PharmaSUG 2016 - Paper TT05

Generalized Problem-Solving Techniques for De-bugging and Diagnosing Logic Errors

Brian Fairfield-Carter, inVentiv Health, Cary, NC

Tracy Sherman, Chiltern International, Wilmington, NC

ABSTRACT

The most troublesome code bugs are those that seem to elude rational diagnosis: these are the 'logic errors' in syntactically-correct programs, where the SAS® log offers little or no insight. More disturbingly, these crop up in programs in which we otherwise place reasonable confidence, convinced as we are of the logic behind our initial efforts.

Syntax errors are almost always easy to spot, as they are the primary focus of built-in de-bugging features, and also enjoy a wealth of documentation to aid in diagnosis and correction. When it comes to logic errors, however, we're very much on our own: logic errors are rarely captured by built-in de-bugging features, and discussion of strategy seldom extends beyond general recommendations like "fully understand your data" and "carefully review output". Common software-testing paradigms, which assume compartmentalization and the separation of 'interface' and 'implementation', also tend not to fit well in the analysis-programming world, meaning that instead of the application of any sort of formal or systematic de-bugging strategy, what we often see is an ad hoc 'brute force' approach to bug and logic error diagnosis.

This paper offers, as an antidote to brute-force problem-solving, a generalized error-trapping strategy (co-opting ideas behind 'software regression testing', combined with sequential functionality-reduction), supported by simple technical solutions tailored to analysis programming. Application of this strategy is illustrated in a few case studies:

- Trapping unintended artifacts of code revision & adaptation
- Problems in rendering RTF and XML output
- Mysterious record- and value-loss in data derivation

INTRODUCTION

Logic Errors and Code De-Bugging

Literature on the subject of code de-bugging largely falls into two categories: understanding and coping with errors in syntax, and recognizing when syntactically-correct code performs unintended operations (what we often refer to as 'logic' errors). Syntax errors are the easiest to spot and correct, are almost always flagged by automated de-bugging interfaces and log reporting, and seldom permit the generation of erroneous output (since they tend to just cause programs to fail to execute), but ironically also tend to be the best-represented in the discussion of de-bugging. Logic errors are far more insidious and difficult to spot, are the usual source of erroneous output, are less likely to be captured by de-bugging interfaces, and are also where discussion tends to get thinner and vaguer.

Take for example the SAS institute's "Debugging SAS Programs: A Handbook of Tools and Techniques" (http://support.sas.com/publishing/pubcat/chaps/57743.pdf), which gives a fairly comprehensive run-down on errors and warnings written to the log and then devotes just a couple pages at the end to the topic of "What Errors Doesn't SAS Find for Me?" The limited representation given this topic is of course understandable, since the potential for unintended application of syntactically correct code is essentially infinite, but what a prescription like

> ***"How can you find the logic errors that you introduce into your programs? The answer is by fully understanding your data, closely reading the messages in the SAS log, carefully reviewing the results, and using the tools that SAS provides for detecting logic errors."***

really highlights is the need for formal strategy, or a systematic approach, in trapping logic errors. 'Fully understanding your data' and 'carefully reviewing results' are no small tasks, and require the careful and organized application of techniques and tools in order to be realized in a project of any size or complexity. The most troublesome and time-consuming problems generally fall into that 'SAS doesn't find for me' category, and at odds with the complex nature of these problems, problem-solving approaches in the SAS analysis-programming world are often disorganized and ad hoc. Creativity is required for solving novel or unexpected problems, but given the wide range of possibility often presented by logic errors, this creativity must be directed in an organized fashion.

Does Analysis Programming fit common Software-testing paradigms?

In casting around for suggestions on formal strategy for testing and de-bugging, one might start by considering the general (and ubiquitous) software-testing categories 'Unit testing', 'Integration testing' and 'Regression testing'. The MSDN (Microsoft Developer Network) website (https://msdn.microsoft.com/en-us/library/aa292197(v=vs.71).aspx) offers a definition and brief discussion of each, and definitions can be summarized as:

Unit testing

Takes the smallest piece of testable software in the application, isolates it from the remainder of the code, and determines whether it behaves exactly as expected.

Integration testing

Two or more units that have already been unit-tested are combined into a 'component' (aggregate of units) and the interface between them is tested.

Regression testing

Attempts to uncover new bugs (regressions) in existing systems after changes (updates, enhancements, patches) have been made.

The question is, do these present a reasonable analogy for the kind of testing and bug-fixing we tend to do in analysis programming, and in particular in the context of really problematic logic errors? Partly it depends on environment (i.e. whether or not projects are amenable to the use of 'standard macros' or libraries of utilities, which in turn requires a certain degree of standardization across projects), and partly it depends on what you consider to be the 'smallest piece of testable software'. An informal review of conference papers discussing Unit testing for SAS code suggests that the concept is generally applied to systems and applications, where interface and implementation are separated in the manner of (i.e.) a Windows application; circumstances under which a given 'unit' operates can be defined/dictated by the interface, which (rather conveniently) acts as a constraint on the range of possible conditions.

