STRUCTURED SYSTEMS ENGINEERING AND THE SAS® SYSTEM: GODZILLA VS. MEGALON

Stuart Leven, Bureau of Labor Statistics
James E. Johnstone, Bureau of Labor Statistics

Abstract

The SAS system is well known as a powerful data analysis and report generation system, and its capabilities are now extending into Executive Information Systems (EIS) and other enterprise-wide information systems applications. It is also a full-featured programming language suitable for use in traditional, large-scale, production-level data processing systems. This paper examines the relationship, sometimes harmonious and sometimes not, between the SAS system and the structured systems engineering techniques used to build traditional systems. To the extent that this relationship is not harmonious, the authors suggest ways to overcome or mitigate the difficulties.

INTRODUCTION

Godzilla vs. Megalon

In the science fiction movie Godzilla vs. Megalon, two titans clash, with Japan as the battleground. Godzilla is the champion of Japan. Megalon is the champion of Seatopia, a hidden, underwater world. The Japanese and Seatopians don’t know each other very well. They live in completely different worlds, with different values and very different infrastructures. If Godzilla and Megalon could cooperate, the power of each separately would be multiplied several-fold. Instead, Godzilla and Megalon, the defenders of these different worlds, fight each other with their arrays of weapons, with predictable injury to one another and the surrounding countryside.

This is somewhat the situation of the SAS system and structured systems engineering. In many ways, they inhabit different worlds, each complete in itself, but with different values and different infrastructures. Like Godzilla and Megalon, they are both quite large and, at least at first glance, maybe a little terrifying, and they are both capable of wreaking significant havoc if they aren’t managed carefully. When the SAS system and structured systems engineering work together, the benefits are significant. Sometimes, though, the SAS system and structured systems engineering are at odds with one another, and, like Godzilla and Megalon, their disagreements can cause injury to the surrounding countryside, in this case our development and maintenance efforts.

Overview

This paper examines both the advantages and difficulties in developing and maintaining traditional, large-scale, production-level data processing systems using SAS software in a structured systems engineering environment. The paper draws on the experiences of the Bureau of Labor Statistics (BLS) in the development, maintenance, and operation of a SAS software based, large-scale production system that computes the Consumer Price Index (CPI). The system is written almost entirely in SAS software code. The SAS software code performs all of the core computations needed to produce the CPI, as well as data analysis and report generation. At many installations, the core processing of a production system would be done in a traditional programming language, with the SAS system reserved for such functions as the creation of user interfaces, data analysis, report generation, and other activities that use the data but don’t alter it. In the CPI production system, SAS software does the core processing as well as all of these other activities.

This paper is an extension of work which was published in last year’s SUGI proceedings under the title "The Evolving Role of the SAS System in the Development of the Production System for the Consumer Price Index (CPI)," and which is cited fully in the references section. Portions of this paper include points from that work in the form of reiterations where appropriate.

IMPORTANCE OF THE CPI PRODUCTION SYSTEM

The Consumer Price Index is a modified Laspeyres index representing a ratio of costs of purchasing a set of consumer goods and services of constant quality and quantity in two different time periods. This fixed set of items is referred to as the consumption market basket. The ratio of these costs from month to month is called a price relative and is used to inflate the index values from a previous time period to the current time period. The Consumer Price Index measures these price changes in specific geographic areas as well as by region and across the U.S. The reader is referred to the BLS Handbook of Methods (1992) for further details concerning the index as a measure, computational formulae, and statistical concepts employed in various methodologies.

The impact of the index is far reaching. The CPI is widely used as an economic indicator in a variety of economic models. It is used as a deflator in the Personal Consumption component of the Gross Domestic Product (the replacement for the Gross National Product), retail sales, and other important economic measures to provide estimates in constant dollars. In addition, the index has a direct impact on the livelihood of more than half of the U.S. population. The index is used within federal programs to adjust cost of living increases in social security, school lunch subsidy programs, food stamp allocation and other welfare benefits, and
as an inflator for the official poverty threshold. Within municipal programs, it is widely used as an inflator or limit for rental payment increases, child support and other court stipulated payments, as well as other distributions from municipal funds. In addition, it is used within many wage earner contracts to determine increases in pay and other forms of compensation. Errors in the CPI at the national level as small as 0.1 percent can result in misdirection of hundreds of millions of dollars within the national economy.

Because of the foregoing, it is clear that accuracy of estimates, and, hence, program correctness and software reliability, are of crucial importance in the CPI production system. In addition, the schedule of the monthly cycle for computing and releasing the Consumer Price Index is extremely tight. There is almost no room for correcting program errors detected in production or for resuming the system after corrections are made.

