Sounding the Trumpet: Effective Failure Notification

Don Hopkins, Ursa Logic Corporation, Durham, NC

ABSTRACT

A typical data management system is a complex collection of manual and computerized procedures. Operational dependencies among the various parts are often intricate, and there are many things that can go wrong. When something does go wrong, it is often essential to catch the failure quickly to avoid a cascade of costly downstream problems. In a system that employs SAS programs, one way to find out about system failures is to scan the SAS log window after a program executes. Manual scanning of the SAS log may be all that's needed for a simple program, but as a system grows in complexity, manual scanning may cease to be a reliable or efficient technique for noticing problems. Scanning is a vigilance task—something humans are not particularly good at—and the probability of overlooking a critical failure message increases when programs are long and complex, when they are executed as part of a routine sequence, or when they are executed in batch mode with no one in attendance. Fortunately, there are alternatives to manual scanning. This paper describes programmatic techniques that are useful for capturing failures in SAS programs and alerting those who need to know about them.

INTRODUCTION

Careful review of the SAS log by a knowledgeable reviewer is usually all that is needed to determine if a SAS program performed as intended and produced the desired results. However, as a strategy for monitoring the performance of a complex system over time, manual review of the log may not always be feasible, practical, efficient, or effective. Some systems need to be more assertive in checking for performance problems and bringing these problems to the attention of a responsible party.

Ironically, it is the most reliable systems that may be most in need of assertive performance alerts, because a track record of reliable performance tends to cause system users to become less vigilant. Instead of waiting quietly for someone to come along and review the log, a complex, reliable system should sound a loud trumpet (so to speak) when something goes awry.

This paper describes a system for embedding performance checks in Base SAS programs and actively delivering the results of these checks to a designated recipient. The results are reported in a convenient format that makes it difficult to overlook when a problem has occurred. The system was developed at Ursa Logic to facilitate performance monitoring for a complex system that involves multiple, interacting components, a blend of SAS and non-SAS technologies, and batch processes scheduled to run at night with no one in attendance.

The Alert System, as it is called here, makes use of automated, inline tests to confirm that data steps and procedures are performing as intended and producing expected results. This style of testing is similar in spirit to automated test frameworks such as jUnit, which have become increasingly popular with object-oriented developers. The Alert System was inspired, in part, by jUnit and similar tools.

The paper is divided into three sections. The first section provides an overview of what the Alert System does and how it is used. The second section addresses the question of when it makes sense to build and deploy a system like this. The third section describes the system architecture and provides code samples illustrating techniques used to implement the system.

OVERVIEW OF THE ALERT SYSTEM

The Alert System has the following components:

- An **initialization** routine that establishes the infrastructure for capturing and storing information about any problems that may occur.
- An **error-trapping** routine that captures information about problems detected by the SAS system.
- A set of **assertion** routines. These routines allow customized assertions to be made about the results expected from data steps and procedures. They capture information about the assertions and their outcomes.
- A **cleanup** routine that produces a report using the information captured by the Alert System, and then tears down the infrastructure used to capture and store this information.

One of the goals for this project was to develop techniques that would make it easy for programmers to insert assertions and alerts into both new and existing programs. We approached this by implementing the Alert System as a set of SAS macros. To avoid name collisions with other macros employed in our programs, the names of Alert System macros all begin with the prefix "as." To use the system with a new SAS program, you insert macro calls in a few strategic places in the code.

Consider the sample program outlined below:

```
data surveyResponses;

proc sort data=surveyResponses;

proc transpose data=surveyResponses
   out=surveyTall;

ods output ObliqueRotFactPat=pattern;
proc factor data=surveyTall;

proc print data=pattern;
```

This program creates a dataset of survey responses, sorts and transposes them, performs a factor analysis, and prints the pattern obtained from the analysis. The details of the program are unimportant (and many have been omitted in the outline) but like all Base SAS programs, it consists of a series of data steps and procedure calls.