In contrast, consider the linear and essentially single-purpose programs that create analysis data sets: source data and analytical rules are highly project-specific (and source data is generally governed by highly variable data-quality constraints), and even where common data-manipulation themes arise, individual project budgets don't provide for the development of really robust cross-project utilities. Attempts to compartmentalize functionality into a macro library invariably results in a collection of macros that are, at worst, unintelligible to all but their original authors or, at best, require at least some project-specific modification. The development of 'boilerplate' code, that can be treated as a testable 'unit' and can literally be used unmodified across a diversity of projects, is simply not practical within the constraints (budgets and timelines) of individual projects, and this in turn means that 'unit testing' in the context of analysis programming is neither practical nor necessary. And if the 'unit testing' concept doesn't really fit well with analysis programming tasks, then by extension, integration testing presents the same quandary.

So if the concepts behind unit and integration testing don't really fit nicely with analysis programming, how about regression testing? Consider what happens when derivation rules are revised, or when QC discrepancies prompt revisions to derivation code: output needs to be checked to confirm that intended changes did in fact take place, and that no unintended changes (artifacts) arose, and this sounds strikingly like the intent behind regression testing (to "uncover new bugs (regressions) in existing systems after changes (updates, enhancements, patches) have been made"). You might say that when it comes to dealing with logic errors, the 'fluid' and evolving nature of source data and requirements, combined with the constraints within individual projects, kind of flips the usual software-testing paradigm on its head: the proactive implementation of test cases is de-emphasized (essentially limited to localized 'defensive code' designed around expected or anticipated conditions), while greater emphasis is placed on the more reactive regression-testing that is employed when source data or requirements change, or when counter-intuitive results appear because of unexpected conditions.

We might conclude then that from among the Unit/Integration/Regression concepts, its 'Regression testing' that offers a reasonable conceptual framework that we can apply in the world of analysis programming.

What tools are available to aid in de-bugging and logic-error trapping?

Correcting syntax errors requires an understanding of programming grammar. Finding and correcting logic errors on the other hand requires an understanding of program- and data-flow, or the sequence of events that actually take place during program execution. An understanding of syntax will tell you if statements are likely to execute, but not whether the resulting data manipulation is actually appropriate given analytical objectives.

Arguably there are aspects of 'interactive mode' in SAS, supplemented with the DATA step de-bugger (if available), that resemble de-bugging features in an 'integrated development environment' such as Visual Studio: its possible to step through code one statement at a time, 'watch' data transformation by following specific variables and values, and suspend execution at designated events. Supplemental diagnostic code can be executed as needed, and various

things (attribute changes, multiple merges, etc.) can be checked retrospectively in the SAS log. Interactive mode is helpful in understanding the effect of 'co-located' program statements (i.e. those within a single DATA step), but what remains difficult without a truly interactive de-bugging environment is understanding the contributions of 'remote' sections of code to a given derivation: visualizing these large-scale relationships generally involves laboriously reading through programs in their entirety, and tossing in PROC PRINTS here and there to try and figure out 'what is going on'.

Especially in complex and/or 'inherited' code, this 'brute force' approach is very time-intensive, and it would be helpful to have some automated way of producing a summary or overview of data transformation steps within a program (i.e. even just a list of DATA steps, with source and output data set names and record counts). SAS Enterprise Guide offers a 'Process Flow' which, to some extent, captures these relationships visually, but it is not particularly customizable, and is obviously only available if you have SAS Enterprise Guide deployed. The limited discussion of user-developed 'code analyzers' seems to fall into three streams: (a) prospective generation of process-flow information by parsing SAS code, or by augmenting SAS code with a pre-processor, (b) concurrent generation, using output from PROC SCAPROC, or (c) retrospective generation by parsing the SAS log. For each of these a certain amount of work is required, and no method is without draw-backs and limitations.

SAS metadata tables such as SASHELP.VCOLUMN also offer powerful insight into the relationship between data transformation steps: they do not provide for a convenient summary of DATA steps and input/output data sets at each step, but they do allow for certain post-hoc investigation on (i.e.) chronologically-ordered lists of work data sets, which is often helpful in tracing the role of specific variables and values in a derivation.

A general and systematic approach to problem-solving

Programming is all about problem-solving, and we might think of the process of implementing the analyses described in a stats plan as being a large and real-world instance of the kind of word problem that we used to solve in elementary algebra. The difference is that when solving an algebraic word problem, we tend to be more aware of the strategy behind the solution, and recognize the discrete steps involved:

- Problem definition: state the given quantities and conditions (i.e. the input, paraphrased for clarity, and eliminating spurious information).
- Definition of unknown: state the unknown as a function of known quantities and conditions.
- Translate to suitable notation
- Solve for the unknown

Whether or not we formally recognize these steps, they describe much of what we do when writing analysis code. For example, counts of randomized patients, membership in analysis populations, and occurrence of disease progression provide 'quantities and conditions', and a cumulative assessment of incidence of progression is the unknown that we state as a function of known quantities and conditions. Translating to suitable notation involves assigning known and unknown quantities to variables, with 'solving for the unknown' taking place via syntactically-correct data-manipulation steps.

Now the problem is that we can go through these steps, and still end up with incorrect results; logic errors often occur in the very operations where we think we've already expended our greatest efforts at logical construction. This means it's usually futile to try and solve logic errors by simply repeating the same problem-solving steps. What we need is an analogous set of steps to follow when our initial attempts at problem-solving have failed; we need a methodology for trapping exactly the errors in logic that escaped our initial efforts, and the absence of this methodology is often why logic errors are tackled in a time-costly manner.

