Paper 219-2008

CSI: San Antonio - Common SAS® Issues in Our Programs and Tips for Better Investigation of Your SAS Code

Rachel Brown, Westat, Houston, TX
Jennifer Fulton, Westat, Houston, TX

Abstract

Good documentation of the issues discovered when SAS programs are developed and validated can provide a wealth of information to help improve programming over time. In 2004, our SAS programming department implemented a new documentation process, whereby we record program tracking information that includes every change request made for each program. This change control documentation has resulted in an extensive database providing valuable information to help improve the way we write and check programs. The authors defined 10 categories to summarize the variety of comments in the change requests. Over 3,000 records from 14 projects were then categorized to compile a "top ten" –style list of the most common issues found during the development and validation of our SAS programs. How will our list compare with the kinds of issues you frequently uncover when checking programs? We'll share some of the change requests from our database, exposing the good, the bad, and the downright comical. And we will compliment this list with another: our checklist of ten items to remember when validating SAS programs. We hope by sharing our mistakes and providing ten ounces of prevention, we can help you avoid these Common SAS Issues too!

Setting the Scene

"Those who do not learn from the mistakes of history are doomed to repeat them." - George Santayana. In the programming world, the words of this famous quote ring true. Every experienced programmer knows it is rare to impossible to write a program perfectly on the first try. But good documentation of the various issues discovered when programs are developed and checked can provide a wealth of information to help improve programming over time. With these famous words in mind, we decided to take a look back at the various issues that arose during validation of hundreds of programs written by programmers in our department since implementation of our current validation practices to see if we could uncover any trends.

Westat is a multiservice research corporation, and our department in Houston is part of the clinical trials division. Hence, we are required to meet strict FDA regulations and standards. This includes producing documentation that tracks the development and validation of all SAS programs. In 2004, our SAS programming department implemented new methods to better facilitate these requirements. For each project, we record the name of the program developer, the date on which initial program development was completed, the name of the program validator, the date of validation, all change requests that come up during validation, and developer responses to the change requests. While this change control documentation has not been flawless, it has resulted in an extensive database of records which provides valuable information to help improve the way we write and check programs.

Methods of Investigation

In an effort to summarize the large amount of available validation data, we first compiled the change control documents for 14 different projects into one large spreadsheet. We selected projects for which programming was initiated after the new record-keeping process was implemented, and selected only one project if there was a group of related projects (such as two studies by the same client for the same drug and indication). Using May 18, 2006 as the cutoff date, we included all studies that fit our criteria, even if the project was ongoing. We then defined ten categories to summarize the variety of comments in the list of change requests. The categories were based on our programming group's experience with the types of issues seen most often, and on some of the other internal training we had received on checking our programs. We categorized over 3,000 records to compile a "top ten" –style list showing the most common and least common issues found during the development and validation of SAS programs within our department.

The ten categories, which will be explained in more detail in the next section, are:

1. Unacceptable Documentation
2. Inconsistencies with Project Requirements
3. Problems in the Log
4. Spelling and Typographical Errors in Tables, Graphs, and Listings
5. Problems Related To Calculations

6. Incorrect or Inefficient Coding or Errors in Program Syntax
7. Problems with Formatting in Tables, Graphs, and Listings
8. Data Omitted From an Analysis Dataset
9. Data Issues
10. Comments That Did Not Require a Change

As we proceeded to categorize the 3,000+ entries, we soon learned that our data had some additional characteristics that should be addressed. We found, for example, that even with the pre-defined categories, many comments were subject to interpretation and could be categorized more than one way. Since the two authors had split the chore of categorizing the comments, we wanted to verify that we had some consistency in our interpretations. Each author randomly selected 50 records from the other author's section and assigned categories a second time. Results were compared, and we found that 80% of the time the categorizations matched. Given the difficulties of categorizing free-text comments, we felt that this level of agreement provided adequate assurance that our list would capture the most common and least common issues validators were finding. We also considered the possibility that certain projects may tend to have a disproportionate number of one specific category. This could be due to unusually messy data, a particularly picky client, or other issues specific to a project. We felt that since we had 14 projects included in our database, such cases would be balanced by the unique intricacies of other projects, and the project-specific information could be useful for future work with those same clients or types of clients.

