

THE ART OF TESTING

J. Meimei Ma
Quintiles

INTRODUCTION

This tutorial introduces general testing strategies related to the problem of being convinced that a program is ready for use. The focus is on data management programming related to statistical analysis, as opposed to creation of an interactive application. Although testing should proceed logically, the process also requires imagination and has common pitfalls. Large file processing is emphasized since potential losses are greater if an inappropriately tested program becomes a production version. Examples from both mainframe and microcomputer environments are provided.

The methods discussed are directed toward programmers, data managers, project managers, and anyone who does not like, or cannot afford, surprises after a production run. Only basic knowledge of DATA step programming is assumed. Testing is presented as an important aspect of programming development.

LARGE FILE CHARACTERISTICS

Before discussing special problems associated with large file processing, we must first attempt to define *large*. A basic characteristic of working with large files is that testing thoroughly is always worthwhile. This applies equally to raw files or SAS datasets. The actual number of observations and/or variables that should trigger the use of large file processing techniques depends on the particular computing environment involved. A large file may be *short and wide*, which means having relatively few observations but a large number of variables, or *long and narrow*, in which only a few variables exist for many observations.

In a mainframe computing environment a large file may have as few as ten thousand records or more than one million. Batch processing, as opposed to interactive processing, is more common for large mainframe files. In certain environments, overnight processing is used to reduce computing costs. For very large files, tape storage becomes more common.

For computing environments using microcomputers or minicomputers, the definition of *large* clearly depends on

the computer system's capacity. Large file processing on a micro may be described as "walk away" processing, i.e., you submit a program and walk away for a well deserved break while your micro keeps working. Programming in a microcomputer environment is generally interactive, but batch processing can be appropriate for extreme situations.

Regardless of the computing environment, processing any large file is relatively expensive in terms of programmer time and/or computing dollars. Any attempted *production run*, which must process an entire file, becomes a significant event. Put another way, discovering a mistake after processing a large file makes most people very unhappy. Finding problems in programs as early as possible by thorough and efficient testing becomes more important in such a situation. When machine efficiency significantly effects processing time and cost, the extra programming time spent testing code pays off directly in comparison to the possibility of a wasted production run. In addition, the indirect advantages of reduced frustration when testing pays off should not be underestimated.

TESTING STRATEGIES

The testing strategies described here range from broad generalizations to specific coding tricks and pitfalls. The most important idea is to always, always, run a test program before a production run. Then keep testing until you are convinced you can afford to stop. (For a more scholarly approach to deciding when testing should stop, see Dalal and Mallows (1988).) A *test run* is simply any execution of a program that cannot be kept as a production run. Effective testing reduces the likelihood that a production run becomes an expensive test run.

Process Alternatives

The overall procedure used for program development can influence testing effectiveness. Incremental or structured program development, in which each section of code is tested before being added to the overall program, is the recommended approach. (Using the macro language, which can help reduce testing time, is not addressed directly in this tutorial.) However, even a program with fully tested parts should still be retested as a whole.

During development, compare alternative approaches with appropriately chosen test data instead of using the actual large file. Regardless of the exact testing technique, time and money must be allocated for test runs.

Intermediate datasets are useful during testing and can provide other benefits as well. Consider storing a subset permanently, even one only required for a quick analysis. Checking the results with a permanent subset is simpler since another check can be run without changing the production program. In other cases, simply including a restricted PRINT procedure for a temporary work dataset is sufficient. When designing more sophisticated output, with the TABULATE procedure for example, the use of stored summary statistics makes testing much easier (Fischell and Hamilton 1987).

Imaginative Test Data

The SAS system provides many easy ways of creating test datasets in order to generate effective and inexpensive early warnings about programming errors. A test run based on the system OBS option may be sufficient for less complex programs. Testing with a small number of records is better than OBS=0 since both syntax and logic problems may then be detected. Note that using the OBS option is not appropriate for programs that use WHERE statements. Permanent test datasets may be appropriate for large databases used repeatedly for analyses. The *SAS Applications Guide* contains examples of creating probability or systematic samples.

For more complex or specialized programs consider creating a *fudged* dataset with observations that are guaranteed to trigger execution of all sections of code. Carefully designed fudged data assures that the processing of special cases, which are relatively rare in a large dataset, is fully tested. Example 1 demonstrates data fudging. Remember that your program must work correctly for situations that occur less than ten percent of the time as well as for the relatively normal ninety percent. Use all available information about the data being processed in the construction of appropriate test data. Ask for advice from other people involved in the project, such as statisticians or data processing staff. The objective is to create test datasets that focus on the important issues, not necessarily every possible situation.

