

Validation of SAS[®] Macro Systems

Philip M. Pochon, Computer Task Group, Inc., Indianapolis, IN
Thomas H. Burger, Source Consulting[®], Indianapolis, IN

Abstract

The validation of systems for regulated environments requires rigorous testing. System validation is generally achieved through structural testing ('white box') and functional testing ('black box'). SAS[®] macros permit construction of analysis and reporting systems using highly testable modular components. We describe a systematic method of validating macros which permits structural testing and (multilevel) functional testing while preserving regression testability.

Introduction

Formal software processes contribute to the overall adequacy of systems development (Humphrey 1989). As a distinct phase of the Software Development Life Cycle (SDLC), testing's purpose is to insure system reliability. Because the importance of software validation is particularly acute in regulated settings, the FDA has mandated Computer System Validation Policies (CSVP).

Formal testing typically occurs at three levels: unit, integration and system. The purpose of *unit* testing is to insure that individual functions, components, or modules execute only as intended. *Integration* testing insures that embedded applications interface properly. *System* testing verifies the readiness of an application for production implementation. *Validation* has traditionally referred to program execution in production environments; *design* testing occurs iteratively in development and/or test environments.

Because SAS[®] systems are frequently implemented as nonintegrated or one-off programs, testing may be *ad hoc* and cumbersome. Formalized methods of validation both ease testing and address compliance. Using a systems engineering approach, we describe a method of SAS[®] system *verification* and *validation* to ensure that:

- (sub)module requirements are met,
- development phases fulfill requirements of previous phase(s)
- (sub)modules comply with requirements.

This method permits system evaluation during each phase of the life cycle in *parallel* with development, rather than at its completion (Fujii and Wallace, 1997).

Problem Statement

A critical aspect of Software Quality is system reliability. Reliability is attained through validation of system components, which requires the ability to test function tasks; their testability rests in their composition.

System validation is typically done using two methods of testing. *Structural* testing ('white box') is a design-based method of testing control-flow at the unit level. *Functional* testing ('black box') is a requirements-based, data-driven method for testing data-flow through modules (Coward, 1997).

Sympatric test cases can be designed to address both aspects of reliability. Further care in their construction ensures vertical adherence to requirements (*validation*) and lateral precision across phases (*verification*). We present a method for applying these strategies in design testing SAS[®] macro systems.

Design

System design influences module testability and, therefore, has an impact on validation. Design also serves a number of validation roles outside actual code development. Design walk-throughs reveal bugs and disclose limitations in software requirements where they are least costly to fix. Designs also serve as Validation Deliverables for QA/QC and may be critical to support staff for maintenance.

Design has a direct bearing on program content and how code performs a task. Programs have reduced value if their design is unsound. System integrity and robustness rest in architecture because it defines the context in which code is executed.

The macro facility's ability to encapsulate native routines (PROCs, DATA step) as modules allows development of robust systems. Modules are logically grouped functions that perform a defined process. This feature can be leveraged through functional decomposition to compartmentalize tasks. This eases design, construction, and testing by isolating processing mechanics. Modularization is attained through hierarchical structuring, loose coupling, and discrete operations. Processing isolation is achieved through effective modularization.

Methods

Functional Hierarchy Diagrams (FHD) are used to model the physical design of modules. FHDs are horizontal decomposition charts showing the hierarchy and calling sequence of programs. It is useful to show conditional logic in diagrams to completely represent logic, as indicated by hatched blocks (Fig. 1). Shown is the organization of a mock driver that performs data preparation followed by analysis.

We impose hierarchy by building macros as *control* and *terminal* functions. **Drivers** control modules by calling secondary drivers or sub-drivers. **Sub-drivers** control sub-modules by calling secondary sub-drivers or utility functions. **Utilities** are terminal functions typically <20 lines of code that perform very discrete operations (Burger and Pochon 1997).

Decomposition limits redundancy by exposing consolidation and shared functionality. Modularization permits well-structured processing, limiting the risk of spaghetti coding. With few exceptions (conditionals, localization), embedding code in the body of control functions should be avoided; it reduces testability by decreasing processing isolation. An attractive feature of modularization is that components can be validated without disrupting the regression testability of unaffected modules.

