

Structured Design and CASE Tool Use in SAS® Macro Systems

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

Abstract

The SAS® macro facility provides powerful tools for building analysis and reporting systems. SAS® macros permit construction of generalized modular components which can be reliably tested and integrated as a system. However, sound design principals may be overlooked during the translation of requirements to code specification. CASE tools provide a useful means of designing system components and relating them to function. We describe a general approach using pseudo-structured design and CASE technology for the construction of SAS® macro systems.

Introduction

Experience has shown that formal software processes ensure a greater likelihood that coding efforts will produce a (reliable) system meeting user expectations. The design phase of the development cycle is critical because it is where requirements are linked to system entities and where a breakdown may occur in the translation of requirements to code.

A reliable design process aids in the conceptualization of how to build a system to meet requirements and provides a map by which to construct it. In addition, proven design methodologies increase the accuracy and precision of project planning and cost estimation by providing function counts. System design influences the testability of its components and, therefore, has a bearing on its validation and reliability. Further, maintenance costs are tenfold development costs and are inversely proportional to a system's stability and its ease of support and enhancement.

This is not a tutorial on Structured Design nor a philosophical debate of its primacy as a design methodology. Our purpose is to demonstrate the utility of a structured approach in designing SAS® macro systems by making the following points: 1) SAS® macro systems are well-suited to structured design methodology 2) CASE tools can effectively represent the design of macro systems 3) Macro systems built using structured design and CASE tools are highly testable. It is worth noting that the methods we describe are adaptable to AF/SCL® application development and may be adaptable to Intranet application deployment, as well.

Structured Design Overview

Tightly written ('perfect') programs have diminished value if the design of their processing is unsound. Although a level removed, the design of a system has a direct bearing on the contents of a program and how the code fulfills its purpose. The stability, testability, efficiency and maintainability of a system rests in the architecture because it defines the context in which a block of code is executed.

Structured Design is a formal method of organization for designing a system's components and their interrelationships to best meet a set of requirements (Yourdon and Constantine 1975). It embodies the principals of functional decomposition,

modularity, cohesion and coupling, among others. It is frequently applied as a top-down decomposition of the physical design after a logical view of the problem domain has been defined. The purpose of functional decomposition is to divide complex problems with abstract solutions into a hierarchy of smaller elements having a more clearly defined resolution. The simplification of a problem domain into logical components clarifies interrelationships and dependencies. This aids in the identification of code responsible for achieving individual tasks, thereby ensuring that requirements are met. An important aspect of decomposition is that it reveals sites of consolidation and shared functionality, thereby identifying opportunities for reuse.

Modules are typically defined as a logical grouping of functions whose collective effect is to perform some defined process. Modularity refers to the degree of isolation achievable in delineating the tasks of the constituent functions for a (single) specific purpose. Effective modularization is achieved by hierarchical structuring, good cohesion and loose coupling.

Cohesion describes the degree of relatedness of operations within a function. Strong cohesion results in structural integrity and is characterized by functionally homogenous operations in a routine. Coupling denotes the relatedness of functions or the strength of connectivity among them. Loose coupling produces flexibility in the association of functions with one another. Clearly, ideal functions possess both strong cohesion and loose coupling.

How then should a macro system be organized to ensure maintainability? Split into modules? How should modules be structured to maximize testability and reusability? Further, how should modules relate to one another through dependence, hierarchy and sequence?

SAS® Macros

The appeal of the macro facility for constructing systems rests in its ability to produce powerful and generalized modular functions. Because the macro facility is an extension of BASE SAS®, it provides an easy means of encapsulating existing routines (DATA step, PROCs) into macro functions for various degrees of cohesion. Because it uses its own language, the macro facility provides its own hooks (parameters, macro variables) for loose coupling. By using modular functions for (conditional) execution and dynamic code generation, the risk of spaghetti coding is greatly reduced. This permits the creation of well-controlled, sophisticated processing.

To impose hierarchy, programs should be decomposed into control functions and terminal functions as much as possible. We define primary drivers as module control functions which may call secondary drivers or sub-drivers. Sub-drivers control small modules and may call secondary sub-drivers or utility functions. Utility functions perform specific operations and are typically <20 lines of code in length. Nested macro function definitions should be avoided since they are less efficient than calls for externally defined macros. With a few notable exceptions (conditionals, localization), embedding code in the body of

drivers and sub-drivers should be avoided because it reduces testability.