Finally, we compiled a list of ten helpful tips for "investigation" of SAS code that will aid in uncovering the ten issues that we categorized. We hope that these tips provide a simple list of helpful reminders that will improve the level of quality in writing and checking SAS programs. We solicited several of our most experienced programmers for their thoughts on what to look for when checking SAS programs. We hope that these tips, in conjunction with our list of the most common programming issues recently encountered, will provide information that can help all levels of programmers in a variety of industries write better SAS programs.

What the Clues Tell Us

Taking a look back at our history of program validation comments has provided some useful insight into the common issues our programming team faces. Do you recognize your most common issues; those that come back to haunt you every time you work on a program? Maybe our review can give you some clues. Some of ours come as no surprise given the nature of our work in an FDA-regulated clinical trials industry. Here is what we uncovered[1], beginning with the least common issue and working our way toward the most common issue.

#10 – Spelling and Typographical Errors in Tables, Graphs, and Listings (1.1%)

Some medical terminology that is difficult to spell may be a prime source of spelling errors. Such issues in the data fall within the "Data Issues" category and are usually corrected by our Data Management department, but spelling and typographical errors in titles, column headers, and footnotes are programmer problems. Examples of change requests due to spelling or typographical errors are:

- *"Volume is misspelled in the table header."*
- *"Term is misspelled in the title."*
- *"Add a space between 'Adverse' and 'Experience' in the footnote."*

#9 – Problems in the Log (1.4%)

Perhaps the most apparent programming issues are those identified in the SAS log. This includes the problematic presence of any of the following keywords in a SAS log: "error", "warning", "uninitialized", "invalid", "missing", "w.d", "repeats of", "unreferenced", "resolved", or "outside". Had there been an abundance of comments falling into this category, we would have been concerned about the work being done in our group given that "check the log" is a basic rule of programming. Fortunately, we had very few of these, and some of them may be related to unexpected data that was not in the database at the time of program development. As a general rule, programs should not contain certain keywords that result in comments such as:

- *"CXNAME not resolved."*
- *"WARNING: Truncated record in log."*
- *"There are a few other errors and warnings in the log, which cause SAS to stop running."*

[1] Some comments have been minimally modified to remove names, confidential or sensitive information, and to correct spelling and grammatical errors.

#8 – Data Omitted From an Analysis Data Set (1.7%)
Since creating analysis data sets comprises a large part of our programming work, a category specific to an analysis data problem is warranted. In our programming process, we often take SAS data sets extracted from Oracle Clinical® and/or other external data sources, and create analysis data sets according to industry recommendations. These data sets should include all of the information needed to program reports, and all data should be cleaned and formatted. Data omissions from analysis data sets can lead to less efficient programming when creating reports, and in some cases, may even be the source of inaccurate output. Examples of comments related to data omissions are:

- *"On the CRF, there is a Number 13 under Clinical Assessment. But it's not in the data set."*
- *"The variable ECGND is needed for the listing. However, it is not in the analysis data set."*
- *"Shouldn't all the masterindex variables be kept in the final data set?"*

#7 – Unacceptable Documentation (3.4%)
Like many companies, our group uses a standard program header in every program to detail the project name, program name, program description, program developer, and log of changes to the program. Additionally, comments are included throughout a program to explain SAS code and reference sources containing program assumptions and changes. Emails or other documents requesting program changes or defining assumptions that are not reflected in requirements should be cited in program code. Every programmer benefits from good comments and explanations in programs. Having these types of comments presents any programmer the ability to open another programmer's program and understand what has been done. Examples of comments regarding documentation include:

- *"Please document how baseline is selected, i.e. one record or average of prior dose values."*
- *"Modify header for program name and output files."*
- *"Add a comment to the top of the program that explains that although the mock table displays z-score, we don't have this data so were told not to display it per an email from [sender name] to [recipient] on 8/5 called '[email subject line]'. I'll save the email to the network."*