SYSTEM SIZE AND COMPLEXITY

The CPI production system is a large, complex system. It includes hundreds of programs that total roughly one million lines of code. The initial implementation of the system required several calendar years and scores of person years in software development activities. Maintenance and enhancements have required a considerable expenditure of time and effort.

The system processes a large volume of data. The data for the computation of the CPI includes data from the Commodities and Services (C&S) Survey, which collects up to 85,000 individual sampled price quotations collected each month from approximately 21,000 retail and service establishments. The system also processes data from the Housing Survey, which collects information on the cost of shelter and accommodations totalling up to 600 pieces of individual information for each sampled housing unit. Approximately 60,000 units are priced in the sample.

The system also performs a large number of computations and generates a large volume of results. To arrive at price changes (termed price relatives), the price information from the C&S and Housing surveys goes through a complex series of procedures, including imputation (the estimation of values for missing items based on other information within the survey), editing, and estimation operations. These price relatives are used to move (inflate or deflate) index values associated with the goods and services in the market basket over time. These indexes are aggregated at various product and geographic levels using weighting schemes which represent the relative importance of the commodity within the geographic or product hierarchy. The highest level of aggregation is the all items index at the national geographic level (typically, this is the CPI reported on the news). Each of the indexes described above is computed monthly, semiannually, and annually. All together, there are roughly 70,000 indexes computed every month.

The individual operations performed in the system can be quite complicated. For example, sophisticated imputation operations are common in the system. Imputation methods include percentage distribution, hot deck matching, cell collapsing, and weighted means. Many subsystems employ a hierarchy of imputation methods where SAS software is employed to examine imputation results and make a determination of the optimal method based on requirements. Perhaps the most complex method is hot deck matching. The Consumer Expenditure Survey (CE) system, which is a subsystem of the CPI program, does imputation and hot deck matching intensively. In CE hot deck matching, cells of valid and invalid records are counted by all combinations of the non-collapsible and collapsible cell definition variables. The cells are collapsed until sufficienty is reached or, failing that, until only non-collapsible combinations remain. Then, each invalid record is matched with a valid record from the same collapsed cell. In the simplest implementation, the valid matching record is selected randomly. In a more complex implementation, a match variable is identified, and the valid record selected is the one where the value of the match variable is closest to the value of the match variable on the invalid record. Ties are broken by random selection, alternating direction. In a very complex implementation, several match variables are identified, and the best match on the combination of variables is sought, using predefined scoring criteria. These and other imputation methodologies are discussed in more detail in previous papers (e.g., Mopsik and Dippo, 1985).

The system also generates a large volume of printed results. At the start of the processing cycle, the system generates the forms used to collect pricing information. This processing is complex, since each quotation is processed individually, and the corresponding collection form entries have to be individually customized. In addition, there are hundreds of reports produced for large-scale consistency checks and expert review at various stages of processing. These reports include those generated from the usual PROCs such as PRINT, FREQ, and TABULATE as well as highly complex custom reports which are data-driven and require sophisticated full-page processing.

The sheer volume of information collected requires a large amount of data storage and considerable computer resources to process the data through each stage of processing. The system is executed on IBM® 3090™ series computers or equivalents, running under MVS, mainly in batch mode. The data and results are stored and accessed using the non-commercial RAPID database management system from Statistics Canada. Using SAS/Toolkit™ software, the Bureau of Labor Statistics wrote and maintains interfaces between the SAS system and RAPID.

Processes in the system are sequential, cumulative, and dependent. Database updates performed by a process feed into subsequent processes. Because of this "value added" approach, the programs in the system have to be carefully integrated, even though they can be maintained and run separately. The foregoing would not be the case with many SAS software based systems.
where the data is used but not altered. In many such systems, the programs can be run at will, without regard to the other programs in the system.

THE SUCCESS OF THE SAS SYSTEM IN THE CPI PRODUCTION SYSTEM

The use of SAS software for the CPI production system has been very successful. One measure of the success of SAS software in this application is the fact that all new and replacement systems developed since the initial system implementation have been written using SAS software. When the decision was made to use SAS software as the implementation language for the system, certain advantages were anticipated. All of these benefits have been realized. Indeed, many managers have noted additional advantages beyond those that were originally attributed to SAS software during the strategic planning phase. These advantages are spelled out in greater detail later in this paper.

There are isolated pockets of the system that were implemented in traditional programming languages. These are perfectly good programs that have run successfully for years. Often, however, especially when one of these programs has to be maintained, there is a chorus of calls for the program to be rewritten using SAS software. This is a further, and perhaps even more telling measure of the success of SAS software in the CPI production system.