We might summarize the tenets of such a methodology as:

- Implement 'regression testing': compare pre- and post-revisions results, and check against expectations (specifically that intended changes did in fact occur, and that unintended changes did not).
- Implement 'functionality-reduction': reduce code to a functioning 'kernel' (the largest collection of correctly-functioning statements), and then progressively re-introduce statements until something goes wrong.
- Apply an experimental approach: find causal relationships by allowing only one factor to vary at a time (i.e. only test code updates on a stable data snapshot, and only check the results of a new data snapshot on stable code); run code against test data deliberately constructed with 'extreme' cases.
- 'Reverse-engineer': determine what the 'correct' answer *should* be, by hand-calculation on a sub-set of raw data if necessary, and see what program statements will 'reverse engineer' this correct answer.

Since strategy on its own is not particularly helpful, and since tools are seldom effective unless guided by strategy, the following section provides a concrete illustration, using some common problems that crop up in analysis programming.

APPLICATION OF DE-BUGGING STRATEGY AND TOOLS, BY EXAMPLE

Regression Testing

The life cycle of analysis programs can be roughly divided into code development, and code maintenance/revision, the former characterized by the accumulation of code statements, and the latter by (relatively) limited changes to existing code, either in response to QC/validation or to revisions to analytical objectives. Changes to code during maintenance/revision provide the perfect opportunity to apply regression testing, both to confirm intended changes and to defend against unintended artifacts and new errors.

The term 'regression testing' might sound grandiose, but the tools and techniques involved do not need to be fancy or sophisticated at all. Consider an instance where membership in a given analysis population requires that a particular assessment falls after the date of first treatment of investigational product, but the specification as initially written indicated "asmtdt>=fdosedt" rather than "asmtdt>fdosedt". Senior review of the analysis data set identifies 3 patients that should be excluded from the population, since asmtdt=fdosedt, and the production programmer goes ahead and makes updates to use ">" rather than ">=", over-writes the analysis data set, and declares the problem solved. The QC programmer then finds that while the three patients are now correctly excluded from the population, asmtdt is also now (*incorrectly*) missing on the analysis data set for the three patients in question.

Regression testing on the update would have avoided this QC discrepancy: by (prior to over-writing the analysis data set) running PROC COMPARE against the original data set and the new/updated data set, the artifact would have been easily spotted:

```
%macro write_(overwrite=N);
  %if &overwrite=Y %then %do;
    data analysis.mydsn;
      set mydsn;
    run;
  %end;
  %else %if &overwrite=N %then %do;
    data original;
      set analysis.mydsn;
    run;
    proc compare base=original compare=mydsn;
    run;
  %end;
%mend write_;
%write_(overwrite=N);
```

Following this very simple approach, the production programmer, prior to setting &overwrite=Y, would have confirmed that the three patients in question were, as expected, now excluded from the population, but would also have noticed that an unintended artifact had cropped up, namely that values that were previously present were now missing.

Functionality-reduction

Problems with complex systems are almost always impossible to identify by looking at the system as a whole; things need to be broken down into discrete and manageable components before root cause can be discerned, and the most important part of this 'deconstruction' is in identifying fail points. The point to functionality reduction is to reduce demands on a non-functioning system until it works, and then to re-introduce specific functional components and/or input, to try and isolate the factors that contribute to the fail-point.

For example, imagine you've created a script or macro intended to operate on a series of files, perhaps to collate them into a single document. You develop this for a specific project, and then turn it over for adaptation and use in a different project, where it is run blindly on a collection of 400 files and promptly bombs. Users then complain that the utility "doesn't work". In de-bugging the program, you probably won't run it over and over again on the same set of files hoping that eventually the bug will just go away, nor will you endlessly read through the code hoping to miraculously spot a logic error, but you might instead follow a logical series of steps:

1. Run the utility on a set of test files that you know worked previously, and confirm that it still works
2. Compare a sample of your test files with a sample of 'new' files, and see what differs between them; if there are no obvious differences that can be translated to code revisions, then…

3. Run the utility on a very small subset of 'new' files
4. Progressively increase the set of 'new' files until the fail-point is reached
5. Comment out and then re-introduce functional elements in order to isolate that which is actually failing (i.e. if the script generates a file list and then performs a series of actions like 'read/open', 'copy/paste', 'save as', start by confirming that the file list is generated correctly, and then that the script can successfully open all files in the list).

Hopefully what these steps will do is to help you determine what is unique or special about the conditions right at the fail point (i.e. the nature of a specific file where things bomb, or some aspect of file count or cumulative file size), and the specific functionality that is causing problems.

Functionality-reduction is really just an extension of regression testing, since the fail point you are trying to identify is really the point where 'regression' occurs. Although regression testing is essentially reactive, the conditions under which regression occurs can be controlled and manipulated in an experimental way.

This 'reduction' approach is particularly useful when tackling problems involving 'un-renderable' output. Take for example statistical summary tables output as RTF, HTML or XML: each of these file types consists of simple text, containing data as well as instructions to a target application on how the data is to be displayed, but the structure and syntax of these instructions may well be completely unfamiliar to you. But unfamiliar or not, problems in the output may make it impossible for the target application to open or render the file, and it may fall to you to try and identify the source of the error.