Easy Coding Tips

The SAS system includes functions and procedures that simplify testing. The OBS option, both the system option

and the dataset option, is shown in Example 1 and Example 2. When developing a DATA step, selective printing of all variables is easily accomplished. For instance, the statement

```
IF _N_ < 10 THEN PUT '***' IDNUM= / _ALL_ ;
```

prints a clear listing of all variables for the first nine records; temporary variables, such as FIRST. and LAST., are included. By using an appropriate condition, special cases are quickly checked regardless of KEEP or DROP statements applied to the output dataset. Another method for verifying calculations based on variables that are not kept is to change a KEEP statement to a comment (*KEEP ... ;) for test runs.

Procedures such as CONTENTS and PRINT are ideal for checking DATA step results. A good habit is to include code similar to the following after any significant DATA step:

```
PROC CONTENTS POSITION;  
TITLE 'description';  
PROC PRINT DATA=lib.sasname (OBS=nn);
```

The next trick is to remember that including these procedures is only helpful if the programmer reviews output carefully. Depending on the type of variables involved, consider checking variable values with other procedures such as FREQ or UNIVARIATE.

Ideally, statements used for testing should be easy to identify, and change, or should not require any changes before a production run. Restricting a PRINT procedure with the dataset OBS option eliminates the risk of a production run attempting to list an entire large dataset. When test statements must be removed, change them to comments in the production run. Having these "test comments" provides useful documentation for future reference.

Pitfalls

While you dutifully test your program, keep potential problems in mind. Avoid the temptation of thinking that if OBS=1000 is good, then OBS=5000 is better. In the long run the most efficient testing is based on having the largest variety of situations in the smallest possible test dataset. For those migrating from a mainframe batch environment to an interactive one, remember that using the system OBS option has a lasting effect. Just changing the OPTIONS statement to a comment for a production run is not enough. The system OBS setting must be changed back to the default (OBS=MAX). Forgetting this

is especially devastating if you are sorting a permanent dataset because you risk replacing the large dataset with a small sorted test dataset. (You do have a backup copy, don't you?) In any computing environment, consider if using the OUT option of the SORT procedure or sorting the permanent dataset beforehand in another program would not be safer and/or more efficient.

The hardest part of testing is keeping alert. The easiest trap is to only glance at the log and output, instead of reviewing test run results carefully, because a program, or the latest change, is "simple."

EXAMPLES

The examples demonstrate some of the testing techniques discussed. The first example applies more to a mainframe, batch processing environment while the second applies to an interactive environment.

Example 1

The objective of this program is to generate tables about young front seat passengers in car accidents. The large input dataset has data on motor vehicles involved in accidents in 1988; about 300,000 observations with 250 variables. A permanent analysis file is created by selecting observations—cars only—and variables for the analysis. The original analysis request related County with Age, Seat Belt Usage, and Injury Level of drivers and right front passengers. Figures 1a, 1b, and 1c show three test runs sequentially run to accomplish this task. Additional comments, such as "/* TEST ONLY */" are not included in the test statements to better simulate real programming. The final production run would differ from Figure 1c only in that the OPTIONS statement would be changed to a comment.

In the first program, a fudged dataset is created to guarantee the occurrence of "young" passengers. In car

Figure 1a. Example 1 - Test Run 1

```

OPTIONS OBS=100;

DATA outlib.CAR88(LABEL='subset description');
  SET inlib.ACC88;

IF _N_ < 20 THEN RFAGE=1;
ELSE IF _N_ < 40 THEN RFAGE=4;

  IF VEHTYPE=1 OR VEHTYPE=2; /* CARS ONLY */

  KEEP  COUNTY
        DRAGE DRINJ DRBELT DRSEX      /* DRIVER */
        RFAGE RFINJ RFBELT FRSEX;     /* RIGHT FRONT */

PROC CONTENTS;
TITLE 'description';
PROC PRINT DATA=outlib.CAR88(OBS=50);

```

accidents, the percentage of right front passengers less than six years old is so low that without fudging even OBS=25000 might be too small to be useful. Note how not indenting test statements (IF _N_ ...) makes them easier to find and remove later. In addition to the original analysis variables, DRSEX and RFSEX are included in anticipation of follow up analyses since a permanent subset is being created. The CONTENTS and PRINT procedures provide quick checks of the CAR88 dataset.

