling component of the proposed method could be as simple as writing the bogus records to a spreadsheet using ODS, which would be forwarded to Data Management. In PC SAS, the XLS output would be shown in the Results window. Otherwise, as specified in the code above, the spreadsheet would open in an HTML window, which can be saved as an Excel spreadsheet, as illustrated in Display 4.1.



study day at post-baseline <= 1

Obs	SUBJID	VISIT	RANDFL	VISIT	TMDTC	RANDDT	TRTSDT
105	01052005	Week 6	Y	Week 6	2011-10-15	2011-11-09	2011-11-10
856	03052004	Week 6	Y	Week 6	2012-03-16	2012-04-11	2012-04-16
1040	07012001	Week 6	Y	Week 6	2012-03-19	2012-03-26	2012-03-27

Display 4.1 Error-handling spreadsheet to be sent to data management

5.0 ADCM (CONCOMITANT MEDICATIONS ANALYSIS DATASET)

5.1 MedDRA coding completed/special interest concomitant medications

Assume that a flag variable was created that identifies subject taking specific concomitant medications, such as: Tylenol, paracetamol, or acetaminophen, at different stages of data processing, including data entry, cleaning, and coding. Defensive code could be implemented to ensure that all possible combinations of medications were captured. Using the FIND function accomplishes the task. Notice the 'i' option in the FIND function, which ignores case sensitivity; otherwise, the function would look only for those characters specified in the argument of the function.

```
data tyl;
   set adcm;
   if FIND(cmdecod,'paracetamol','i')
      then ANL01FL='Y';
      else if FIND(cmtrt,'tylenol','i')
         then ANL01FL='Y';
         else if FIND(cmtrt,'acetaminophen','i') then ANL01FL='Y';
run;
```

Error-handling code to check if the data is completely coded should be added here to tell the programmer the status of the coding and to print a list of these uncoded medications. A warning message can be printed to the log file with:

```
    if anl01fl='Y' and cmtrt ne '' and cmdecod='' then put 'WAR' 'NING: med is not
coded. Check for tylenol and associated meds ' subjid= cmtrt=;
```

```
WARNING: data is not completely coded. Check for tylenol and associated meds
SUBJID=U00022741304 CMTRT=Lidocaine
```

Output 5.1 Error-handling message sent to the log file

6.0 ADLB (ANALYSIS DATASET LABORATORY RESULTS)

6.1 Study Day

You receive SDTM data from a third party and notice something doesn't look right with the study day derivation. You perform a quick data reliability check and output a message in the log to alert programmers:

```
 data adlb;
  set adlb;
  if nmiss(trtsdt,lbdt)=0 then ady=lbdt-trtsdt+(lbdt>=trtsdt);
  if lbdy ne ady then put 'WAR' 'ING: CHECK SDTM LBDY DERIVATION '
  subjid= lbseq= lbdt= trtsdt=;
run;
```

```
894   data adlb;
895    set adlb;
896    if nmiss(trtsdt,lbdt)=0 then ady=lbdt-trtsdt+(lbdt>=trtsdt);
897    if lbdy ne ady then put 'WAR' 'ING: CHECK SDTM LBDY DERIVATION ' subjid= lbseq= lbdt=
trtsdt=;
898   run;
WARING: CHECK SDTM LBDY DERIVATION SUBJID=01001001 LBSEQ=1 LBDT=30SEP2010 TRTSDT=2010-10-08
WARING: CHECK SDTM LBDY DERIVATION SUBJID=01001001 LBSEQ=5 LBDT=13OCT2010 TRTSDT=2010-10-08
WARING: CHECK SDTM LBDY DERIVATION SUBJID=01001001 LBSEQ=9 LBDT=27OCT2010 TRTSDT=2010-10-08
NOTE: There were 64644 observations read from the data set WORK.ADLB.
NOTE: The data set WORK.ADLB has 64644 observations and 72 variables.
NOTE: DATA statement used (Total process time):
      real time           0.54 seconds
      cpu time            0.37 seconds
```