To illustrate, let's say you're using ODS Tagsets.ExcelXP to generate output as Excel-readable XML:

```
ods tagsets.excelxp file="myfile.xls" style=styles.sasweb options(…);
   proc report data=final ...;
     --etc.--
   run;
ods tagsets.excelxp close;
```

You run the program, the output file is generated, and the SAS log is clean. Yet when you go to open the file in Excel, you get a 'Problems During Load' dialog that reads "***Problems came up in the following areas during load:/Table/This file cannot be opened because of errors. Errors are listed in: C:\Users\...\Content.MSO\B5428339.log.***".

Excel is unable to display (render) the XML file, and neither the SAS log nor the dialog provide any helpful information; the dialog refers to a log file which it *claims* provides a list of errors, but this log file is actually inaccessible (or possibly non-existent). If you're lucky, a review of the SAS code producing the table might yield something obvious, but attempting to locate the problem by reading the XML file as simple text is usually a fruitless exercise, meaning that some strategy for isolating the offending piece of the XML file needs to be employed.

As in the previous example, a content- and functionality-reduction approach can be applied to try and trap the 'regression' (fail) point: either remove elements one at a time, or reduce the output to something very simple and then add content back in. For instance, you might start by commenting out titles and/or footnotes, and reducing the number of records (rows of output) being written to the output file, and even reducing the number of columns (i.e. comment out variables in the PROC REPORT 'COLUMN' statement). Eventually you'll have a reduced output file that Excel is able to open, and this should give you a general idea of where the problem is located.

Having localized the problem to some extent, quite often the cause becomes intuitively obvious, but there are also tools and techniques available to more precisely highlight potential issues. For instance, imagine you remove footnotes and discover that the XML output is now renderable, and find that the addition of a single/specific footnote causes the output file to fail. Since XML is simple text, files generated with and without the problematic footnote can be compared using 'diff' utilities such as ExamDiff (http://download.cnet.com/ExamDiff/3000-2248_4-10059626.html):

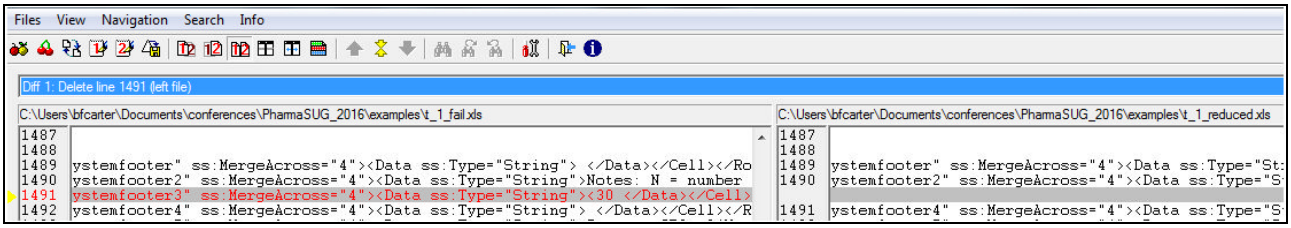

Figure 1. Side-by-side comparison of non-renderable and renderable XML output, using ExamDiff.

Since we can now see in the XML file exactly where the problem occurs, and can identify the string '<30' as the content causing the error, we can actually manually edit the XML file (i.e. in Notepad) to see what is required to get the file to open correctly in Excel. In this example, a quick Google search indicates that a '<' symbol, since it functions as an XML control character, needs to be replaced by the escaped string '<':

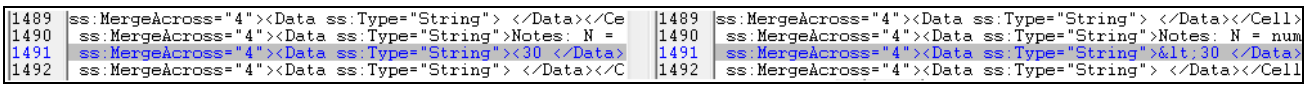

Figure 2. Renderable output (on the right) produced by manually editing non-renderable XML (on the left).

Once we know the correct form of this footnote, which will allow the file to render correctly in Excel, we can go back to SAS and reverse-engineer the correct 'escaped' character. This is a fairly simple example (it may seem somewhat remedial), but it does illustrate how problems even in unfamiliar code such as XML can be systematically isolated and debugged, and that the 'creative' aspects of problem-solving work best when they are focused on a localized fail-point, rather than on the system as a whole.

Similar types of problems can arise with RTF output (especially when updates are made to a legacy reporting environment, and/or where programmers aren't familiar with ODS RTF nuances and character escaping), and similar problem-solving techniques can be applied. As an example, consider the following instance: underlining and bold text are applied to column headings in an RTF table by means of in-line RTF code:

```
proc report ...;
  column ...
     (' \b  Treatment Group \brdrb\brdrs\brdrw1' _1 _2) _3;
```

, where '\b' instructs Word to display 'Treatment Group' in bold text, and '\brdrb\brdrs\brdrw1' to display a lower border for the table cell:

Table 15.1.1.2 Summary of Major Protocol Deviations
ITT Set

System	Treatment Group Treatment A (N=66)	Placebo (N=60)	Total (N=126)
Number (%) of subjects with at least one major protocol deviation	4 (6.1%)	7 (11.7%)	11 (8.7%)
IP Non-Compliance	3 (4.5%)	6 (10.0%)	9 (7.1%)
Other	1 (1.5%)	3 (5.0%)	4 (3.2%)

Figure 3. Summary table, with formatting (bold text and lower cell border) provided via in-line RTF.