Once the first test run is successful, the next version is created to take the program farther (Figure 1b). After looking at the PRINT output from the first test, the programmer adds ACCNUM and VEHNUM to the KEEP statement. A check using the FREQ procedure also seems a useful addition since all the variables are categorical. By

specifying OBS=1000, the FREQ output will provide a better test/confirmation for the production run even though the test run is only for OBS=100. Now that the programmer is convinced the DATA step works, the requested analysis is added (FORMAT and FREQ).

The final test run, shown in Figure 1c, has just production run statements with an OBS option. The final test is necessary since a problem was found in the second test run and deleting test statements may create a syntax error. ACCNUM and VEHNUM are now dropped from the FREQ designed for checking only. A syntax error in the first step of a production run is just a minor irritation, but imagine situations in which the error is in the third of five steps or is discovered the next morning after using "midnight priority."

Figure 1b. Example 1 - Test Run 2

```

OPTIONS OBS=100;
DATA outlib.CAR88(LABEL='subset description');
  SET inlib.ACC88;

IF _N_ < 20 THEN RFAGE=1;
ELSE IF _N_ < 40 THEN RFAGE=4;

      IF VEHTYPE=1 OR VEHTYPE=2;                      /* CARS ONLY */

      KEEP  ACCNUM VEHNUM
            COUNTY
            DRAGE DRINJ DRBELT DRSEX                  /* DRIVER */
            RFAGE RFINJ RFBELT RFSEX;                  /* RIGHT FRONT */

PROC CONTENTS;
PROC PRINT DATA=outlib.CAR88(OBS=50);
PROC FREQ DATA=outlib.CAR88(OBS=1000);

PROC FORMAT;
  VALUE DRAGEF      15-35='15-35' 36-55='36-55' 55-97='55 +';
  VALUE RFAGEF      0-1='<2' 2-3='2-3' 4-5='4-5'
                    6-14='6-14' 15-97='15 +';

PROC FREQ DATA=outlib.CAR88;
  TABLES COUNTY * DRAGE * RFAGE;
  TABLES DRAGE * RFAGE * RFINJ;
  FORMAT DRAGE DRAGEF. RFAGE RFAGEF.;

```

Figure 1c. Example 1 - Test Run 3

```

OPTIONS OBS=100;
DATA outlib.CAR88(LABEL='subset description');
SET inlib.ACC88;

    IF VEHTYPE=1 OR VEHTYPE=2;                /* CARS ONLY */

    KEEP  ACCNUM VEHNUM
          COUNTY
          DRAGE DRINJ DRBELT DRSEX            /* DRIVER */
          RFAGE RFINJ RFBELT RFSEX;          /* RIGHT FRONT */

PROC CONTENTS;
PROC PRINT DATA=outlib.CAR88(OBS=50);
PROC FREQ DATA=outlib.CAR88(OBS=1000
                             DROP=(ACCNUM VEHNUM));

PROC FORMAT;
  VALUE DRAGEF    15-35='15-35' 36-55='36-55' 55-97='55 +';
  VALUE RFAGEF    0-1='<2' 2-3='2-3' 4-5='4-5'
                  6-14='6-14' 15-97='15 +';

PROC FREQ DATA=outlib.CAR88;
  TABLES COUNTY * DRAGE * RFAGE;
  TABLES DRAGE * RFAGE * RFINJ;
  FORMAT DRAGE DRAGEF. RFAGE RFAGEF.;

```

Figure 2. Input Format for R-Type Records

Original Format

Record Type	1	'R'
ID Code	3-9	CCNNNNN
Test Name	10-21	character
Result Type	22	character
Result	23-44	character
Alert Flag	45	character
Sequence	76-80	numeric

Revised Format

Record Type	1	'R'
ID Code	3-9	CCNNNNN
Test Name	10-21	character
Result Type	22	character
Result *	23-44	character
Alert Flag	45	character
Sequence	76-80	numeric

* Note: If Result Type=C then Result is in 23-75

Example 2

Example 2 is a set of programs that demonstrate an evolving process of developing a complex input statement when programming interactively on a microcomputer. Imagine that the raw input file has about 50,000 records and several different record formats distinguished by a character value in Column 1. The R-type records initially had a simple format that changed as shown in Figure 2. The information for RESULT became longer for certain records (RSLTTYPE='C'). The program statements are in Figures 3a-3d. Assume that all programs are run during one interactive SAS session. The OPTIONS statement would be changed to "OPTIONS OBS=MAX;" in Figure 2d for the production run.