Output 6.1 Output from the log

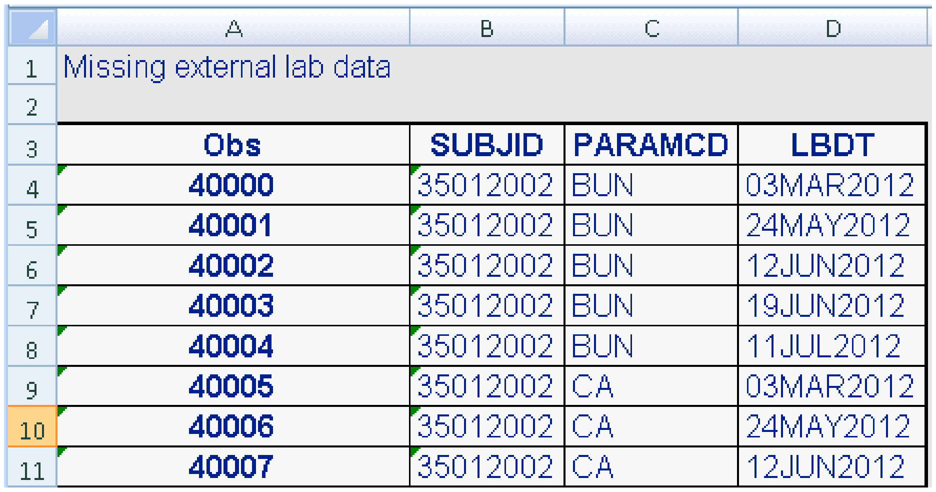
6.2 Reconcile CRF data to third party lab data

Consider that there is a need to reconcile third party lab data with data from the Case Report Form (CRF). Often, there is a mismatch between the two sources. Defensive programming can be applied to validate the data transfer by comparing the two sources. The Data step below accomplishes the task by using a match merge on the two source data sets.

```
data crfext;
  merge crflb (in=incrf)
        ext   (in=inext);
  by usubjid paramcd lbdt;

  if incrf and not inext then do;
     put 'WAR' 'ING: Missing external lab data '
     subjid= paramcd= aval= lbdt=;
     extmiss=1;
  end;
run;
```

Similar to example 4.1 above, when there are extensive discrepancies, it can be useful to provide a spreadsheet to the party responsible for data cleaning. ODS HTML along with PROC PRINT is used again to output an Excel spreadsheet to the SAS Results window (Display 6.2).

```
ods html close;
ods html file="lb_prob.xls" HEADTEXT= "<STYLE> TD {MSO-NUMBER-FORMAT:\@}</STYLE>";

proc print data=crfext (where=(extmiss=1));
       var subjid paramcd lbdt;
       title 'Missing external lab data';
run;

ods html close;
ods listing;
```



Missing external lab data

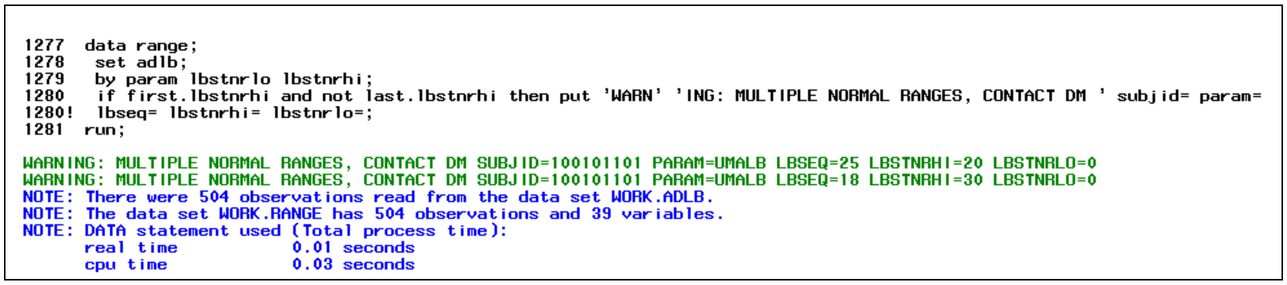
Obs	SUBJID	PARAMCD	LBDT
40000	35012002	BUN	03MAR2012
40001	35012002	BUN	24MAY2012
40002	35012002	BUN	12JUN2012
40003	35012002	BUN	19JUN2012
40004	35012002	BUN	11JUL2012
40005	35012002	CA	03MAR2012
40006	35012002	CA	24MAY2012
40007	35012002	CA	12JUN2012

Display 6.2. Error-handling spreadsheet of missing external lab data

6.3 Multiple normal ranges for same lab test

Suppose you’re writing a program to produce an out of range summary table. You notice that a lab parameter has more than one normal range for the same Gender / Age group. You need to anticipate or recognize situations like this in order to write additional code that will generate an appropriate warning message. In this case, by-group processing is used along with the FIRST/LAST operators, as show below.

```
data range;
 set adlb;
 by param lbstnrlo lbstnrhi;
 if first.lbstnrhi and not last.lbstnrhi then put 'WARN' 'ING: MULTIPLE NORMAL RANGES,
CONTACT DM ' subjid= param= lbseq= lbstnrhi= lbstnrlo=;
run;
```