Unknown to the table programmer, inserting 'un-escaped' RTF in this way only works because the style template being applied includes an instruction for headers to *not* protect (escape) special characters in the generated RTF:

```
style header from headersandfooters /
  protectspecialchars=OFF (--etc.--)
```

'protectspecialchars=OFF' means that RTF control code is written verbatim to the RTF output, meaning that Word interprets it as a rendering instruction rather than as literal text. If the style template is updated with 'protectspecialchars=ON' (i.e. another table requires that characters like '\' and '{' are displayed as text), and the next time the above table is updated, what appears is:

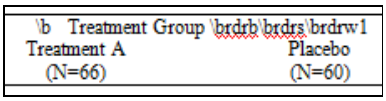

Figure 4. In-line RTF rendered as literal text as a result of protectspecialchars=ON

The difference between the original and current versions (and the reason that Word renders things the way it does) is easy to see in ExamDiff:

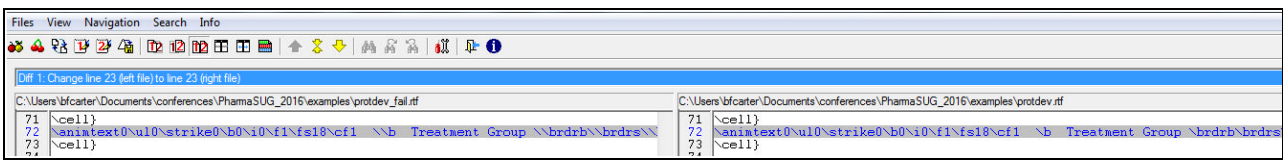

Figure 5. Escaped and non-escaped RTF control code (left side displays in Word as literal text)

In other words, the instruction to protect special characters means that SAS 'escapes' RTF control sequences '`\b`' (bold text) and '`\brdrb\brdrs\brdrw1`' (lower border) as '`\\b` ' and '`\\brdrb\\brdrs\\brdrw1`', which Word then handles by displaying the characters after each 'escape' as literal text.

Handling problems with escaped and non-escaped RTF control characters can be tricky. A stray '{' character in a listing (i.e. originating on a free-text field in a CRF) can result in a (not particularly helpful) dialog that says "***Word was unable to read this document. It may be corrupt. Try one or more of the following: *Open and Repair the file. * Open the file with the Text Recovery converter.***". The '{' character is the proverbial 'needle in a haystack', but can be systematically isolated through exactly the same sort of content-removal steps.

A good rule of thumb when it comes to inserting in-line RTF code though is to never make assumptions about whether 'protectspecialchars' is set to ON or OFF, but instead to always inform SAS about your intentions:

```
proc report ...;
  column ...
    (' `R/RTF"\b"  Treatment Group `R/RTF"\brdrb\brdrs\brdrw1"' _1 _2) _3;
```

The `` `R/RTF `` string (where "`" is assumed to be the ODS escape character) tells SAS to always treat the quoted text that follows as an RTF control sequence rather than as literal text, regardless of whether or not 'protectspecialchars' is set to ON or OFF. (Alternately, the 'protectspecialchars' setting from the style template can be over-ridden via in-line formatting, such as `` `S={protectspecialchars=OFF} ``).

Tracing program- and data-flow

Logic errors in data derivation often play out on a large scale, over multiple data-manipulation steps, and require an understanding of program and data 'flow'. Developing this understanding can be troublesome with complex derivations, especially if implemented in code that has been through multiple authors. So how do we take an entire and possibly unfamiliar derivation program and acquire a workable image of program and data flow? Specifically, if we consider a given computed value on a derived data set, how do we trace its origins (source data set, intermediate WORK data sets, variables, transformations)? The answer, unfortunately, is 'with some difficulty': in spite of the obvious value of detailed run-time information, it's surprisingly difficult to acquire; the resources available to supplement a brute-force 'study the code' approach are really pretty thin.

To start with, probably what we're after is some sort of 'code analyzer' that will help in visualizing the relationships between different parts of a program, and between (i.e.) intermediate WORK data sets. As mentioned previously, SAS does offer a scattering of features to (at least partially) serve this function: the DATA step de-bugger, PROC SCAPROC and the SAS Enterprise Guide 'process flow' are all intended to help visualize and understand program events and data manipulation, but none is without limitations.

Before considering either run-time or 'retrospective' approaches, we might look at the 'prospective' approach of automating code analysis: simply read a program in as data, and apply a set of text-manipulation rules to extract references to data sets and variables. Aside from the complexity involved in identifying step boundaries (i.e. consider the number of ways in which a data set might be referred to in program code), there are a number of problems with this approach: with commented-out sections of code and macro branching, it can be exceptionally difficult to anticipate what actually gets executed and (in the case of macro branching) in what order. Further, this approach will provide no information on things like changes to values of specific variables during program execution, or on record count at each DATA step.

Some of these problems can be overcome by applying the 'retrospective' approach of parsing the SAS log: by reading the log and identifying data set references, we can extract the names of input and output data sets at each DATA step, as well as record count at each DATA step, and we can also (to a limited extent) infer step type (DATA/PROC, concatenation, merge, SQL). Further, since the information is gleaned retrospectively, after program execution, we can tell what steps actually executed and in what order, in spite of macro branching or commented-out code.