Figure 1.
Functional Hierarchy
Diagram (FHD)

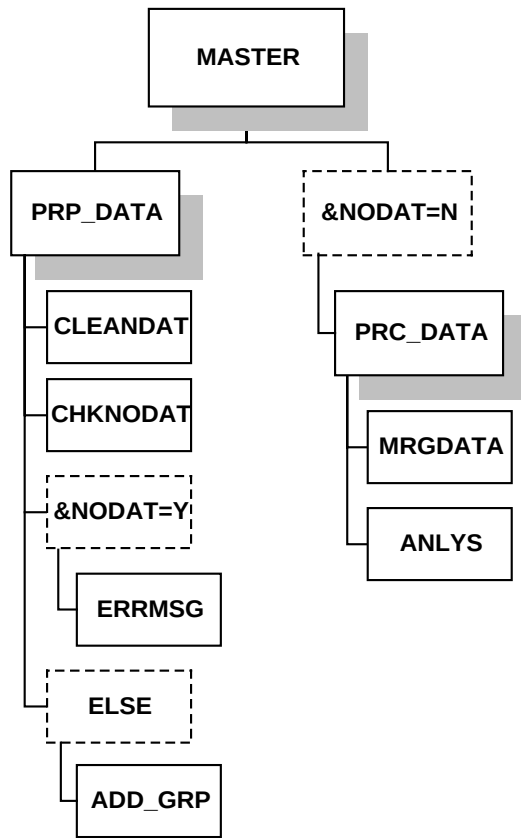


Figure 2. illustrates the program structure of the driver shown in the FHD. The macro function contains calls for two utilities followed by the conditional execution of two mutually exclusive functions. The calls for the first two functions are always executed. The third call is conditionally made by an %IF-%THEN-%ELSE clause based on the value of a macro variable (NODAT) set in CHKNODAT. When &NODAT=Y, the third function (ERRMSG) is called; when &NODAT=N the fourth function (ADD_GRP) is called.

Figure 2.

```

%MACRO PRP_DATA(_datadsn=WORK.RESULTS,
               _nulrslt=WORK.NULLDSN,
               _excrslt=WORK.EXCLDSN);

  %CLEANDAT(_indsn=&_datadsn,
           _nulldsn=&_nulrslt,
           _excldsn=&_excrslt)

  %CHKNODAT(_indsn=&_datadsn)

  %IF (&NODAT=Y) %THEN %DO;
    %ERRMSG
  %END;

  %ELSE %THEN %DO;

```

```

    %ADD_GRP(_indsn=&_datadsn,
            _chkcol=day,
            _colname=GROUP)

  %END;

%MEND PRP_DATA;

```

Figures 3, 4, and 5 show the program structure of the utilities called in the driver. The CLEANDAT utility (Fig. 3) removes NULL and EXCLUDED results from the input data set.

Figure 3.

```

%MACRO CLEANDAT(_indsn=WORK.INDATA,
               _nulldsn=WORK.NULLDSN,
               _excldsn=WORK.EXCLDSN);

  %LET ERRFLAG=N;

  DATA &_datadsn &_nulldsn &_excldsn;
  SET &_datadsn;
  IF RSLTVAL=. THEN
    OUTPUT &_nulldsn;
  ELSE IF QCCODE="EXCLUDE" THEN
    OUTPUT &_excldsn;
  ELSE
    OUTPUT &_datadsn;
  RUN;

%MEND CLEANDAT;

```

The CHKNODAT utility (Fig. 4) checks the input data set for remaining data following the removal of NULL and EXCLUDED results.

Figure 4.

```

%MACRO CHKNODAT(_indsn=WORK.DATA);
  %LET NODAT=N;

  PROC SQL;
    SELECT COUNT(*) INTO :obsnm FROM &_indsn;
  QUIT;

  %IF &OBSNM=0 %THEN %LET NODAT=Y;

%MEND CHKNODAT;

```

If **no** results remain following the removal of NULL and EXCLUDED values, the ERRMSG routine (Fig. 5) is called to set an error flag.

Figure 5.

```

%MACRO ERRMSG;
  %LET ERRFLAG=Y;
  %PUT &nodat;
%MEND ERRMSG;

If results do remain following the removal of NULL and EXCLUDED values, the ADD_GRP routine (Fig. 6) is called to add group markers.