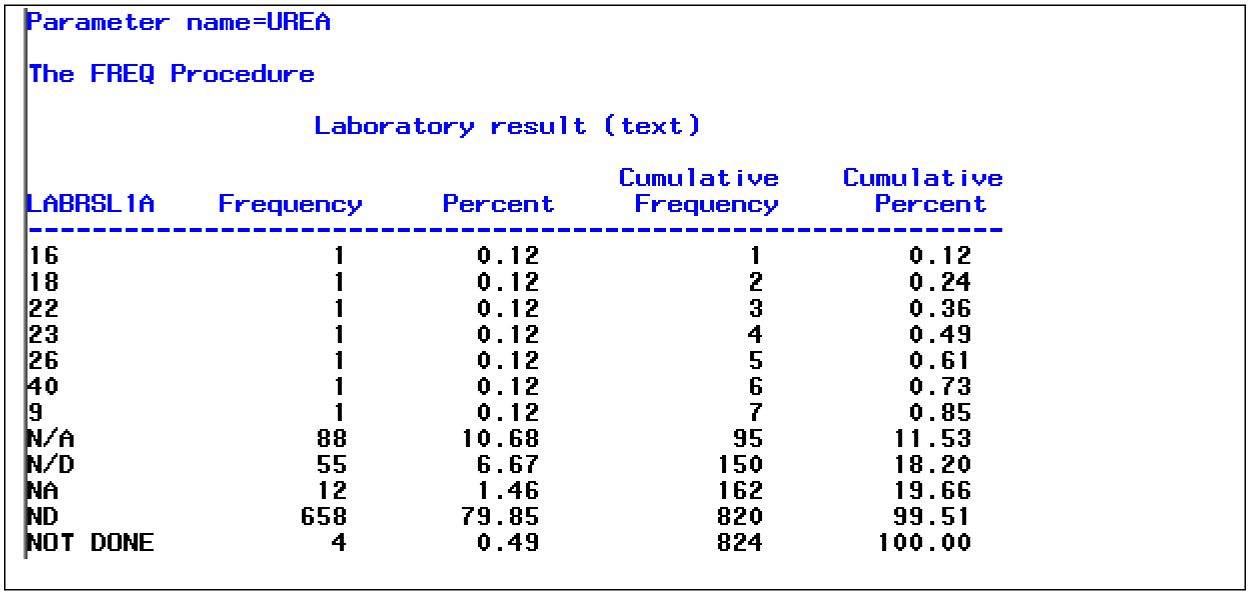
```
data range;
 set adlb;
 by param lbstnrlo lbstnrhi;
  if first.lbstnrhi and not last.lbstnrhi then put 'WARN' 'ING: MULTIPLE NORMAL RANGES, CONTACT DM ' subjid= param=
 lbseq= lbstnrhi= lbstnrlo=;
run;

WARNING: MULTIPLE NORMAL RANGES, CONTACT DM SUBJID=100101101 PARAM=UMALB LBSEQ=25 LBSTNRHI=20 LBSTNRLO=0
WARNING: MULTIPLE NORMAL RANGES, CONTACT DM SUBJID=100101101 PARAM=UMALB LBSEQ=18 LBSTNRHI=30 LBSTNRLO=0
NOTE: There were 504 observations read from the data set WORK.ADLB.
NOTE: The data set WORK.RANGE has 504 observations and 39 variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.03 seconds
```

Output 6.3. Output from log file

6.4 The power of the PROC FREQ

PROC FREQ is a popular procedure that can be used also for defensive programming. Obviously, this procedure facilitates the process of getting to know the data, that is, the frequency of values as well as revealing possible outliers for a given variable. It can be a powerful tool when producing ADaM data sets, specifically: ADLB, ADVS, ADAE, and ADEG. For example, it may be important to identify the values for a specific lab parameter because it becomes apparent that there are many missing values (See Output 6.4). The code below generates a report that corroborates our expectation that the table for UREA will be sparsely populated.

```
PROC FREQ data=data_s.lrs (where=(parnam1c='UREA')) ;
   table labrsl1a / out=temp ;
   by parnam1c ;
run;
```