The principle behind log-based code analysis is very simple: we exploit the predictable way in which the log reports on DATA steps, in order to capture information on input and output data sets. For instance, a simple DATA step will be reflected in the log as

```
NOTE: There were <n> observations read from the data set <library>.<dataset>.
NOTE: The data set <library>.<dataset> has <n> observations and <v> variables.
```

Since the phrases '`NOTE: There were `' and '`NOTE: The data set `' are used consistently (and are fairly specific), they can be used to identify log lines giving library and data set names, and record counts, for input data sets, and those giving library and data set names, record counts, and counts of variables for output data sets. Note

that these lines in of themselves will not distinguish between a simple DATA step and a PROC step (i.e. the lines shown above will look the same if generated by PROC UNIVARIATE or by a DATA step).

Steps involving multiple input (and/or output) data sets can be similarly identified:

```
NOTE: There were <a> observations read from the data set <library>.<dataset>.
NOTE: There were <b> observations read from the data set <library>.<dataset>.
      WHERE <conditions>;
NOTE: There were <c> observations read from the data set <library>.<dataset>.
      WHERE <conditions>;
NOTE: There were <d> observations read from the data set <library>.<dataset>.
      WHERE <conditions>;
NOTE: The data set <library>.<dataset> has <n> observations and <v> variables.
```

From this we can capture the names of four input data sets, and the record counts on each, and the name, record count, and variable count on the output data set. If the record count on the output data set is equal to the sum of record counts on the input data sets, then we *might* assume that this was a concatenation step (and otherwise we *might* assume that it was a merge step); while this assumption is not necessarily reliable, it can sometimes help in identifying specific steps in the program code.

With very little basic text manipulation (sample program available from the PharmaSUG web site, or from the authors on request), the log from a derivation program can be condensed to a simple step summary:

INPUT_	INOBS_	OUTPUT_	OUTOBS_	OUTVARS_	STEP_TYPE
RAW.DM	181	WORK.RAW_DM	181	18	DATA/PROC
WORK.RAW_LB	365	WORK.CREAT	365	31	DATA/PROC
WORK.CREAT	365	WORK.CREAT	147	32	DATA/PROC
WORK.RAW_DM	181	WORK.DM	181	17	DATA/PROC
RAW.DS	648	WORK.DS	648	21	DATA/PROC
WORK.DM|WORK.DS| WORK.DS|WORK.DS	181|129|181|0	WORK.DM	181	22	MERGE
WORK.DM	181	WORK.DM	181	24	DATA/PROC
WORK.DM|WORK.CREAT	181|147	WORK.DM	184	48	MERGE
WORK.DM	184	WORK.DERIVED_DM	184	10	DATA/PROC

This gives a chronological summary of the key data-manipulation steps in a derivation program, and in spite of being somewhat simplistic there are a number of things we can learn: the raw DM data set starts with 181 observations but, after being merged with the 'CREAT' data set (possibly supplying baseline Creatinine values, since it originates from the RAW_LB data set and undergoes a record-count reduction) increases to 184 records. This means that lab data is the culprit in causing some unexpected record-count inflation (possibly indicating something as simple as a multiple merge, or alternately that non-enrolled patients have been included on the lab data set), and highlighting the merge step where code modification may be required. We can also see (row 6) that three subsets of disposition (DS) records are merged with DM, probably capturing things like randomization date, discontinuation date, etc., and that the data set 'DM' gets replaced on three occasions (this raises a bit of a warning flag, namely that tracing alterations to specific records on DM will require that 'generation' data sets be used, probably by adding 'GENMAX=3' the first time WORK data set 'DM' is created).

There are a couple of fairly obvious limitations to the information drawn from the log: first, the log file needs to be up-to-date (there's no guarantee that the program was not modified more recently than the last time the log file was generated), and second, if you wanted information on specific variables or attributes, they would have to be captured in a post hoc step. (For instance, if you wanted to find the point in a succession of WORK data sets where the length of a particular variable had gone from $200 to $30, you'd have to re-run the program and capture attribute information while the data sets of interest were available in the WORK library). PROC SCAPROC (described in SAS documentation as 'implementing the SAS Code Analyzer') perhaps offers a 'concurrent' solution that circumvents these issues: an output destination is opened at the start of the program, and closed at program completion, and captures detailed information on each input and output data set at each step:

```
proc scaproc;
  record "scaproc_out.txt" ATTR OPENTIMES;
run;
  proc sort data=raw.dm out=dm;
    by usubjid;
  run;
  --(etc.)--
```

```
proc scaproc;
  write;
run;
```

Output, written to 'scaproc_out.txt', would look something like this:

```
/* JOBSPLIT: ATTR RAW.DM.DATA INPUT VARIABLE:USUBJID TYPE:CHARACTER LENGTH:18
LABEL:Unique Subject Identifier FORMAT: INFORMAT: */

--(etc.)--

/* JOBSPLIT: ATTR WORK.DM.DATA OUTPUT VARIABLE:USUBJID TYPE:CHARACTER LENGTH:18
LABEL:Unique Subject Identifier FORMAT: INFORMAT: */
```

As you can see, 'input' and 'output' data sets can be readily identified at each step, and the 'ATTR' option requests attribute information for the full list of variables on each data set. (Absent, however, is any information on record count.) With a bit of work, SCAPROC output could be reduced to a step summary comparable to one generated by parsing the log file (minus record count info), and could be used to highlight conflicting attributes (for example, when a given variable exists on more than one data set at a merge step, but the attributes of that variable (type, length, format) are not consistent).