STRUCTURED SYSTEMS ENGINEERING AND THE SAS SYSTEM - THE SYSTEM DEVELOPMENT LIFE CYCLE

Introduction

In spite of the successes described above, there have been some difficulties in using SAS software in the CPI production system. Many of these difficulties relate to the sometimes problematic interaction of the SAS system and structured systems engineering.

To facilitate a discussion of the advantages and difficulties of the use of SAS software in this environment, we will review the phases of the system development life cycle and discuss the role of SAS software at each stage. To the extent there have been difficulties, we suggest ways to overcome or mitigate them.

Generally, the life cycle model that has been used in the CPI program can be described as a traditional life cycle approach as championed by Yourdon and DeMarco in the late 1970s and early 1980s (DeMarco, 1978). Thus, for each system or subsystem that performs processing tasks, there are several phases: requirements definition and analysis, system specification, prototyping, system design, program design, coding, testing, and maintenance.

Requirements Definition, Analysis, and System Specification

The first stage is the generation of requirements as defined by the users and refined during structured analysis where the software developers improve their understanding of the users' true needs, both stated and hidden. Once requirements are fully understood, system specification can begin. This process follows structured analysis methodology and results in a functional specification (data flow diagrams and process descriptions) from which the basic data structures at the boundaries of the system can be defined.

Since requirements development and analysis are independent of implementation language, the SAS system and structured systems engineering are in accord at this stage. In fact, the SAS system is of significant help during these processes by facilitating the coding of ad hoc programs to examine methodologies and assumptions underlying the users' perceptions of their needs.

Prototyping

In the CPI program, major prototyping has generally been done after requirements definition, analysis, and system specification. Depending on its purpose, though, prototyping can be done at other points in the system development life cycle. The overall power and ease of use of the SAS system make it possible to develop major prototypes rapidly, and the data analysis and reporting features make it easy to display prototype results. Those elements of the prototype that show promise can be kept and used in subsequent development activities. Those elements that do not look useful can be discarded. Because prototypes can be developed rapidly, there need not be a feeling that a major development effort was wasted when portions of the prototype are discarded.

Design - Generally

The next stage is design, including structured system and program design. The logical functional requirements are redefined in terms of system function. Once the overall system design has been determined, program design continues with the objective of a complete physical definition of the processing. This definition in physical terms can result in processes which correspond directly to PROCs or DATA steps, and the communication or movement of data is accomplished via SAS data sets. Coding can be accomplished quickly through translation of program design elements into SAS language statements. Processes in the design become PROCs or DATA steps. Dataflows in the design become SAS datasets that are created in the process where the dataflow originates and that are read in the process where the dataflow terminates. (Mopsik, 1984) This easy translation of design into code facilitates prototyping if it is done at this stage, permits illustration of system operation, demonstrates feasibility of design constructs, identifies design weaknesses, and facilitates comparison of alternative coding techniques.
This does not eliminate the need for good modularization decisions during design. If physical design elements were simply translated into DATA and PROC steps, the result would be a monolithic program. The program would be "modularized" only because base SAS software is inherently modular - processing is broken into DATA and PROC steps. Usually, however, it is also desirable to break a system into modules where each module is a SAS macro. The appropriate boundaries between macros will not be as clear as the appropriate boundaries between DATA and PROC steps. The designer has a great deal of flexibility in determining how to modularize and has to make this decision guided primarily by good design principles. The decision as to how to modularize into macros will not be hindered by SAS language elements, but neither will this decision be assisted by SAS language elements.

**Design - Processing Paradigms**

Despite the fact that the SAS system includes a full-featured programming language, the SAS system's processing paradigm is quite different from the record processing paradigm of traditional programming languages. This is best illustrated by an example.

In the Consumer Expenditure Survey system, there is a large program that edits data relating to vehicle expenditures and loans. The system extracts collected data from a database, performs a long series of computations, and loads the results back into the database. The computations include examination of the data for suspect values, replacement of suspect and missing values, and computation of derivative fields. The computations are numerous and involve multiple fields. For the most part, each record is processed without reference to the values on the other records.

The way this system would be implemented in a traditional, record-oriented programming language is quite different from the way it was actually implemented using SAS software. In a traditional programming language implementation, the extracted dataset would be read once, or, perhaps, a very small number of times to allow for sorting and merging. In the SAS system implementation, the dataset is read repeatedly, roughly 20 times. In a traditional language, each record would undergo all of the necessary computations before the next record was read. In the SAS system implementation, every record in the file undergoes a single computation or group of related computations before the next computation is executed. In a traditional language, for each computation or group of related computations that had to be performed, selected data elements from the record being processed would be passed by parameters into and out of a subroutine. Only those data elements relevant to the computation would be passed. In the SAS system implementation, for each computation or group of computations, the entire file is passed to a subroutine, and the updated file is passed out of the subroutine. The records on the passed file may include many data elements that are not germane to the processing being done in the subroutine.