We will use this sample program to illustrate the steps needed to add alerts to the code using our Alert System macros. First, we bracket the existing code by adding a call to the initialization routine at the top and a call to the cleanup routine at the bottom. We then insert a call to the error trapping routine after each data step or procedure. The result looks like this (added lines are in bold):

```
%asInitialize;

data surveyResponses;
%asCatchErr(create surveyResponses);

proc sort data=surveyResponses;
%asCatchErr(sort);

proc transpose data=surveyResponses out=surveyTall;
%asCatchErr(transpose);

ods output ObliqueRotFactPat=pattern;
proc factor data=surveyTall;
%asCatchErr(proc factor);

proc print data=pattern;
%asCatchErr(print);

%asCleanup;
```

If we run the sample program now, the cleanup routine will generate a report that shows how many data steps and procedures resulted in problems detected by the Base SAS interpreter. Note the character strings passed into the asCatchErr macro. These are arbitrary descriptions included in the report to help identify where problems have occurred.

At this point we could use the summary report as an alternative to scanning the log for error and warning messages. This would be useful, but perhaps not useful enough to justify the extra coding effort. The real value of the Alert System becomes apparent when customized assertions are added to the code. This is done using assertion routines. Currently there are ten assertion routines in our Alert System:

- **asAssertExists** is used to confirm the existence of a dataset.

- **asAssertEquals** is used to confirm that two numbers are equal.

- **asAssertTrue** is used to confirm that a condition is true.

- **asAssertSameStructure** is used to confirm that two datasets have the same fields and field properties.
- **asAssertDatasetsMatch** is used to confirm that two datasets are identical, i.e., they have the same structure and contain the same data.
- **asAssertEqualNs** is used to confirm that two datasets have the same number of observations.
- **asAssertContains** is used to confirm that specific fields are present in a dataset.
- **asAssertContainsOnly** is used to confirm that specific fields are present in a dataset and that no other fields are present.
- **asAssertNotEmpty** is used to confirm that a dataset contains observations.
- **asAssertCleanLog** is used to confirm that a log file contains no error, warning, or fatal messages.

We will now complete the sample program by adding assertions after the initial data step and after the calls to Proc Transpose and Proc Factor. We will use three of the ten assertion routines described above.

```
%asInitialize;

data surveyResponses;
%asCatchErr(create surveyResponses);
%asAssertExists(target=surveyResponses);
%asAssertContains(target=surveyResponses,fields=target_id answer);
%asAssertNotEmpty(target=surveyResponses);

proc sort data=surveyResponses;
%asCatchErr(sort);

proc transpose data=surveyResponses out=surveyTall;
%asCatchErr(transpose);
%asAssertExists(target=surveyWide);
%asAssertContains(target=surveyWide,fields=target_id q1 q126);

ods output ObliqueRotFactPat=pattern;
proc factor data=surveyTall;
%asCatchErr(proc factor);
%asAssertExists(target=pattern);

proc print data=pattern;
%asCatchErr(print);

%asCleanup;
```

Assertions are valuable for capturing problems that might not cause an error message in the log. For example, the first step in this program creates a dataset called surveyResponses. In order for the remainder of the program to work correctly, the surveyResponses dataset must exist, it must contain two specific fields, and it must contain observations. Many problems that would cause these assumptions to be violated would be detected by the SAS interpreter, and thus would be captured by asCatchErr. But there are just as many programming flaws one could imagine that would produce a data step that executes successfully but fails to create the required output.

The assertions are there to guard against this latter possibility. In our sample program we have placed assertions after the data step to confirm that surveyResponses exists, that it contains the required fields, and that it contains observations. These assertions provide an extra layer of assurance that everything is functioning as intended.

When the completed version of the sample program is executed, the cleanup routine produces a report that has a summary section and a details section. The summary section reports how many assertions were tested, how many of the assertions failed, and how many interpreter errors were captured by the error trapping routine. In the details section every assertion is listed and identified as a “Success” or “Failure” depending on the outcome. The expected and actual results of each assertion are also listed. The details section also includes descriptive information for each error detected by the error-trapping routine.

Shown below is an example of a report that might have been generated by the sample program. Note that the sample report lists seven assertions although only six were added to the sample code. That is because the final assertion, “Is log clean?” is automatically generated by the cleanup routine. This assertion scans the entire program log looking for error, warning, and fatal messages. The log is considered clean if no such messages are found.