The first program (Figure 3a) has documentation and efficiency features. Labels, of both datasets and variables, help clarify logic even for short programs. Getting debugging advice from another programmer is generally easier with documented, indented code that has blank lines. The trailing @ is efficient since only R-type records are converted to SAS. Specifying LENGTH=5 for the numeric variable saves storage space and processing time.

The second program (Figure 3b) is an attempt to adjust the input statement. Assume that the output from the first program is available. A careful comparison of the PRINT output shows that RESULT is now missing for all non-C Results. A further check is made of the C Results (Figure 3c) since the first forty records did not happen to include any of these special cases. No OBS options are used for LOOK because the programmer can always Ctrl-BREAK if the PRINT gets too long. Reviewing the LOOK dataset shows that RESULT is being read correctly, but SEQNUM is always missing. (The programmer is not surprised since the revised version of the input statement

was written in a hurry. However, he/she is glad to have tested thoroughly. This is opposed to being frustrated had the incorrect program been run on the full 50,000 records.)

The final program reads in all the variables correctly (Figure 3d). The check of the C Results is now incorporated into the program since it proved valuable. After reviewing the log and output from this program carefully, the programmer is ready to save the code, change the OBS option, submit the production run, and take a break while 50,000 records are correctly processed.

CONCLUSION

Testing is one of the most creative aspects of SAS programming. The more imagination you use in devising test runs and test datasets, the more likely you are to promptly discover problems. Every step, no matter how small or simple, should be tested deliberately. The examples presented here are just a starting point. Now it is up to you. Happy Testing!

The Main Objective

Effective testing
to avoid
wasted production runs

Figure 3a. Example 2 - Original Format Test

```
TITLE 'description';
OPTIONS OBS=500; /* CHANGE TO MAX */

DATA lib.RTYPE(LABEL='R TYPE RECORDS');
  INFILE raw;

  INPUT RECTYPE $1 @ ; /* ONLY R TYPE */
  IF RECTYPE='R'
    THEN DO;
      INPUT   IDCODE $3-9 NAME $10-21
             RSLTTYPE $22 RESULT $23-44
             ALERT $45 SEQNUM 76-80;
      LENGTH SEQNUM 4;
      OUTPUT;
    END;

PROC CONTENTS;
PROC PRINT DATA=lib.RTYPE(OBS=40);
RUN;
```

Figure 3b. Example 2 - Revised Format Test

```
TITLE 'description';
OPTIONS OBS=500; /* CHANGE TO MAX */

DATA lib.RTYPE(LABEL='R TYPE RECORDS');
  INFILE raw;

  INPUT RECTYPE $1 @ ; /* ONLY R TYPE */
  IF RECTYPE='R'
    THEN DO;
      INPUT   IDCODE $3-9 NAME $10-21
             RSLTTYPE $22 @;

      IF RSLTTYPE='C'
        THEN INPUT RESULT $23-75 @ ;
        ELSE INPUT ALERT $45
                 SEQNUM 76-80;

      LENGTH SEQNUM 4;
      OUTPUT;
    END;

RUN;

PROC CONTENTS;
PROC PRINT DATA=lib.RTYPE(OBS=40);
RUN;
```

Figure 3c. Example 2 - Check of C Results

```
DATA LOOK;
  SET lib.RTYPE;
  IF RSLTTYPE='C';
PROC PRINT;
RUN;
```

Figure 3d. Example 2 - Final Test Run

```
TITLE 'description';
OPTIONS OBS=500; /* CHANGE TO MAX */

DATA lib.RTYPE(LABEL='R TYPE RECORDS');
  INFILE raw;

  INPUT RECTYPE $1 @ ; /* ONLY R TYPE */
  IF RECTYPE='R'
    THEN DO;

      INPUT   IDCODE $3-9 NAME $10-21
             RSLTTYPE $22 @;

      IF RSLTTYPE='C'
        THEN INPUT   RESULT $23-75 @ ;
        ELSE INPUT   RESULT $23-44
                 ALERT $45 @;

      INPUT SEQNUM 76-80;

      LENGTH SEQNUM 4;
      OUTPUT;
    END;

RUN;

PROC CONTENTS;
PROC PRINT DATA=lib.RTYPE(OBS=40);
RUN;

DATA LOOK;
  SET lib.RTYPE;
  IF RSLTTYPE='C';
PROC PRINT;
RUN;
```

SAS PROCEDURES TO REMEMBER

The following SAS procedures are useful for data management or checking programs. Use appropriate procedures to test or verify programming results. Also consider operating system specific copy procedures, such as PDSCOPY, that simplify backup processes designed for self-protection during the testing phase.