This discussion should imply no judgment as to which approach is better. Both approaches have advantages. What is important is that the paradigms are different, and that the traditional paradigm is more common. Structured design and modularization techniques and the coding techniques that accompany them were developed with the traditional processing paradigm in mind. As such, when structured design and modularization techniques are taught or published, a traditional processing paradigm is assumed - all of the examples and case studies use the traditional paradigm, and all of the coding examples show traditional implementations.

This is a case of discord between the SAS system and structured systems engineering. One way to resolve this discord is to change paradigms. The SAS system implementation could use the traditional processing paradigm. The program could be structured as a single large DATA step, and each record could be sent to a series of subroutines and be processed to completion before the next record was read. (Just to be fair, and in the spirit of not judging these approaches, it should also be noted that the traditional programming language implementation could use the SAS system processing paradigm. For each computation or group of related computations, the main program could pass an entire file of records to the relevant subroutine, whereupon the file of records would be read and processed.)

This switch of paradigms, though, would probably never be done. The traditional language processing paradigm is natural for traditional programming languages, and the SAS system processing paradigm is natural for SAS software.

Fortunately, there is another way to resolve this discord, and that is to adapt structured design and modularization techniques so that they can be used with the SAS system processing paradigm. The method of adaptation, however, is not always clear, and there is likely much duplication of effort as different users make their own adaptation.

**Coding**

Coding follows the completion of design. It is at this stage that Godzilla really meets Megalon. Unlike requirements development and analysis, which are language-independent, and unlike design, which is only partially language-dependent, coding is totally engaged with the specifics of the language. At this stage, both the helpful features and the problems of the language become evident.

**Coding - Advantages of the SAS System Generally**

Managers and programmers in the CPI program have noted many advantages of the SAS system in the coding phase. These advantages include the automatic read/write handling features, automatic handling of empty input datasets, the wide variety of formatting capabilities, the large number of useful built-in functions, the ability to read a wide variety of input data...
structures, the availability of preprogrammed PROCs, the self-documenting features of SAS data sets, access of data by variable name without knowledge of file layout, informative messages and diagnostics, and the ability to implement user-written procedures.

In addition, as suggested in the discussion of design, the SAS Macro facility provides a modular packaging tool where processing tasks can be isolated into main routines and subroutines. Each subroutine can be implemented as a macro, often consisting of related DATA and PROC steps. Some generic macros consist of a single DATA step or PROC, or even a portion thereof. These include error trapping routines, the setting of error messages, abort routines, and dynamic determination of PROC coding statements. Modules can be controlled using macro variables as parameters and can be invoked unconditionally or as needed using %IF logic (Phillips, 1985 and 1986). The use of the SAS Macro facility provides considerable flexibility, particularly in the conditional compilation and execution of code as determined by other results within the program. This results in more power than can typically be realized in most traditional programming languages (Henderson, 1982).

Other aspects of the SAS system in the coding phase will be discussed in detail below.

Coding - Processing Paradigms

As described above, the SAS system's processing paradigm is different from that of traditional programming languages. This difference in processing paradigms has a major effect on coding, in addition to the impact on design discussed above. Programmers have to alter their coding technique significantly so that it will fit the SAS system's processing paradigm. This can be a difficult adjustment if the design was not created with a SAS software implementation in mind. This problem can be diminished by creating designs that assume a SAS software implementation and by including in the design the movement of files between modules.

Coding - Modularity and Module Interfaces

One of the elements of a good design is modularity, i.e. the system is broken into pieces, with each piece performing one function. The modules should be as independent of one another as possible. In systems-design terminology, in a good design the modules exhibit loose coupling rather than tight coupling. Loose coupling facilitates making changes and makes it easier for an individual module to be understood in isolation. (DeMarco, 1978, Page-Jones, 1988, Yourdon and Constantine, 1979)

One important way in which modules can be tightly coupled is through undeclared data coupling e.g. module B reads data created in module A, but there is no way to determine this from the code of module A alone or from the code of module B alone. (Meyer, 1988) As will be seen below, it is all too easy to introduce undeclared data coupling when developing SAS software code for the CPI system.