Due to their greater efficiency and ease of use, name style macro invocations (%name) are our preference. Keyword macro parameters greatly increase program flexibility by permitting table names, macro variables and operators to be passed into functions. Judicious use of macro variable localization permits accurate reference to the value of locally set, but globally referenced, flags.

Compiled macros are most efficient, but the source code must be saved for future modifications. However, compiled macros are not portable across platforms, so the source code needs to be ported and then recompiled.

Figure 1 illustrates the program structure of a module which checks the integrity of input data. Shown is the macro definition of a small sub-driver containing calls for three functions. The call for the first function is always executed. Calls for subsequent functions are conditionally made by the %IF-clause based on the value of a macro variable (ERRFLAG) set in the first function. When &ERRFLAG=Y, the sub-driver is exited through the use of the TERM label. The second and third calls provide an example of function reuse. Note that the same function is called twice to generate different output data sets (_outdsn) based upon the value of the operator supplied to the parameter '_cndtn'. The sub-driver terminates following a call for the third utility.

Figure 1.

```
%MACRO CHCKDAT(_rsltdsn = work.one,
               _calcdsn = work.two,
               _subdsn = work.three,
               _excldsn = work.four,
               _nulldsn = work.five,
               _missdsn = work.six);

/* ***** */
/* Call the utility CHCKDUPP to check */
/* for duplicate input result values. */
/* ***** */

%CHCKDUPP(_anldsn = &_subdsn)

/* ***** */
/* If there were duplicate input      */
/* results, then exit the function.    */
/* ***** */

%IF (&ERRFLAG = Y) %THEN %GOTO TERM;

/* ***** */
/* Call the utility RMVRSLT to remove */
/* EXCLUDED input results.            */
/* ***** */

%RMVRSLT(_anldsn = &_subdsn,
        _inptdsn = &_rsltdsn,
        _outdsn = &_excldsn,
        _cmpdsn = &_calcdsn,
        _cndtn = %NRBQUOTE(EXCL='Incl'))

/* ***** */
/* Re-use the utility RMVRSLT to      */
/* remove NULL input results.          */
/* ***** */
```

```
%RMVRSLT(_anldsn = &_subdsn,
        _inptdsn = &_rsltdsn,
        _outdsn = &_nulldsn,
        _cmpdsn = &_calcdsn,
        _cndtn = %NRBQUOTE(RSLT^=.) )

/* ***** */
/* Call the utility RMVMSSNG to       */
/* remove MISSING input results.      */
/* ***** */

%RMVMSSNG(_anldsn = &_subdsn,
         _inptdsn = &_rsltdsn,
         _outdsn = &_missdsn,
         _cmpdsn = &_calcdsn)

%TERM:    %MEND CHCKDAT;
```

The module is called by issuing a name style invocation from within the body of its calling driver, as shown in Figure 2. Notice that it is the value of the parameters supplied in the invocation (Fig. 2) that are processed in the module, rather than the values used in the macro definition (Fig. 1). The latter serve only as placeholders.

Figure 2.

```
%CHCKDAT(_rsltdsn = work.results,
         _calcdsn = work.calcdsn,
         _subdsn = work.subrslt,
         _excldsn = work.exclrslt,
         _nulldsn = work.nullrslt,
         _missdsn = work.missrslt)
```

Figure 3 shows the program structure of the first utility, which checks for duplicate input result values. The SQL procedure is used to take advantage of the reserved macro variable SLOBS, which is the number of rows processed in the SQL procedure. Its value is used to conditionally set the value of &ERRFLAG, which is used in the conditional statement in the calling sub-driver (Fig. 1).