#6 –Problems Related to Calculations (4.2%)
While an incorrect calculation can also be called a coding error, we elected to treat this issue as a separate category because of the nature of the output we normally create. Programming with clinical trials data consists primarily of two components: calculations and presentation of the results. Hence, looking at calculation problems separately adds the value of giving us an indication as to whether programmers have more trouble analyzing the data or displaying the results in an appropriate format. Some examples include:

- *"The calculation of CALCCRCL should be corrected."*
- *"Numbers and percentages are not adding up right."*
- *"Change denominators to be the number of treated subjects instead of only treated subjects with AE data collected."*

#5 – Data Issues (4.9%)
Data that was incorrectly entered into the database and illogical data can be the source of inaccurate statistics or inappropriate formatting. Looking for data issues while programming in SAS gives our data management team an additional check of the data, especially for variables that may not be considered key variables and may not have many logic checks programmed in Oracle Clinical to identify data anomalies. Programming edit checks in SAS code assists in identification of problem data. Most data issues are handled by data management staff; however, there are occasions where data issues have to be handled in SAS code. Programmers are better off trying to anticipate potential data issues and programming more dynamic code from the beginning. These are some data issues we found:

- *"There is a male patient with a valid pregnancy test date in the data. Is this a data problem?"*
- *"One record shows a GLUCOSE UR QUALITATIVE result of 100mg/dL. It seems like the result should be NEGATIVE, POSITIVE, etc.*
- *"Patient 9999 has a temperature of 98.6 C."*

#4 – Comments That Did Not Require a Change (7%)
Sometimes, a note or reminder may be entered as a comment, or a comment may be made asking a developer to check into or confirm something that does not require any programming updates. Programmers may use the change control documentation as an area for dialogue regarding a program, or as a place to enter reminders about future program enhancements or updates. Even better, there are the occasions, when a program (usually a simpler program) is problem-free. In these cases, a line is still entered in the change control document to indicate that the program was validated and no problems were reported. These types of comments are consolidated into one category. Examples of such comments include:

- *"no change requests"*
- *"I sent an email saying that the variable RDTXGPNM should be changed to ACTXGPNM in safety tables and listings. This is just a reminder that this change does NOT apply here since this is not a safety listing."*

#3 – Incorrect or Inefficient Coding or Errors in Program Syntax (18.6%)
These are problems with code that do not specifically relate to calculations or formulas. Often, SAS code is not as efficient as it could be or the programming logic is flawed. An inefficient line of code could be made more efficient by using a different SAS function, a potential category may be excluded from a series of If-Then-Else statements, or problems with logic may produce inaccurate results. These problems can sometimes be concealed while having dire effects on output. While spelling errors, formatting problems, and data issues can often be apparent, a problem with syntax could create no flags in a SAS log while producing incorrect results in a statistical summary table. Uncovering these types of issues often requires diligence on the part of program developers and validators. Some examples of the typical change requests we have recorded regarding syntax problems are:

- *"In the third data step in section 7, the code to deal with the duration overlap problem has a logic problem which will lead to the wrong output."*
- *"Find ways to make this program more efficient so it takes less time to run."*
- *"Codes for the variable OUTCOME are not assigned. Assign and use the code specified in the footer."*
- *"For RACE, the CRF says 'check all that apply'. Currently, the data shows only one response per patient. But possibly in the future, if a person were to check more than one race, the current code will not work."*

#2 – Inconsistencies with Project Requirements (25.8%)
In clinical trials work, project requirements may include the protocol, statistical analysis plan, mock output, or other instructions from our clients. Inconsistencies with these could be a result of programmers not following the provided requirements, inadequate or incorrect requirements that lead to inconsistencies and questions about how to program things, or requirements that were not updated as needed to reflect changes since their initial development. Having a lot of these types of changes can be an indication that more time and care are needed during the planning phases and at the start of programming. Maintaining updated requirements is a necessity in order to avoid excessive change requests based on outdated instructions. Examples of changes related to project requirements include:

- *"Listing title does not match the mock listing."*
- *"Missing the 3rd and 4th footnote."*
- *"The value E in the table is shown with both 1 and 2 decimal places. The mock table shows 1 decimal place."*
- *"Change "Stop Date" to "End Date" to match the CRF."*
- *"According to the protocol, lab visits are done for the first 5 months, and then every 3 months thereafter. So for the first 5 months, the table is fine. But based on the mocks we need to display Month 6 on in 3 month increments."*

#1 – Problems with Formatting in Tables, Graphs, and Listings (31.9%)
These are issues related to the visual appeal of SAS output. Since project requirements usually do not go so far as to define font sizes, column widths and alignment of text within a column, many of these decisions are left to the judgment of program developers and validators. The nature of our programming requires that SAS output be formatted as cleanly as possible so that it can be inserted into formal study reports or even used as a report itself. For this reason, it is not surprising that the majority of all programming issues recorded in our change control documents were included in the category for formatting issues. Some of the typical change requests we have recorded regarding formatting are:

- *""Increase the width of the last column to 16 so that text doesn't wrap."*
- *"Add date in the last column so that data looks like MM/DD/YYYY (RELDAY)."*
- *"In footer 5, put the code REDUCED in all caps."*
- *"Sort labs in the order that they are displayed in Sec. 6.2.1 of the protocol."*
- *"When converted to word, the output wraps around to the next line."*

The fact that this category topped our list reminds us that, like it or not, clients really do care about the appearance of output as much as the content. It also brings to light that getting the format just right is a difficult programming feat. "The numbers" that we get from calculations and syntax are most important, but our emphasis on clean, professional output requires that detailed attention always be paid to the overall appearance of our output.

Upon Closer Inspection

From the start of this exercise, we expected to learn from our mistakes, but we discovered an additional unexpected benefit… a little humor along the way. Comments are entered freely into designated columns, resulting in a variety of commenting styles. Some change requests are long and detailed, some are short and to the point, some are sugar-coated, and some are down-right blunt. A few of our favorites are listed below in true top-ten style:

1. The nicest comment:
   "The visits for urinalysis are only "visit 1", "visit 2"… without "Week x". Please reformat the visit in the output. Please list out all the visits in the data."

2. The easiest-to-fix comment:
   "Add a period to the end of the footnote regarding the use of MedDRA."

3. The most rambling comment:
   "The 3rd programming note on the mock is confusing. The interpretation for this listing was to not include any screening records, and to not include records that are 'Not Done' or 'Not Changed'. That's a lot of NOTs!... The listing excludes screening records as well as any post-screening record that is 'Not Done' or 'Not Changed'. I think the note may really mean to say that we should exclude post-screening records that are Unchanged, hence including all screening records plus post-screening records that are 'Changed' or 'Not Done'. In the validation area, I played with this so there is code if you want to look. This may need to be clarified with [name] since I've probably just tied your brain in 'NOTs' :)"

4. The most cryptic comments:
   - *"In the Proc SQL, one of the on conditions should be a.windown – 12 <= b. windown <= a.windown to match the request??"*
   - *"?? A p-value<0.0005 is deemed statistically significant"*

5. The most detailed comment:
   "Add a space between percentile estimates and the CI. This results in several other changes... Extend length of ESTIM to account for this extra space. Then after the Proc Transpose, the length of TRT1 and TRT2 in TRANQUART1 is 15. When you set TRANQUART1 with the other data at the end of the program, TRT1 and TRT2 take the length from another data set. So put TRANQUART1 first in the list of data sets to set so that the longer length is kept and results are not truncated. Change column widths in the output macro call for the first 2 columns and right justify so that parentheses line up a little better. Add T_ORDER as a second by-variable for sorting in the output macro call and add S_ORDER as a skip variable in the output so that lines don't skip due to the new by-variable."

6. The wimpiest comment:
   "Warning message in the log: 'WARNING: Data too long for column VOLDELI_.' This is only from a Proc Print, so maybe it doesn't matter. Anyway, I just presented it to you."