```

Figure 6.

```

%MACRO ADD_GRP(_indsn=WORK.INDSN,

```

```

                _colname=GROUP,
                _chkcol=day);
DATA &_indsn;
  ATTRIB group LENGTH=3;
  RETAIN group;
  SET &_indsn;
  SELECT;
    WHEN (0<&_chkcol<=10) &_colname=1;
    WHEN (10<&_chkcol<=20) &_colname=2;
    WHEN (20<&_chkcol<=30) &_colname=3;
    OTHERWISE &_colname=.;
  END;
RUN;

%MEND ADD_GRP;

```

Testing

Unit tests require processing isolation to identify errors. Processing boundaries permit the expected output of a unit's input to be defined. *Control-flow* processing is evaluated through path coverage and multi-level structural tests. Path coverage tests ensure that *all* channels of processing are followed. Multi-level tests allow isolation to occur at various scales. *Data-flow* is tested through range of input functional tests, which can also serve as structural tests at mid- and upper levels. Our discussion is restricted to *unit* testing at various levels; we assume that units have a basis in requirements but integration, requirements, and regression tests are performed separately.

Well designed macros have a discrete task. Reliable unit tests can be created by constructing harnesses (scripts) which serve as *scaffolding* that provides sample data in test calls. The resulting output can then be compared with expected results. A suite of test cases across the range of a function's input determines function reliability. This process is repeatable and can be applied to increasing modular levels to establish validation.

Shown below are four examples that demonstrate how harnesses can be structured using this testing approach. Figure 7. illustrates a sample input data set used in a lowest level unit test and later referenced in a mid-level test of a sub-driver.

Figure 7.

```

/* Test data set #1*/
data work.test;
  input rsltno 3.0 +1 day 2.0 +1 rsltval 3.0 +1 qccode $8.;
cards;
001 01 123 INCLUDE
002 14 324 EXCLUDE
003 21 635 INCLUDE
004 25
005 30;
run;

```

Test #1 is a range of input unit test at the function level where some, but not all data are NULL or EXCLUDED. As shown in Figure 8, the function is passed the input data set (Fig 7), and its output are printed. Notice that NULL and EXCLUDED values are placed in new data sets and that non-NULL and non-EXCLUDED results are appropriately retained in the original input data set.

Figure 8.

```

/* Utility test call #1*/

%CLEANDAT(_indsn=work.test,
          _nuldsn=work.nulldsn,
          _excldsn=work.excldsn)

```

/* Utility test output */

WORK.NULLDSN
Test #1

OBS	RSLTNO	DAY	RSLTVAL	QCCODE
1	4	25	.	
2	5	30	.	

WORK.EXCLDSN
Test #1

OBS	RSLTNO	DAY	RSLTVAL	QCCODE
1	2	14	324	EXCLUDE

WORK.TEST
Test #1

OBS	RSLTNO	DAY	RSLTVAL	QCCODE
1	1	1	123	INCLUDE
2	3	21	635	INCLUDE

Test #2 is a range of input unit test at the function level where all data are NULL or EXCLUDED. The function is passed a different input data set (Fig. 9), and its output are shown in Figure 10. Again, NULL and EXCLUDED values are placed in new data sets, but since there were no non-NULL or non-EXCLUDED results in the input data set, it retains no records.

Figure 9.

/* Test data set #2 */

```

data work.test2;
  input rsltno 3.0 +1 day 2.0 +1 rsltval 3.0 +1 qccode $8.;
cards;
001 01 123 EXCLUDE
002 14 324 EXCLUDE
003 21 635 EXCLUDE
004 25
005 30;
run;

```

Figure 10.

/* Utility test call #2 */

```

%CLEANDAT(_indsn=work.test2,
          _nuldsn=work.nulldsn,
          _excldsn=work.excldsn)

```

WORK.NULLDSN
Test #2

OBS	RSLTNO	DAY	RSLTVAL	QCCODE
-----	--------	-----	---------	--------

1	4	25	.
2	5	30	.

WORK.EXCLDSN
Test #2

OBS	RSLTNO	DAY	RSLTVAL	QCCODE
1	1	1	123	EXCLUDE
2	2	14	324	EXCLUDE
3	3	21	635	EXCLUDE

Test #3 is a path coverage unit test at the sub-driver level where some, but not all data are NULL or EXCLUDED. As shown in Figure 11, the function is passed the original input data set (Fig 7), and its output are printed. Notice that NULL and EXCLUDED values are placed in new data sets and that non-NULL and non-EXCLUDED results are appropriately retained in the original input data with the GROUP column added.

Figure 11.

/* Driver test call #1 */

```
%PRP_DATA(_datadsn=work.test,
           _nulrslt=WORK.NULLDSN,
           _excrslt=WORK.EXCLDSN)
```

/* Driver test output */

WORK.NULLDSN
Test #3

OBS	RSLTNO	DAY	RSLTVAL	QCCODE
1	4	25	.	
2	5	30	.	