Figure 3.

```
%MACRO CHCKDUPP(_anldsn = work.seven);

/* ***** */
/* Use a SQL step to create a data    */
/* set containing a count of the      */
/* results in each analysis unit.     */
/* ***** */

PROC SQL;
  CREATE TABLE WORK.COUNT AS
  SELECT ANLYUNIT
  FROM &_anldsn
  GROUP BY ANLYUNIT
  HAVING COUNT(*) > 1;
QUIT;

/* ***** */
/* If there are duplicate values, set */
/* the error flag and exit the        */
/* function.                           */
/* ***** */

%IF (&SLOBS > 0) %THEN %DO;
```

```
%LET ERRFLAG=Y;

%END;

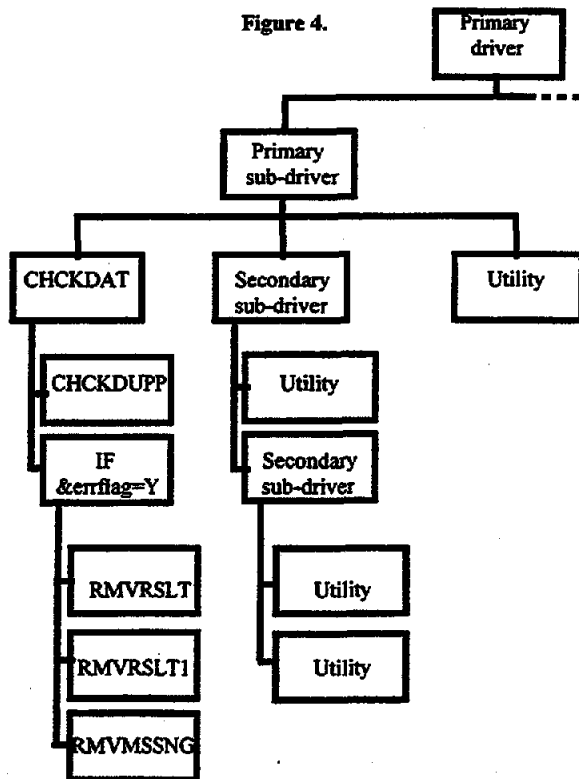
%MEND CHCKDUPP;
```

What convenient means is there for representing the physical structure of a macro system? Modular components naturally lend themselves to structured engineering. CASE (Computer-aided Software Engineering) tools are designed to provide a mechanism for modeling system architecture and function attributes.

CASE Tool Use

The physical design of macro systems can be represented by Physical Structure Diagrams (PSD) and Function Hierarchy Diagrams (FHD), among others. We feel that FHDs provide a more intuitive representation of processing logic. FHDs are simply horizontal decomposition charts with a Data Dictionary (DD) that represent the hierarchic relationships in a program structure. They provide an organizational representation of the entities in a physical structure.

In the examples presented, the Oracle Designer2000® CASE tool was used to represent the FHD and its DD. Figure 4 displays the sequential processing from the top down and from left to right that occurs in the module in Fig 1. A primary driver calls a primary sub-driver which, in turn, calls our module, a secondary sub-driver and a utility. We find it useful to represent conditionals in the body of the diagram to completely represent processing logic. A disadvantage of the diagrammer is that functions can not be referenced more than once by the same name when different parameter values are passed. Aliases must be used to reference a previously defined function.



The DD is a diagram-linked data base repository that provides storage, organization and management of system metadata. Data base structures are cataloged in the repository using the Repository Object Navigator (RON) and are thereafter available to the DD. The DD is accessed by double-clicking a diagram function. It is used to define the tables, entities and attributes associated with each function in the FHD. Figure 5 shows the Definition tab for our module. The function's purpose and point of origin are identified by a definition field and a parent description. Because this is a mock sub-driver, it has no parent.

Figure 6 displays the Entity Usage tab accessed from the same screen. It shows all entities (tables, macro variables) associated with the function. The Attributes tab allows specification of the column(s) or macro variable(s) referenced by the function and whether they are updated, inserted or retrieved.

Figure 5.