In a typical CPI production system program, the data is extracted from a database, then the extracted data goes through a series of transformations in which the output of one transformation is the input to the next. The form by which this data moves through these processes is the SAS dataset - a process creates a SAS dataset, and then that same SAS dataset is the input to a subsequent process. As such, when one of these programs is modularized into macros, one module creates a SAS dataset, and then that same SAS dataset is the input for another module.

It requires some careful work to achieve independent modules when using the SAS system in the CPI production system. Even in the modular systems described above, there is a temptation to hard code the name of the SAS dataset in both the module that creates it and the module that reads it. One reason this is tempting is that this is what would be done in a monolithic program, and designers and programmers sometimes see modular programs as no more than monolithic programs physically broken into pieces. Another reason is that this is an easy way to code, and the SAS system makes it easy. The name of a SAS dataset is globally available to all processes that execute after the dataset is created. A macro in which a SAS dataset is read can reference the dataset by name even if the macro has no structurally evident connection with the macro in which the dataset was created. Unfortunately, this appealing coding technique creates modules that are highly interdependent and very tightly coupled, because the communication between these modules is not explicit, and the communication is not evident from the code of either of the modules.

One solution to this bit of interdependency is to parameterize the names of SAS datasets that are referenced in more than one module. This is the approach that was taken in the CPI program's Variance Calculation System (VCS). Notice that the purpose of this parameterization was not to make the modules general or reusable by allowing them to read and create datasets with any names. In the VCS, a module would be parameterized even if it was used only once. Rather, the purpose of the parameterization was to diminish the degree of coupling between the modules, by eliminating the hard-coded dataset name.

This problem also extends to the variable level. Suppose that a SAS dataset is created in module A, and VAR1 is a variable on the dataset, and the dataset is read in module B, and VAR1 has to be referenced in module B. It is very tempting to reference VAR1 by name in module B. Like a dataset name, a variable name is globally available anywhere the dataset containing the variable is read. Referencing VAR1 by name in module B, however, would make modules A and B highly interdependent.

In a traditional programming language, such undeclared data coupling at the variable level can be easily avoided by
parameterization. If relatively few variables need to be passed to a subroutine, the individual variables would be parameterized. If a complex structure such as a record or a file of records is being passed to a subroutine, then the structure would be declared in the subroutine, or the structure declaration would be incorporated by reference. Once the variables have been parameterized, whether individually or as part of a structure, they can be referenced in the subroutine by their formal names rather than their actual names.

In a SAS system implementation, it is more difficult to avoid undeclared data coupling at the variable level. In a SAS system implementation, as seen above, it is often a complex structure that is passed between modules. Unfortunately, the solution of declaring the structure in the subroutine is not available. Here the lack of variable and structure declarations in the SAS system becomes a mixed blessing. Usually, we don't want to be bothered with having to make declarations, and it is an advantage of the SAS system that we don't have to do so. In this case, however, when declarations would be desirable, they aren't available.

One possible solution to these data coupling problems would be to parameterize the names of the files and the names of the variables that are actually referenced in the module. This mimics the traditional programming language technique of parameterizing the individual variable names, and it can be done even though it's really a file of records that is being passed rather than individual variables. This technique will be feasible if there are relatively few variables referenced in the module.

Coding - Call By Value

Some traditional programming languages offer the option of "call by value" in writing and calling a subroutine. In call by value, the system makes a local copy of designated input parameters, and all subroutine operations on the designated parameters are done on the local copy. The real input parameters aren't touched. This is a useful safeguard. If a programmer knows that certain inputs to a subroutine should not themselves be changed, but they are needed to generate or change other data, they can be designated as this special sort of parameter. The system guarantees that the actual parameters won't be changed even as an inadvertent side effect.

This safeguard is not available as a built-in feature in the SAS system. A DATA step variable that serves as input to a macro can always be changed in the macro, either deliberately or by mistake. Attempting to hide the DATA step variable name behind a local macro variable is no help; the value of the DATA step variable can still be changed. The programmer can evade this problem by taking care to avoid changing such data if it should not be changed or by making a copy of the input data and then using the copy as the input to the macro.

Coding - Data Structures

One hindrance in coding using SAS software is the difficulty of creating certain compound data structures. For example, there is no equivalent of the hierarchical record structure allowed in PL/I or the records in Pascal that can have other records nested in them. There are no complex data structures for macro variables at all. Each macro variable is on its own, and it can't, for example, be part of an array or a record, even if there are other related macro variables with which it might want to be packaged. SAS datasets can serve the purpose of arrays of records in some applications, since random access is possible, but some processing of arrays of records would be very cumbersome with a SAS dataset implementation. Other complex data structures that are common in other languages can be implemented in SAS software. For example, records that include variables and arrays are much like SAS observations in datasets where some of the variables have been defined as array elements.