7. The most ironic comment:
   "Measeurements is mispelled in title."

8. The most blunt comments:
   - *"Start over - Copy this program from aecrt.sas and subset for Serious events."*
   - *"You need comments in your programs (nag nag!) - I put some in for you in the version in the validation area so that you can see what I mean."*

9. The funniest comments:
   - *"It bugs me that there is a 2 and a Part II and .2 in the title, plus in all other listings that have "parts" we do not number them as .1 and .2, and they are all in one program. Can we get rid of this program and copy the code into txadm_crt.sas and just have Part I and Part II like other listings?"*
   - *"Major Amputation is defined as below or above the knee only."*

10. The greatest comment to have for your program:
    "No Change Requests"

We will continue adding to the types of comments described above, and as our group continues to follow and improve upon the current process of documenting program change requests, we can have a better idea of where our strengths and weaknesses lie. This awareness can be a valuable aid in training as programmers are encouraged to focus on commonly reported problems as well as pay attention to those less often reported.

Tips for Better Investigation of Your SAS Code

Now that we have counted down our most common SAS programming issues, we would like to offer ten tips to help programmers do a thorough job of reviewing programs and feel confident about the final product.

1. Give the validator a separate copy of the program.

The validator should have a separate area on the network from the area in which development takes place where he/she can perform checks without the risk that the original program will be affected. A different directory or network drive will suffice, as long as it precisely mimics the area in which the program was developed. This area should also contain copies of exactly the same data files that were used when developing the program, as well as any other files such as external macros or spreadsheets that are necessary to the function of the program. The program in the development area should be "locked" in some manner such that it will not change at all until validation is finalized. In this way the validator can "play" with the program code if he/she chooses, and perform checks in the preferred manner without risk of adversely affecting the original program. "Locking" the original program can include making the file read-only and recording a development sign-off date which does not change and is always prior to the

validation sign-off date. Note also that if the validator requests any changes to the program after one round of checks, the program should again be copied to this validation area where the changes are checked in the same manner for every round of validation.

2. Check program documentation.
The level of documentation expected in a program will differ in various industries and environments. At a minimum the validator should confirm that the program has some style of header that records any general information that might be useful to future users. Some examples are: program name, output file name, input file names, external macros used, client/project for which the program was written, and a description of the function of the program. The validator should also review the comments entered within the SAS code. The program is well-commented if the validator can follow the intentions and meaning of the code without ever having to ask the developer for further explanation. If there are formulas in the code, or other code that was written in response to a request from a manager or client, these should be clearly documented with comments so that the validator can look them up and verify that the code is correct. A good litmus test for comments is to imagine if an employee several years down the road has to look at this code without anyone around who was originally involved in writing or validating it. Would this new programmer be able to explain to the client why half of the data was excluded from the table?

3. Check the SAS log.
This concept is nothing new, but it is such an essential part of validation that it is worthy of mentioning again. The validator has to look at the log and he/she has to look for more than just errors. Some examples of strings to search for are: “error”, “warning”, “uninitialized”, “invalid”, “missing”, “w.d”, “repeats of”, “unreferenced”, “resolved”, “outside”. The goal should be a “clean” log, because even a seemingly innocuous item in the log can hide a serious problem in the code, or a problem that will be worse if the data changes. Occasionally a note or even a warning in the log is acceptable because the validator and developer agree that the results are not affected and it is not worth the time to program around it just to clean up the log. In these cases it can be helpful to future users of the program to insert a comment in the code explaining that this issue in the log is expected and does not adversely affect results.

4. Review the requirements for the program.
Generally this means that the validator should be provided with a copy of the same set of documentation from which the developer worked when writing the program. This may be a formal statistical analysis plan, a memo from the boss or, an e-mail from the client. If a mock-up of how the output should be presented was provided, the validator needs a copy of this as well. It is the validator’s job to verify that the program does what it was supposed to do – in other words that the developer correctly interpreted the assignment. The validator must not assume that the developer was always right or knew what was needed. A thorough validator should question the developer’s coding decisions if he/she does not feel the program meets the requirements.