CATALOG	FORMAT (FMTLIB option)
COMPARE	PDSCOPY
CONTENTS	PRINT
COPY	REPORT
DATASETS	SORT

REFERENCES

Dalal, S.R. and Mallows, C.L. (1988), "When Should One Stop Testing Software?" *Journal of the American Statistical Association*, Vol. 83, 872-879.

Frankel, D.S. and Kochanski, M.A. (1991), "A Debugging Facility for SAS Macro Systems," *Proc. of the Sixteenth Annual SAS Users Group Intl. Conference*, Cary, NC: SAS Institute Inc., 209-214.

Fischell, T.R. and Hamilton, E.G. (1987), "Processing Large Data Sets into Customized Tables," *Proc. of the Twelfth Annual SAS Users Group Intl. Conference*, Cary, NC: SAS Institute Inc., 978-983.

Helms, R. (1978), "An Overview of RDM: I. Project and Data Management System Planning," *American Statistical Association Proceedings of the Statistical Computing Section*, 10-17.

Howard, N., Pickle, L.W., and Pearson, J.B. Jr. (1987), "Effective Methods of Testing Using the SAS System," *Proc. of the Twelfth Annual SAS Users Group Intl. Conference*, Cary, NC: SAS Institute Inc., 400-407.

Kanthan, K.L. (1988), "Automated Test Generator for Software Testing," *Proc. of the Thirteenth Annual SAS Group Intl. Conference*, Cary, NC: SAS Institute Inc., 1380-1385.

Katsani, G. and Skelton, W.K. (1991), "The Balancing Act: An Approach to Analyzing Data Integrity During DATA Step Processing," *Proc. of the Sixteenth Annual SAS Users Group Intl. Conference*, Cary, NC: SAS Institute Inc., 531-535.

Ma, J.M. (1990), "Data Management Procedures for Large and Small Projects," *Proc. of the Thirteenth Annual SAS Group Intl. Conference*, Cary, NC: SAS Institute Inc., 163-168.

Mackiernan, Y.D. (1989), "Don't Do Anything You Don't Have To: Elementary Strategies For Processing Large Data Sets With SAS Software," *Proc. of the Fourteenth Annual SAS Group Intl. Conference*, Cary, NC: SAS Institute Inc., 178-186.

Minor, B. (1991), "Error: What To Look For When Debugging SAS Programs," *Proc. of the Sixteenth Annual SAS Users Group Intl. Conference*, Cary, NC: SAS Institute Inc., 1027-1032.

Muller, K.E., Smith, J., and Bass, J. (1982), "Managing 'not small' Datasets in a Research Environment," *Proc. of the Seventh Annual SAS Users Group Intl. Conference*, Cary, NC: SAS Institute Inc., 371-376.

Muller, K.E., Smith, J., and Christiansen, D.H. (1981), "Rules We Followed and Wish We Had Followed in Managing Datasets, Programs and Printouts," *Proc. of the Sixth Annual SAS Users Group Intl. Conference*, Cary, NC: SAS Institute Inc., 401-405.

SAS Institute Inc. (1987), *SAS Applications Guide, 1987 Edition*, Cary, NC: SAS Institute Inc.

SAS Institute Inc. (1990), *SAS Language: Reference, Version 6, First Edition*, Cary, NC: SAS Institute Inc. WHERE Statement, pp. 498-504.

SAS Institute Inc. (1990), *SAS Programming Tips: A Guide to Efficient SAS Processing*, Cary, NC: SAS Institute Inc.

SAS Institute Inc. (1989), *SAS Language and Procedures: Usage, Version 6, First Edition*, Cary, NC: SAS Institute Inc.

Schaffer, M. and Kalt, B. (1991), "The DATA Step Debugger: Bug Hunting Made Easy," *Proc. of the Sixteenth Annual SAS Users Group Intl. Conference*, Cary, NC: SAS Institute Inc., 1033-1035.

ACKNOWLEDGEMENTS

SAS is a registered trademark of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.