Another problem related to data structures is the difficulty of creating common data structures such as stacks, queues, and trees. There are no pointers in SAS software to facilitate construction of these data structures. These structures can be created in the SAS system, as they can be in other languages that lack pointers. (See, for example, Minor, 1989 regarding linked lists, but note the restriction on dynamic insertion and deletion). Creation of these structures is more difficult, however, if pointers are not available.

Perhaps the crux of the problem is that the SAS system has relatively few built-in data structures. The primary data structure is the SAS dataset. The SAS dataset is tabular, making it ideal for relational and record-oriented operations. Unfortunately, not everything that has to be done in a complex system is relational or record-oriented. As such, developers using the SAS system often find that they have to force data into a tabular structure when that is not the natural representation.

It may be that most users manage quite well without stacks, queues, linked lists, trees, and other data structures that are difficult to create within the SAS system. In some cases, the capabilities provided by these data structures simply aren't needed. In other cases, these capabilities can be provided without ever building the structure. For example, binary searches, usually performed with trees, can be done in the SAS system without explicit construction of a tree (Johnstone and Ray, 1989). In fact, the lack of the facilities to create these structures makes for a simpler language and more accessible code, since there is no temptation to use complex facilities when they might not be needed. Those applications that rely on these structures heavily, however, might be better implemented in a language that facilitates the construction of these structures. In addition, it should be recognized that some of these structures are needed in selected SAS software applications and that they can be developed with a little creativity.

660
Coding - Structure Declarations

It is sometimes difficult in a SAS software program to determine what variables are on a dataset. This is a consequence of the lack of variable and structure declarations in the SAS system, which, as mentioned previously, is usually seen as a boon. Sometimes, the necessary tracing is so complicated that the best approach is to insert PROC CONTENTS and run the program. It would be preferable, however, to be able to determine which variables are on each dataset from the static code alone. One way to accomplish this would be to put a KEEP statement (DROP won't do) on each output dataset. This would be burdensome but feasible in an original implementation, but it would create significant problems in maintenance - if a variable had to be added to a dataset and to all of its progeny, a great many KEEP statements would have to be updated, and care would be needed to ensure that all and only those KEEP statements that needed modification were actually modified.

Testing

The testing phase typically follows program design and coding. Several features of the SAS system can facilitate testing. The SAS system's extensive data analysis and print features are of significant benefit in checking intermediate and final results. The code used to perform software tests within the program can be controlled by macro variables. Thus, testing procedures and diagnostics implemented internally in any given program within a system can be activated or deactivated using a single global macro variable embedded in the production system code.

Development of a toolbox of diagnostics, usually written as macros, can also facilitate testing by elimination of redundant code in favor of generalized code which is parameter driven (Septoff, 1985 and Henderson, 1988). The combination of macro variables to activate specialized testing paired with a toolbox approach dramatically simplifies the testing process. Hence, as code moves from the development and testing stages into production, tests established within the design can be reactivated as needed during subsequent system modifications and compared to benchmark results.

Where the SAS system and structured systems engineering diverge with respect to testing is in the use of certain structured code-based (or white box) testing techniques. These techniques were developed assuming traditional programming language constructs and the traditional processing paradigm. These techniques can be adapted for use with code written using SAS software. As was the case with modularization, however, the method of adaptation is not always clear, and much energy is wasted as separate offices develop separate adaptations.

For example, there is some confusion as to the proper unit for a cyclomatic unit test. Some developers think it should be the DATA step, the inherent unit of the language. Others think it should be a macro, the unit that corresponds to design modules. Unit tests at the DATA step level will result in many tests of potentially very simple logic, with no testing of the complexities that can arise from the interaction of DATA steps. Unit tests at the macro level may result in units of excessively high cyclomatic complexity and corresponding tests of overly complex logic.

Cyclomatic theory would probably suggest testing at the macro level. If a macro is of excessive complexity, the theory would suggest that the macro be modularized further. Consider, though, a macro that consists of two moderately complex DATA steps that interact with one another, and suppose that the macro is truly coherent, i.e. that it really does only one thing. If the complexity of the macro overall exceeds the generally accepted complexity limit, cyclomatic theory says that the macro should be broken into smaller modules. The macro is already submodularized, though, into the two DATA steps. If each DATA step alone is comprehensible, it might be better to leave the macro intact. That leaves the unpleasant prospect of cyclomatic testing on a very complex module.