One fairly common problem in data derivation arises when the same variable exists on more than one data set in a MERGE step, and the variable is *not* a key variable in the merge: values from the first data set listed in the MERGE statement are over-written by those from the last data set listed, and this can lead to some highly counter-intuitive results. While SCAPROC can list out all variables present on each data set, it does not distinguish between key and non-key variables at a merge step, meaning that it does not provide much help in diagnosing this error. The best way remains via the use of 'options msglevel=I', which results in an 'INFO' line being written to the log for each non-key variable where over-writing takes place:

```
INFO: The variable AGE on data set WORK.DM will be overwritten by data set
WORK.CREAT.
```

Another fairly sizeable draw-back to PROC SCAPROC is that it really only works well with open code; DATA steps generated by in-line or external macros result in SCAPROC output that is hopelessly disordered, and make it virtually impossible to reconstruct the chronological sequence of steps that actually took place during program execution. The SAS log, in contrast, perfectly reflects this chronological sequence of events, regardless of whether the program consists of open code or a complex series of macros.

Cross-sectioning derived data using SAS metadata

Although the step-summary (shown above, generated from the SAS log) gives a rough overview of 'data flow', and has the potential to highlight things like unexpected changes in record count, it does not tell us anything about specific data values or records (i.e. where specific values get altered), and this is where it can be useful to use SAS metadata to trace through (in chronological order) the data sets in the WORK library.

For example, imagine a derivation that starts with a calculation which is then 'augmented' in subsequent DATA steps: the initial calculation results in VALUE=123 for a given patient, but the final derived data set ends up with VALUE=127. Somewhere between the original derivation and the final data set the value has been altered, but the question is where? Rather than adding a PROC PRINT after every DATA step, we might 'cross-section' the derivation by getting a list of WORK data sets (ordered chronologically) containing the variable 'VALUE', and see for each data set what value this variable contains (for the target patient). First we'd create an ordered list of WORK data sets containing the variable 'VALUE':

```
proc sql;
  create table dsn_list as select memname, crdate from sashelp.vtable
    where upcase(libname)="WORK"
    & memname in(select distinct memname from sashelp.vcolumn
                 where upcase(libname)="WORK" & upcase(name)='VALUE')
    order by crdate;
quit;
```

, and then iterate through this list, running a PROC PRINT on each data set (subset by the focal patient) and displaying just the variable 'VALUE':

```
proc sql;
  select memname into :dsnlst separated by '|' from dsn_list
  where memname^='dsn_list';
```

```
quit;
%let i=1;
%do %until(%scan(&dsnlst,&i,'|')=);
  title "----- %scan(&dsnlst,&i,'|') -----";
  proc print data=%scan(&dsnlst,&i,'|');
    var usubjid value;
    where usubjid='123';
  run;
  %let i=%eval(&i+1);
%end;
```

The same approach can be taken for diagnosing a variety of similar problems: if we want to know where in a derivation program the length of a given text variable goes from (i.e.) $200 to $30, we simply capture data set name, variable name, and variable length in a chronologically ordered list of WORK data sets containing the given variable, and see where the length value changes:

```
proc sql;
  create table dsn_list as select memname, crdate from sashelp.vtable
    where upcase(libname)="WORK"
    order by memname;
  create table dsn_len as select memname, name, type, length
    from sashelp.vcolumn where upcase(libname)="WORK" & upcase(name)='VALUE'
    order by memname;
quit;

data temp;
  merge dsn_list dsn_len(in=_2);
  by memname;
  if _2;
run;

proc sort data=temp(keep=memname crdate length);
  by crdate;
run;
```

Or if we wanted to know where a specific record got dropped (i.e. a given patient being dropped out of an analysis population, where the population definition applies several criteria that are tested sequentially, over several DATA steps), we could again start with a chronological list of data sets, iterate through the list and for each data set check for the presence of a record for the given patient.

Example

The following simple example provides a composite of some fairly common but potentially baffling issues resulting from logic errors in derivation programs, and shows how some of the techniques described above might be applied. In creating an ADSL derived data set, you might imagine starting with a raw (or SDTM) DM (demography) data set, incorporating baseline covariates such as baseline Creatinine, capturing randomization, informed consent and death dates from a DS (disposition) domain, and deriving things like age, age group, and flags for patients with baseline covariates falling within a particular reference range.

These tasks all sound pretty straight-forward, so it might be a bit surprising to be presented with validation issues like:

- Three 'extra' patients on ADSL
- Discrepancies on AGE (missing and conflicting values, and records on the ADSL data set where age is missing but age group is populated).

Resolving validation issues invariably demands matching record counts, and the first issue ('extra' patients on ADSL) can be fairly easily dispensed with; recall the step summary developed from the log file:

```
INPUT_               INOBS_         OUTPUT_           OUTOBS_  OUTVARS_ STEP_TYPE
…
WORK.DM|WORK.CREAT   181|147        WORK.DM           184      48       MERGE
WORK.DM              184            WORK.DERIVED_DM   184      10       DATA/PROC
```

The increase in record count appears at the merge between WORK data sets 'DM' and 'CREAT', so we can go straight to that point in the ADSL program and see if (a) we need to remove duplicate baseline Creatinine records prior to the merge, or (b) filter the output data set to only include patients from the 'DM' input data set.