By default, the cleanup procedure writes the report to the output file. However, the macro also includes an option to send the report as an email message to a designated recipient, with the program log attached as a file.

```
Program executed on 05AUG2004 at 15:34

Summary of inline test results:

7 assertions, 3 failures, 2 errors

Details:

Success : Does surveyResponses exist?
          Expected result: Dataset surveyResponses exists.
          Actual result:   Dataset surveyResponses exists.

Success : Does surveyResponses contain all expected fields?
          Expected result: Dataset contains all expected fields
          Actual result:   Dataset contains all expected fields

Success : Does surveyResponses contain observations?
          Expected result: Dataset contains observations
          Actual result:   Dataset contains observations

Success : Does surveyWide exist?
          Expected result: Dataset surveyWide exists.
          Actual result:   Dataset surveyWide exists.

Failure : Does surveyWide contain all expected fields?
          Expected result: Dataset contains all expected fields
          Actual result:   Dataset does not contain: Q126

Error   : proc factor
          Expected result: Syserr equals 0
          Actual result:   Syserr equals 3000

Failure : Does simple exist?
          Expected result: Dataset simple exists.
          Actual result:   Dataset simple does not exist.

Error   : print
          Expected result: Syserr equals 0
          Actual result:   Syserr equals 3000

Failure : Is log clean?
          Expected result: Log contains no errors or warnings or fatals
          Actual result:   Log contains errors or warnings or fatals
```

WHEN DOES IT MAKE SENSE TO DO THIS?

In reality, the Alert System described here is probably overkill for any program as simple as the one used in the example. It is probably overkill, in fact, for any small to medium-sized program that is created and executed by the same individual. It is just not that hard to review the SAS log and determine if a program worked.

There are circumstances, though, in which an Alert System using inline testing can be invaluable. In general, the more complex a system becomes, the more likely it is that an Alert System will prove useful and cost-effective for monitoring system performance. There are also some specific factors that can make the case for inline testing more compelling. These factors include:

- Multiple components
  When a system becomes so large or complex that it must be divided into multiple, interacting components, changes in one component can have unanticipated effects on other components. Inline testing can help to detect such interaction problems.

- Communication with non-SAS components
  Some systems require SAS programs to pass information to and from other technologies, such as an Oracle database, an Excel spreadsheet, or a Java program. Careful inline testing on the SAS side can help to monitor the results of such communications and detect failures when they occur.

- Multiple developers
  When code development is a shared responsibility, defects may be introduced by programmers working at cross-purposes. Inline testing makes it more likely that when Bob breaks Mary's code, one of them will notice.

- Unattended program execution
  Some programs are designed to be executed on a schedule, automatically, with no one watching. An automated Alert System – especially one that uses email to deliver alerts – is a convenient way of verifying that a scheduled run occurred as planned and produced the expected results (or not).

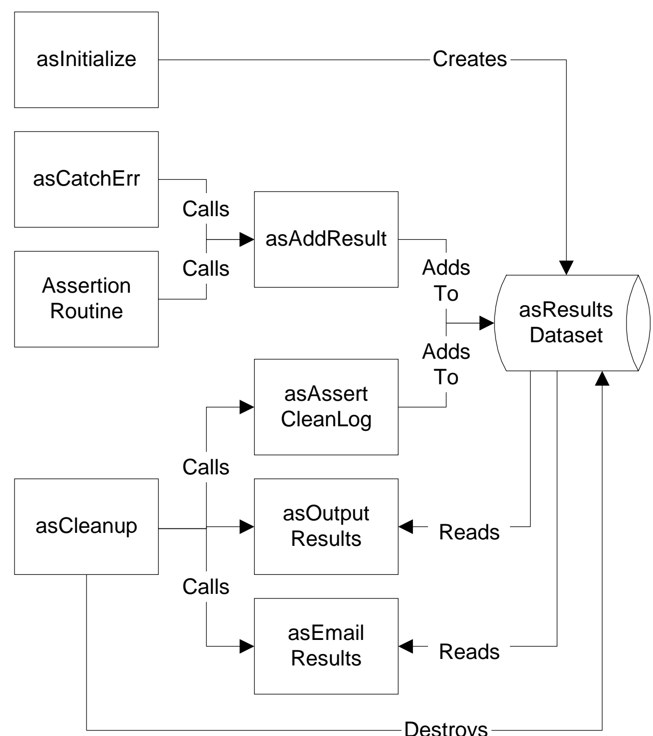
- Routine program execution
  Even when there *is* someone watching, they may not notice that a problem has occurred. This is especially true for programs that are used routinely and have a long history of reliable performance. It is natural to become somewhat lax about reviewing the log for a program that has executed correctly every day for the last six months. An Alert System makes it much more likely that a rarely occurring problem will be noticed in a timely manner.