One solution is to compromise in the selection of the test unit. This may require judgments and decisions that would not be needed in a test of a traditional language program. The guideline would be to test at the macro level if the overall complexity is within bounds. Otherwise, related DATA steps could be packaged into ad hoc test units of reasonable complexity, although the code itself would not be altered. Sometimes, as in the example above, the package might be a single DATA step. This method has been used successfully in the CPI program.

Another possible solution is to de-emphasize the testing described above and to concentrate on those aspects of testing that the SAS system facilitates, such as boundary testing and functional testing. This approach has also been successfully used in the CPI program.

Maintenance - Advantages of the SAS System Generally

The SAS system facilitates maintenance because, at least in base SAS software, the code is highly readable. The power of the language and the wide range of built-in features make it possible to code much smaller programs than would be required in other languages. The English-like syntax and the inherent modularity of PROC and DATA steps also enhance understandability. Maintainability is also increased by the descriptive nature of SAS software code with respect to file handling, and the fact that several operations that must be coded explicitly in traditional programming languages are done automatically in the SAS system. In addition, to the extent that the SAS software code mirrors the design, the associated DATA and PROC steps corresponding to identified design processes can be lifted and replaced when maintenance is required.
The power and flexibility cited above for the SAS Macro facility have an associated cost in the maintenance phase. Complex SAS macro code is often difficult to understand and maintain. The full workings of applications written using Macro are often obscure to the novice, and the relationship between the code and the design becomes obscured. Many users find all but the simplest SAS macro code difficult to read. In fact, often the only feasible way to determine what SAS code will be generated and executed is to run the program and examine the SAS log. Even then, understanding is hindered because much of the careful code structuring, including indentation and provisions for statements that span more than one line, is lost in the code printed by MPRINT. Even experienced users are challenged by the complexities of certain features such as referencing environments and quoting in the Macro facility. As such, there are concerns regarding the understandability of complex macro applications and their maintainability in the face of changing needs. These factors have lengthened the time required for software development, redevelopment, and testing.

The problem is not that the SAS Macro facility is inherently difficult. Like the rest of the SAS system, the SAS Macro facility is easy to learn and use for routine applications. The problem arises when sophisticated applications such as those in the CPI production system have to be implemented. Even this is not a problem if only base SAS software is used, because code written in base SAS software remains straightforward even when the code becomes complex. The straightforward nature of the base language disappears, however, when working with complex applications written using the Macro facility.

One way to mitigate these maintenance problems is to discourage tricky or complicated use of the Macro facility in coding. This same technique applies to development in those traditional languages that foster tricks and elegant but incomprehensible code. Of course, this restriction takes away some of the power and flexibility of the language, so each installation has to make a judgment on the proper balance of power vs. maintainability.

Automated System Development Tools

The divergence of the SAS system and structured systems engineering is particularly evident in the lack of automated software development tools that are applicable to building systems using SAS software. This affects several phases of the life cycle, primarily coding, testing, and maintenance. Automated tools such as code generators based on design, complexity analyzers, testing tools, and code restructurers are language specific, and the commercial versions of these tools do not support code written using SAS software. This is a major obstacle for developers who want to go beyond tedious and unreliable manual techniques in creating, testing, and maintaining large-scale traditional systems.

STRUCTURED SYSTEMS ENGINEERING AND THE SAS SYSTEM - VERSION CHANGES

Version changes in the SAS system have been time consuming to implement in the CPI production systems. The problem is that SAS versions are not completely upwardly compatible.

In some applications, the effect of SAS versions changes may be minimal. Limited duration systems or ad hoc programs may outlive their usefulness before conversion to a new version becomes mandatory. Systems of relatively low complexity may require few changes. Non-critical systems can afford to be tested live under the new version and can be converted as needed when needed.

On the other hand, the impact of SAS version changes on the CPI production system is considerable. The system is long-range, so its lifetime spans version changes. It's also highly complex, and its operation is time critical. As such, SAS version changes require that every program in the production system be thoroughly inspected, modified, and tested before being run in live production. In addition, the BLS-written SAS procedures have to be inspected, modified, and thoroughly tested. If an installation uses any in-house software development tools that are specific to the SAS system, such as code generators, these too have to be modified for version changes. The CPI production system even has PL/I programs that generate SAS software applications code on the fly, so these PL/I programs have to be modified for version changes. In the CPI program, these conversion efforts have required and continue to require a considerable investment in time and personnel.