5. View the output in its final (deliverable) form.
The validator may be tempted to only look at the output in interactive SAS or in an ASCII editor. But while the output may pass the validation with flying colors in its raw format, it may have a new set of problems that are only observed when the file is converted to a Word document, for example. The validator should see what the final recipient will see to avoid unexpected (sometimes embarrassing) surprises. In its final form, the validator should review how the output fits on the page, page breaks, page numbering, wrapping of text, etc. And spelling can be easier to check in a word processor. Graphical output can even be affected by different print drivers, so the validator should print a sample of the output to verify that colors are as expected or symbols are not skewed.

6. Does the output make sense?
The validator should take the point of view of the client, manager, or other reviewer who will ultimately look at the output with little or no understanding of the intricacies of the data or the SAS programming that went into it. The combination of titles, footnotes, and column/row headers should clearly and completely convey what is contained in the output so that the output can stand alone – not requiring that the developer come with it to explain what is displayed. Some other things to look for: Can the reviewer easily tell what is being tested by the placement of the p-value or with a footnote? If the sum of the individual columns does not equal the total column, is this explained by the title or a footnote? Do the row labels, column headers, or graph legends clearly explain how the data are being grouped? Is it clear from looking at the numbers which value was used as the denominator in percent calculations, and was this an appropriate choice? Are units printed wherever appropriate? The validator should also make sure all footnotes have a reference within the corresponding output, and that they make sense for the given table.

7. Check consistency.
Consistency is important within a single output file as well as across all output files for a project. Some examples of items that should match from one output to another include: the sample size (any discrepancies should be explained or fixed unless the reason for the differences is easy to see, such as in the case of a subset analysis); terminology (for example, if "Heart Rate" is written on the listings, "Pulse" should not be in the tables); the general structure of output; and formatted variables (one common example of inconsistency is to print date variables with different date formats in two or more places). Consistency can be a more subjective item to evaluate, but the validator should be sure to point out anything that he/she feels strays from the standards of the rest of the output for the project.

8. Look at the code.
Besides reviewing the program documentation, the validator should also read through the logic of the program. Below are some specific suggestions for reviewing the code.

All formulas should be verified with the appropriate reference by the validator.

Anything that alters data or creates new data should be checked, such as if-then clauses, data subsetting, merges, or transposes. Note in particular that all if-then-else clauses should take into account "exception values" which are unexpected values that may appear in the data. The program log can be useful in determining if the number of observations is correct after each data step.

Macro language can present an entirely different set of issues to evaluate, and of course, the more complex the macro, the more possibility for error. If there is any macro language, trace through the logic and be sure the code is in place properly, and get an idea of what was intended by the programmer. It can be helpful to run one macro call at a time and check results. The options MPRINT and SYMBOLGEN can be particularly helpful in reviewing how the data is affected throughout the macro itself.

SQL code can be used for many purposes, so the validation of SQL code depends somewhat on how it is used in the program. It can be used simply for efficiency, because it may take the place of several data steps. In this case a validator could write the code the "long way" and see if the results match.

Some suggest that the validator should not look at the program code until after he/she has checked as much of the output as possible using raw data sets and other sources. This helps minimize any influence (conscious or otherwise) that may be made on the validator by seeing the way the programmer approached the program. If the validator can come up with the same results in a truly independent manner, it is very likely that the results are correct. But even if independent checks produce the same results as the original program, it is still imperative that the validator thoroughly review the actual program since there could be syntax problems that would create incorrect results given updated data.

9. Check the results.
It does not matter how nice the output looks if the data and analysis results presented therein are not correct. All of the validation steps are important, but getting this one right is crucial. Here are some suggestions for approaching this step:

The validator should compare the data in the output to the most raw form of the data available. For example, if a listing contains data from several different data sets merged together, the validator should not refer to earlier data sets within the program being tested, but rather look at the individual raw data sets as provided by the client or entered by data management. If the output is simple enough, the validator can simply view each of the raw data sets and select the same individual record in each and then compare to the output to see if the data matches appropriately. In the case of very large data sets, the validator may want to subset the data and do checks on random selections.

For more complicated programs, a good way to check numbers is for the validator to write his or her own mini-programs, and check the output in pieces as appropriate. The key is for the validator to keep his or her own mini-programs simple, so the validator comes as close as possible to comparing the output to the original raw data. If the results of the validator's program and the programmer's program do not match, the validator should figure out why, and alert the programmer if a change is necessary.