- Execution by non-programmers
  Many SAS programs are executed by the programmer, but this is not always the case. Non-programmers can do a good job searching for error or warning messages in the log, but they are less likely to notice subtler problems that don't produce such messages. Inline testing with assertions is a great tool for alerting a non-programmer that something is wrong, even though the log is clean.

- Regulatory requirements
  One final value of inline testing is that it can provide explicit documentation that a program has been tested and has executed correctly. In regulated industries, where software development procedures may be subject to external audit, inline testing reports can be a valuable part of a software validation package.

ARCHITECTURE AND CODE SAMPLES

The picture at right show shows the basic architecture of the Alert System. The asInitialize macro creates a dataset called asResults with the fields necessary to store information about assertions and errors. When the asCatchErr macro detects an error, it passes information to the asAddResult macro, which creates a new record in the dataset. The assertion macros perform the code necessary to check their assertions, and then invoke asAddResult to store information about the results. The asCleanup macro does three things: first, it invokes asAssertCleanLog to do a final check for error, warning, and fatal messages in the log; second, it invokes either asOutputResults or asEmailResults to generate a report using the data stored in asResults; and finally, it destroys asResults.



The asInitialize Macro is shown in the first code sample below. Note that in addition to creating asResults, it also redirects the log output to an html file, using the SAS-supplied macro %out2htm. This redirection is useful later for accomplishing two things: searching through the log file for error, warning, and fatal messages, and attaching the log file when emailing the results to a designated recipient.

```
%macro asInitialize;
   * Create the dataset for storing results of inline testing;
   data asResults;
      length type $7 desc $80 expected $50 actual $50;
      if 1=0;
   run;

   * Redirect log output to an html file;
   %out2htm(capture=on,
      window=log,
      runmode=b
      );
%mend;
```

The next code sample shows two macros: asCatchErr and asAddResult. The first one simply checks the automatic macro variable &syserr to determine if the Base SAS interpreter detected any problems during the most recent data step or procedure. If so it passes information about the error to asAddResult, which uses the information to append a new record to asResults.

```
%macro asCatchErr(desc);
   * Get the most recent value of &syserr;
   %let syserrValue=%sysfunc(compress(&syserr));

   * If no description was passed in, create a generic description;
   %if %quote(&desc)= %then %let desc=Any problems during procedure?;

   * If the &syserr value was not 0, add an Error record to the results dataset;
   %if &syserrValue ne 0 %then %do;
      %asAddResult(
         type=Err,
         desc=&desc,
         expected=Syserr equals 0,
         actual=Syserr equals &syserrValue);
   %end;
%mend;

%macro asAddResult(type=,desc=,expected=,actual=);
   * Create a dataset with one record to capture the new result;
   data asNewResult;
      length type $7 desc $80 expected $50 actual $50;
      type = "&type";
      desc = "&desc";
      expected = "&expected";
      actual = "&actual";
   run;

   * Append the new result to the results dataset;
   proc append base=asResults
      data=asNewResult;
   run;
%mend;
```