One natural recruiting technique is to hire experienced SAS users. However, many of the individuals who are skilled in the use of the SAS system use it primarily for applications other than traditional data processing systems. For example, the SAS system is widely used for data analysis, report generation,
database and other data management applications that do not require intensive computation, graphics applications, and user interfaces. The skills required to develop these systems are considerable, and the range of SAS system products used for these applications is sizable, but they are not necessarily the skills and products that are essential for the development of a traditional data processing system like the CPI program system. As such, when attempting to recruit experienced SAS developers, CPI program managers have found that even highly experienced candidates often lack experience in complex DATA step processing, macro processing, large-scale development, structured programming techniques, analysis and design, and structured testing methods. 

Another way to recruit might be to hire developers who are experienced in the full life cycle, even though their coding experience might be in a traditional programming language. This route usually works quite well. Even excellent programmers, though, sometimes have trouble adapting to the SAS system's different processing paradigm. Even when they have become proficient at programming using SAS software, some qualified developers who started with another language report that they never became comfortable with the SAS system's paradigm.

Those candidates who are experienced in both the SAS system and structured systems engineering are often familiar with them as two separate worlds. For example, such candidates often know structured systems engineering using some traditional programming language, and they know the SAS system for data analysis and report generation. These candidates are, of course, well ahead of the game, but they are still lacking specific experience in the use of the SAS system for large-scale development in a structured environment.

This problem is even more evident in recruitment at the entry level. Many of the individuals hired in the systems development offices of the CPI program are recent college graduates in disciplines such as computer science and information systems. Use of SAS software to develop large-scale production systems or even small-scale data processing applications is generally not taught in the computer science and information systems curricula. Many new hires in the CPI program have no experience or education in the SAS system at all.

In the CPI program, because of the difficulties in hiring the precise mix of necessary abilities, expertise is developed through training. SAS Institute training is available to develop SAS software expertise, and the CPI program has availed itself of the basic courses. Agency sponsored training is available for structured systems engineering. These training options do not include specific training in the use of the SAS system for large-scale development, so that has been covered in-house, usually informally. Permanent application development backlogs have prevented development of a comprehensive in-house SAS system training program or a formalized, widespread mentoring program. Even the mix of training and informal mentoring described above is expensive and time consuming. Fortunately, the basics of the SAS system are quite easy to learn, and the system is easy to use, so qualified developers become proficient more quickly using SAS software than they do using other languages.

None of the foregoing reflects any deficiency in the SAS system itself. More likely these problems stem from the fact that the SAS system is not the dominant computer language, and it is not widely used for large-scale traditional data processing applications that are developed in a structured systems engineering environment.

CONCLUSIONS

SAS software is the cornerstone of mainframe software architecture for the production processing systems of the CPI. The choice of the SAS system for software architecture is still regarded as fundamentally sound. The CPI program probably will continue to use SAS software for future modifications and new development, and the program should be increasingly able to take advantage of products and features beyond the base system. There have been some problems in using SAS software in a structured systems engineering environment. Therefore, system development using SAS software has not been as easily accomplished as was thought when SAS software was originally chosen as the implementation language for this system.

Nevertheless, with skillful adaptation, SAS software can be made to work quite well in a structured systems engineering environment.

ACKNOWLEDGEMENTS

The authors wish to acknowledge all of the employees of the Bureau of Labor Statistics and contractors who provided assistance in the development of this paper. They include, but are not limited to, the following: Carl J. Lowe, Assistant Commissioner, Office of Technology and Survey Processing; John Sinks, Director, Directorate of Survey Processing; Stephen G. Wright, Division Chief of Consumer Prices and Consumption Studies (CPCS), and CPSC managers Karen Hassemer, Richard Schroeder, Leanna McEnearney, Donald Jones, and Edward Pratt, all of whom gave of their time to review this paper and to offer their comments; our colleagues Ellen Dory, Peter Lorenzetti, Larry Goldman, David Chen, Cheana Hsu, and Arthur Yao, who cheerfully discussed their impressions and experiences and who provided many valuable ideas and insights; and Paul Armknecht, Assistant Commissioner for Consumer Prices, and Marybeth Tschetter for their review on behalf of the CPI program office.
AUTHOR CONTACT

The authors may be contacted at:

Bureau of Labor Statistics
2 Massachusetts Avenue N.E. Room 5935
Washington, D.C. 20212

DISCLAIMER

The opinions presented herein are solely those of the authors. Impressions and conclusions reached do not necessarily reflect the views of the agency.

REFERENCES


Johnstone, James and Rabb, Merry and Sharlin, Joshua. "The Use of the SAS System in the Development of the Production System for the Consumer Price Index (CPI)," SUGI '86 Proceedings, SAS Institute, Cary, N.C.


SAS and SAS/Toolkit are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration. IBM and 3090 are registered trademarks or trademarks of International Business Machines Corporation.

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