When checking graphing programs, check not only the actual data points that appear in the graph, but also the range of the data to ensure that the ranges of the axes do not cause any data to be eliminated from the graph.

One important item often overlooked by the validator is the percentage calculations included in many tables and sometimes other output. It is not sufficient to assume that because the counts are correct, the accompanying percentages are also correct! The validator should check if the correct numerators and denominators were used in percent calculations, if the number of decimals displayed in the percentage is consistent across all output, and whether the percentages add up to 100 where appropriate. If the percentages do not sum to 100, an explanation of some kind is warranted on the output.

Cross-checking between various forms of the output can be important. For example, often a graph is simply a different way to present the same data in a corresponding table, so the two types of output should have the same results. Also listings are frequently created to accompany tables, so the validator may be able to use one to check the other. It can be a helpful practice to include in the list of all project programs a column which indicates which listings support each table and/or graph, so the validator can easily do the appropriate comparisons.

Make sure values are formatted appropriately. Make sure that the correct number of decimal places is consistently displayed, and "missing" or "not done" codes, or other programmer-defined formats are appropriately displayed in the output.

When checking the program after change requests have been implemented, it can be helpful to save the old version of the output with a different file name (such as [filename]_old.sas), and compare the old and new versions to be sure that only the items that should have changed did change.

When the population is split into subgroups or different categories (such as different treatment groups, categories of the outcome variable, etc.) be sure that all items in the original population are accounted for. In other words if the sum of the parts does not equal the whole, a correction to the program or an explanation (such as a footnote) is warranted.

10. Keep good documentation of the validation process.
Certainly the level of detail required for this step varies greatly depending on the industry for which the programming takes place. At a minimum, a record of the developer name and sign off date, the validator name and sign off date, and a summary of the changes/problems that were issued during the course of validation can be a valuable tool. This documentation can help programmers learn from past mistakes, and provide a level of accountability for who did what when – validating the checking that was done. In our department where we are strictly regulated by the FDA, we have implemented a System Development Life Cycle (SDLC) by which we keep an even more detailed accounting of the steps each program goes through from inception to final delivery of the output to the client. If you are interested in learning more about our SDLC documentation process, contact Ken Gerald at *KenGerald@westat.com*.

Mystery Solved!

Finding out our department's most and least frequent programming issues provided some insights that we can use to improve in several areas. We found that we spend the majority of our time making our output look clean and professional, as well as correcting mistakes that resulted from misinterpreting or not reading the instructions provided. And according to our results, we are doing well in the areas of minimizing spelling and typographical errors and problems in our SAS logs. We can use the countdown to consider if we should shift our focus in any way by asking ourselves questions such as:

- Do we worry TOO much about "looks"?
- Should we invest more time early in the project on mock-ups and programmer instructions?
- Should we place more emphasis on the importance of reading and following requirements in regular training?
- Is additional training needed to remind programmers of good programming practices to keep program syntax clean?
- Are some of the less frequent issues actually underreported?

Whatever your most common SAS issues may be, we suggest that all of the ten categories of issues need to be addressed by developers and validators alike, and that keeping this information at hand and following the guidelines for investigating all of the issues on this list will ensure that a comprehensive check is done for all programs.

Acknowledgments

We would like to thank our coworkers for providing us with hundreds of comments from their program validation, and for their feedback that helped us put this paper together. We would also like to thank our supervisors who allowed and encouraged us to put together and present our findings at SAS Global Forum.

Contact Information

Your comments and questions are valued and encouraged.

Rachel Brown, MS
Westat
5615 Kirby Drive, Suite 710
Houston, TX 77005
Work Phone: 713-353-7916
Fax: 713-529-4924
Email: RachelBrown@westat.com

Jennifer Fulton, MS
Westat
5615 Kirby Drive, Suite 710
Houston, TX 77005
Work Phone: 713-353-7925
Fax: 713-529-4924
Email: JenniferFulton@westat.com

Disclaimer
The contents of this paper are the work of the author and do not necessarily represent the opinions, recommendations, or practices of Westat.