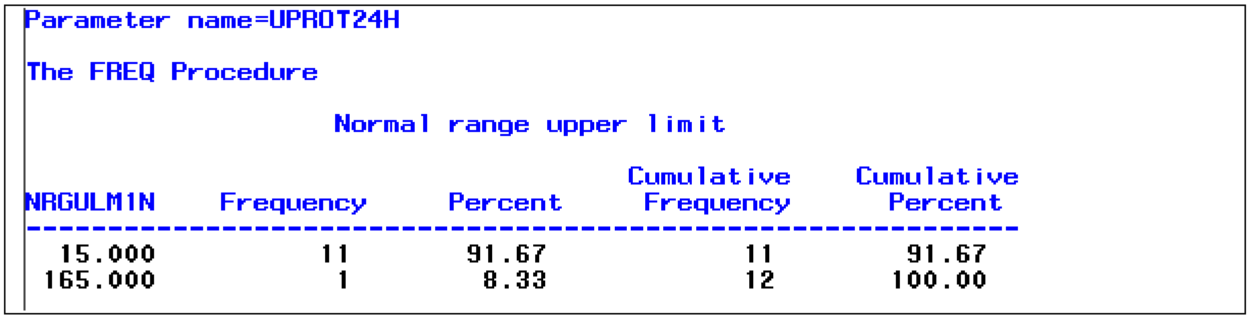


```
Parameter name=UREA

The FREQ Procedure

                     Laboratory result (text)

                                           Cumulative    Cumulative
LABRSL1A    Frequency      Percent         Frequency       Percent
-----------------------------------------------------------------
16                  1         0.12                 1          0.12
18                  1         0.12                 2          0.24
22                  1         0.12                 3          0.36
23                  1         0.12                 4          0.49
26                  1         0.12                 5          0.61
40                  1         0.12                 6          0.73
9                   1         0.12                 7          0.85
N/A                88        10.68                95         11.53
N/D                55         6.67               150         18.20
NA                 12         1.46               162         19.66
ND                658        79.85               820         99.51
NOT DONE            4         0.49               824        100.00
```

Output 6.4.1 Output from PROC FREQ

Use PROC FREQ to locate outliers or possible data issues. The following example is parameter and subject specific. As shown in PROC FREQ output (Output 6.4.2), the value of '165' for the upper normal range is definitely an outlier for this subject and one that should be questioned to data management.

```
PROC FREQ data=data_s.lrs (where=(parnam1c='UPROT24H' and sid1a='1009_00001')) ;
  table nrgulm1n / out=temp ;
  by parnam1c ;
run;
```



```
Parameter name=UPROT24H

The FREQ Procedure

                       Normal range upper limit

                                                    Cumulative     Cumulative
NRGULM1N      Frequency        Percent              Frequency        Percent
-------------------------------------------------------------------------------
  15.000             11          91.67                     11          91.67
 165.000              1           8.33                     12         100.00
```

Output 6.4.2 PROC FREQ

7.0 CONCLUSION

Defensive programming and error-handling should be common practice for clinical programming. Anticipating data issues and logic errors can be time-consuming at first since the specifications are sometimes ad hoc, even vague. However, making that upfront investment of time will save far more time as the project proceeds, Also, needless to say, these techniques will prove to be invaluable at 'crunch time' when the timelines are tight.   Most importantly, however, these techniques ensure the integrity of the deliverables by checking derivations and data issue during different stages of the data process. This paper presented several examples of defensive programming and error-handling specifically for the production of CDISC ADaM data sets.

8.0 REFERENCES

Nelson, Gregory S. and Zhou, Jay. 2011."Good Programming Practices in Healthcare: Creating Robust Programs". Proceedings of the 2011 Pharmaceutical Industry SAS Users Group (PharmaSUG) Annual Conference. Available at http://www.lexjansen.com/pharmasug/2011/tt/pharmasug-2011-tt05.pdf.

ACKNOWLEDGMENTS

We would like to say a special thanks to InVentiv Health Clinical and Matt Becker for encouraging and supporting conference attendance. Further thanks to John R. Gerlach for his insight and editorial review of this paper.

CONTACT INFORMATION

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

Name: Tracy Sherman
Address: 1000 Winstead Drive, Suite 200
City, State ZIP: Cary, NC 27513
Work Phone: 609-580-8217
E-mail: tracy.sherman@inventivhealth.com

Name: Angela Ringelberg
Address: 1000 Winstead Drive, Suite 200
City, State ZIP: Cary, NC 27513
Work Phone: 609-580-8285
E-mail: angela.ringelberg@inventivhealth.com