It would require too much space to list all ten assertion macros here, but the asAssertExists macro is shown below to provide an example of how they work. All the assertion macros have a similar structure, which is explained in the comments embedded in this code sample.

```
%macro asAssertExists(desc=,target=);
   * Create a global variable.  This variable will be used to return the outcome of the
   * assertion (true or false) to the calling program;
   * Set the return value to false;
   %global asExists;
   %let asExists=0;
```

```
   * If no description was passed in, construct a generic description;
   %if %quote(&desc)= %then %let desc=Does &target exist?;

   * Check to see if the assertion is true or false.  Each assertion macro has different
   * code at this point, because each routine tests a different condition.  In this case,
   * the macro checks to see if the dataset identified by &target exists;
   %if %sysfunc(exist(&target)) %then %do;

      * The assertion is true, so set the return variable to true and add a Success record
      * to the results dataset;
      %let asExists=1;
      %asAddResult(
         type=Success,
         desc=&desc,
         expected=Dataset &target exists.,
         actual=Dataset &target exists.);
   %end;
   %else %do;

      * The assertion is false, so add a Failure record to the results dataset;
      %asAddResult(
         type=Failure,
         desc=&desc,
         expected=Dataset &target exists.,
         actual=
            Dataset &target does not exist.);
   %end;
%mend asAssertExists;
```

The asCleanup routine is shown in the next code sample. It closes the html file to which log output was redirected, and then scans that file for error, warning, or fatal messages. Finally, it creates a report summarizing the results of inline testing. The report can be sent to the output file, to an email recipient, or both, depending on what the calling program has requested. When the report is sent via email, the html log file is sent as an attachment with the email message.

```
%macro asCleanup(logfile=,outputResults=,emailResults=,recipient=,subject=);

   * Turn off the redirection of log output to the html file;
   %out2htm(htmlfile=&logfile,
            capture=off,
            window=log,
            openmode=replace,
            runmode=b,
            ecolor=red,
            wcolor=green,
            esize= 3,
            wsize= 3);

   * Generate an assertion to check that there are no error, warning or
   * fatal messages in the log;
   %asAssertCleanLog(logfile=&logfile);

   * If requested by the calling program, send a report of the results
   * to the output file;
   %if &outputResults=yes %then %do;
      %asOutputResults(detail=);
   %end;

   * If requested by the calling program, send a report of the results
   * via email to a designated recipient, with the log file attached;
   %if &emailResults=yes %then %do;
      %asEmailResults(recipient=&recipient,subject=&subject,
         attach=&logfile,detail=);
   %end;
%mend;
```

I have not listed code samples for asOutputResults and asEmailResults because the techniques used there are straightforward. Both routines use a data _null_ step to create a report based on the contents of the asResults dataset. The only difference between the two routines is in the file statement used to indicate where output lines should be written. The data step in asOutputResults has a “file print;” statement which sends output lines to the output file. The data step in asEmailResults has a “file asEmail;” statement, which directs output lines to a file reference previously defined as follows:

```
filename asEmail email "&recipient"
   subject="&subject"
   %if &attach ne %then attach=&attach;
   ;
```

CONCLUSION

In the introduction I mentioned that the Alert System was inspired, in part, by jUnit and other unit testing frameworks. Like these frameworks, the Alert System makes it easy to create automated tests for verifying that a program works. There is an important difference, however. jUnit and its derivatives are used to create automated unit tests that are run during development of a system. The tests are not embedded in the production code and are not executed during a production run. The Alert System, on the other hand, is specifically intended for performance monitoring during production use, so the automated tests are embedded in the production code and are executed as part of the production run.

The Alert System is not foolproof, of course. It is always possible that problems will occur that are not detected by assertions embedded in the code. It is a good practice, whenever that happens, to analyze the failure, define a new assertion that *would* have captured the problem, and modify the code to embed the new assertion. If this practice is followed conscientiously, the Alert System becomes more and more comprehensive over time, evolving into a tight safety net that allows fewer and fewer problems to slip through unnoticed.

CONTACT INFORMATION

Don Hopkins
Ursa Logic Corporation
2625 McDowell Road
Durham, NC 27705

Phone: 919.490.9025
Fax: 919.490.9088

Email: hopkins@ursalogic.com
Web: www.ursalogic.com

TRADEMARK NOTICE

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

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