Discrepancies related to AGE might seem baffling at first: a check of the raw data shows that all patients provide a birth date and randomization date, and the age calculation itself seems pretty simple and fool-proof. Further, a print of the WORK data set right at the AGE calculation shows that all patients end up with an AGE value, so it's pretty clear that AGE is being over-written in some subsequent step. To determine the exact location in the program, two options exist: first, set 'options msglevel=I' and look for a note like this in the log:

```
INFO: The variable AGE on data set WORK.DM will be overwritten by data set
WORK.CREAT.
```

Alternately, use SAS metadata to create a chronologically ordered list of data sets containing the variable 'AGE', iterate through the list and print out the AGE value from each data set for one of the patients showing a discrepancy on AGE. This might seem cumbersome compared to just setting 'options msglevel=I', but is useful to explore since not all issues that involve unintended changes to data values involve over-writing during a merge step. Note that as indicated by the step summary, the 'DM' WORK data set is over-written several times, which presents a bit of an additional challenge in this kind of post-hoc processing: in order to print data from each 'version' of the DM data set, it is first necessary to declare 'generation' data sets at the first point the DM data set is created, using the 'GENMAX' option:

```
data dm(GENMAX=10);
  set raw_dm;
run;
```

Individual versions can then be referenced using the following syntax (though be forewarned, the relationship between generation number as appended to the data set name and generation number ('gennum') used to reference a given version is fairly confusing, so it's worth spending a bit of time reviewing the documentation at http://support.sas.com/documentation/cdl/en/lrcon/62955/HTML/default/viewer.htm#a000934566.htm):

```
proc print data=dm(gennum=1);
run;
```

In the present example (sample code available from the PharmaSUG web site, or from the authors on request), we find that WORK data sets in the following list (chronologically ordered by CRDATE) contain the variable 'AGE':

```
MEMNAME             CRDATE          GEN

RAW_DM          03MAR16:15:48:40
RAW_LB          03MAR16:15:48:44
CREAT           03MAR16:15:48:46
DM#003          03MAR16:15:48:48      2
DM              03MAR16:15:48:49      3
DERIVED_DM      03MAR16:15:48:50
```

Adjusting MEMNAME to contain the appropriate GENNUM reference, we get:

```
MEMNAME               CRDATE          GEN

RAW_DM           03MAR16:15:48:40
RAW_LB           03MAR16:15:48:44
CREAT            03MAR16:15:48:46
DM(GENNUM=3)     03MAR16:15:48:48      2
DM(GENNUM=0)     03MAR16:15:48:49      3
DERIVED_DM       03MAR16:15:48:50
```

In other words, the version referenced by MEMNAME='DM#003' (GEN=2) is the older version, which resulted from 2 replacements of the original version (and is referenced in PROC PRINT as 'DM(GENNUM=3)'), while the version referenced by MEMNAME='DM' (GEN=3) is the 'current' version (referenced in PROC PRINT as 'DM(GENNUM=0)').

Tracing through this list and checking the AGE values for a specific patient, we can see where the value was altered:

```
----- RAW_DM -----
     USUBJID    AGE
     01-002     43
----- RAW_LB -----
     USUBJID    AGE
     01-002     43
----- CREAT -----
     USUBJID    AGE
     01-002     43
----- DM(GENNUM=3) -----
     USUBJID          AGE
     01-002           44
----- DM(GENNUM=0) -----
     USUBJID          AGE
     01-002           43
----- DERIVED_DM -----
     USUBJID          AGE
     01-002           43
```

The variable 'AGE' exists on the raw demography data set, but is then re-derived on the WORK.DM data set at GENNUM=3; the variable 'AGE' also exists on the raw lab data set, and over-writes the derived value at the merge step between DM (GENNUM=3) and CREAT, meaning that the current version of DM (GENNUM=0) contains the value from the lab data set rather than the derived value.

CONCLUSION

We're not well served by 'higher education' if all we do is to accumulate raw information; the point behind a University education is to develop analytical ability to systematically confront novel problems. The same is true of programming: knowledge of SAS syntax does not on its own serve an 'analysis programmer' role, particularly in this era of rapidly-changing project teams. Far more important is the ability to analyze programming problems and isolate causal factors behind unexpected output.

They say that when looking for a needle in a haystack, it’s a good idea to bring a strong magnet. Code bugs and logic errors are the 'needles' we're often confronted with in analysis programming, but the available 'magnets' (built-in de-bugging features and diagnostics) are actually fairly weak. Solving novel and unexpected problems demands creativity, but this creativity must be focused in an ordered and structured way in order to be effective. This order and structure are in turn provided by the application of systematic approaches to problem-solving, where things like 'regression testing', 'sequential functionality reduction', and the use of step summaries and 'metadata cross-sectioning' to trace program and data flow, provide us a 'magnet' of the necessary strength to trap even the most troublesome code bugs and logic errors.

ACKNOWLEDGMENTS

We would like to thank our employers inVentiv Health and Chiltern International, and managers Colleen Benjamin and Robert Diseker in particular, for providing encouragement and supporting PharmaSUG participation, as well as all our family and friends and colleagues.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Name: Brian Fairfield-Carter
Enterprise: inVentiv Health
E-mail: fairfieldcarterbrian@gmail.com