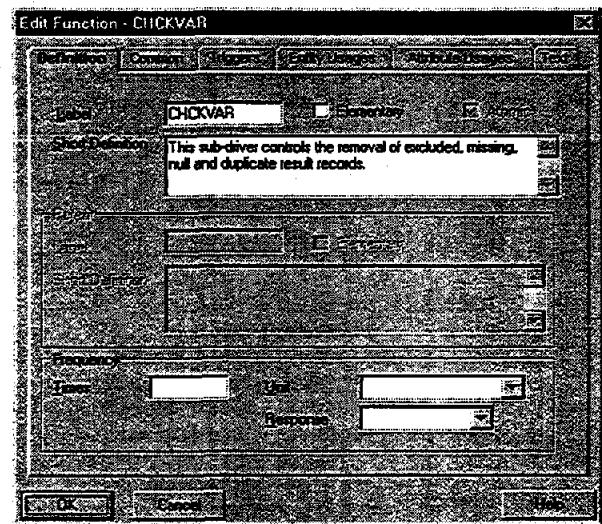
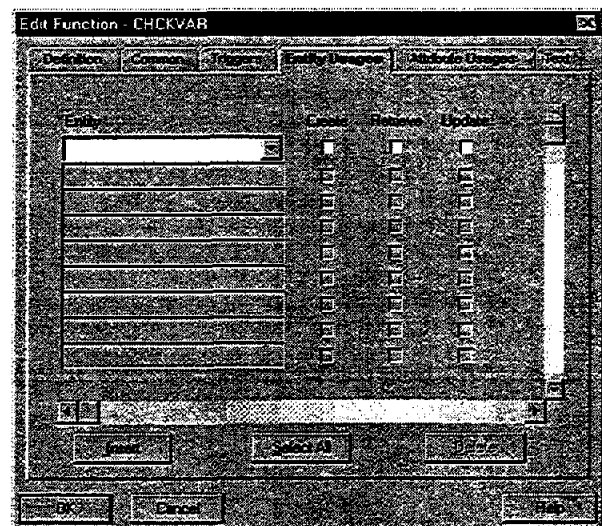


Figure 6.



*Mention of product name does not constitute endorsement.

A CASE tool provides an invaluable way to manage and represent complex relationships during system design. A very useful feature is that the tool produces reports describing metadata associations. Design documents also serve a number of purposes external to actual code development. Peer-reviewed design walk-throughs provide an opportunity to detect flaws at the point where they are least costly to mitigate. They may reveal limitations in the existing definition of software requirements. They serve an important role as Validation Deliverables for the systems QA/QC division. Designs may be critical to support staff who provide ongoing maintenance and enhancement.

Testability

A critical aspect of Software Quality is the reliability of a system in performing its intended purpose. Reliability is achieved by the validation of a system and its components. Validation requires the ability to test a function's task; its testability rests in its composition. The development process and reliability testing work hand-in-hand to produce high quality software.

The entire testing process frequently detects less than two-thirds of errors. Incremental stages of testing often reveal only half the errors present. Establishing a solid foundation minimizes the magnification of instability that occurs with increasing complexity. The stabilization of a structure beginning with its most atomic element followed by a progression up its hierarchy is a fundamental principal of engineering.

Unit testing is the testing of individual functions or modules and represents the first tier of validation. The scope of our discussion is restricted to this level. Successful unit testing requires the isolation of processing in order to identify missteps. Effective modularization results in processing isolation.

Well designed macros have a discrete task. The presence of processing boundaries lends itself to a clear definition of the expected outcome of a macro's input data. This feature can be leveraged to create reliable unit tests by constructing test harnesses (or scripts) that provide sample data in macro test calls. The resulting output can then be compared with expected results to assess function reliability. A suite of test cases representing the full range of function inputs assures its stability. This process is repeatable and can be applied to higher modular levels to establish validation effectively.

Summary

We have shown the application of a standard software design methodology to SAS® macro program structure and physical design. Significant gains in processing simplification and interpretation can be made using a formal approach in program development. The resulting program structure is highly modular and easily applied to macro system construction. CASE tools provide a useful means of representing this structure and associating data elements. Functions created using this approach are easily maintained and highly testable.

References

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

SAS Institute Inc., *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.

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

Trademark Notice

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

Oracle and Oracle Designer2000 are registered trademarks of the Oracle Corporation, Redwood City, CA and other countries.

Other brand and product names are registered trademarks or trademarks of their respective companies.

Acknowledgment

The authors would like to thank Eli Lilly and Company®, under whose auspices this work was performed.

Author Contact

Thomas H. Burger
Source Consulting
Bank One Tower ~ Suite 3930
111 Monument Circle
Indianapolis, IN 46140
Phone (317) 631-2900

Philip M. Pochon
Computer Task Group, Inc
Castle Creek IV ~ Suite 208
5875 Castle Creek Parkway
Indianapolis, IN 46250-4344
Phone (317) 578-5106