WORK.EXCLDSN
Test #3

OBS	RSLTNO	DAY	RSLTVAL	QCCODE
1	2	14	324	EXCLUDE

WORK.TEST
Test #3

OBS	GROUP	RSLTNO	DAY	RSLTVAL	QCCODE
1	1	1	1	123	INCLUDE
2	3	3	21	635	INCLUDE

Test #4 is a path coverage unit test at the sub-driver level where all data are NULL or EXCLUDED. The function is passed a different input data set (Fig. 9), and its output are shown in Figure 12. Again, NULL and EXCLUDED values are placed in new data sets, but since there were no non-NULL or non-EXCLUDED results in the input data set, it retains no records.

Further, since no records remain in the input data set, the NODAT flag was set which, in turn, triggered an error message to the log (not shown).

Figure 12.

/* Driver test call #2 */

```
%PRP_DATA(_datadsn=work.test2,
           _nulrslt=WORK.NULLDSN,
           _excrslt=WORK.EXCLDSN)
```

/* Driver test output */

WORK.NULLDSN
Test #4

OBS	RSLTNO	DAY	RSLTVAL	QCCODE
1	4	25	.	
2	5	30	.	

WORK.EXCLDSN
Test #4

OBS	RSLTNO	DAY	RSLTVAL	QCCODE
1	1	1	123	EXCLUDE
2	2	14	324	EXCLUDE
3	3	21	635	EXCLUDE

The effectiveness of range of input tests is increased when a suite of zero-many-all test cases is used as the minimum range of test conditions. Tests #1 and #2 illustrate the *many* and *all* cases; a third test would contain test data where *zero* NULL or EXCLUDED values occurred. Examples #3 and #4 test both paths through the driver; conditional execution allows more control of sophisticated processing, but increases the number of test cases for validation. A suite of test cases could be constructed similarly to perform structural and functional tests of the MASTER driver for a higher level of unit testing.

Summary

We have shown a method for designing and constructing macro systems that allows use of standard validation techniques. Forethought in the design of macros produces programs which are easy to build and test for production-use in regulated environments. Care in program construction permits evaluation of data-flow and control-flow by allowing path coverage, range of input, and multi-level tests.

References

- Burger, Thomas H. and Philip M. Pochon. 1997. *Structured Design and CASE Tool Use in SAS® Macro Systems*. Proc. of the Eighth Annual MidWest SAS Users Group Conference. Chicago, IL.
- Coward, P. David. 1997. *A Review of Software Testing*. In Software Engineering. M.Dorfman and R.H. Thayer, eds. IEEE Computer Society Press. Los Alamitos, CA.

Fujii, Roger U. and Dolores R. Wallace. 1997. In Software Engineering. M.Dorfman and R.H. Thayer, eds. IEEE Computer Society Press. Los Alamitos, CA.

Gill, Paul. 1997. *The Next Step: Integrating the Software Life Cycle with SAS® Programming*. Cary, NC: SAS Institute Inc. 384pp.

Humphrey, Watts S. 1989. *Managing the Software Process*. Software Engineering Institute. Addison-Wesley Publishing Co.

McConnell, Steve. 1993. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press. Redmond, WA.

Newhouse, Russell. 1997. *Validation and SAS® Programming: Benefits of Using the System Life Cycle Method*. Proc. of the 22nd Annual SAS Users Group International (SUGI) Conference. San Diego, CA.

SAS® Macro Language: Reference, First Edition, Cary, NC: SAS Institute Inc., 1997. 304pp.

SAS® Macro Facility Tips and Techniques, Version 6, First edition, Cary, NC: SAS Institute Inc., 1994. 319pp.

SAS® Programming Tips: A Guide to Efficient SAS® Processing. Cary, NC: SAS Institute Inc., 1990. 155pp.

Sloan, Faith R. 1993. *Is the System Analysis Phase Really Necessary ?* Proc. of the 18th Annual SAS Users Group International (SUGI) Conference. Miami, FL.

Yourdon, Ed and Larry Constantine. 1975. *Structured Design*. Yourdon, Inc. New York, NY.

Trademark Notice

SAS is a registered trademark of the SAS Institute Inc, Cary, NC and other countries.

Author Contact

Philip M. Pochon	Thomas H. Burger
Computer Task Group, Inc.	Source Consulting
Castle Creek IV~Suite 208	Bank One Tower ~Suite 3930
5875 Castle Creek Parkway	111 Monument Circle
Indianapolis, IN 46250-4344	Indianapolis, IN 46140
Phone (317) 578-5106	Phone (317) 631-2900