PharmaSUG 2013 - Paper AD09

A Simple Approach to the Automated Unit Testing of Clinical SAS® Macros

Matthew Nizol, United BioSource Corporation, Ann Arbor, MI

ABSTRACT

Federal regulations require software used in the analysis of clinical trials to be validated. In the pharmaceutical industry, SAS programs used to generate data sets, tables, listings, and figures are often validated via double programming. However, unit testing is often a more appropriate solution for validating shared macros which are used across multiple studies. Unit testing is the process of executing the smallest component of a software system on a known set of inputs and comparing the resulting output to a predefined set of expected results. Unit tests provide confidence that code is implemented correctly; they are also a safety net that protects against unintentional changes to software. But, unit tests can be challenging to write and tedious to run. A unit test framework makes the tester's life easier: it provides a library of macros which both standardize and simplify the writing of tests; it provides a means to run multiple tests at once to make regression testing easier after software is changed; and it automatically reports the results of all test runs so that feedback is immediate. This paper will discuss how to write, in as little code as possible, a simple, maintainable, and robust unit test framework for the testing of clinical SAS macros. This barebones test framework will be useful in its own right, easily validated due to its simplicity, and may also serve as a starting point for an organization to develop a more complex testing solution to meet their own unique needs.

INTRODUCTION

Validation provides confidence that software has been implemented correctly. For clinical programmers, validation is not an optional activity: U.S. Federal Regulations mandate it [4]. Most pharmaceutical companies and clinical research organizations fulfill their validation requirements through double programming. Double programming typically involves two programmers independently developing two programs following the same specification. The programmers execute both programs using the clinical database as input. If the results match, the output is declared validated. This approach only validates the output with respect to the current state of the clinical database. As a result, when the clinical database is updated with new data, previously validated programs may fail on unexpected data points. A shared macro which is to be used across many studies, however, must work properly and robustly on data which the original programmer has never seen. To validate a shared macro, a programmer needs two things:

1. A set of input data. Designing good test data is a large topic beyond the scope of this paper: a good book on software testing can provide guidance.
2. For each set of input data, a corresponding set of expected results based on the macro's specification.

The programmer then executes the macro using the test input data and compares the actual results to the expected results. This process is called unit testing. This brief description of unit testing and double programming has presented something of a false dichotomy between the two. In reality, there is significant overlap between the concepts. If a SAS program is a unit, then double programming can be thought of as a form of unit testing: the clinical database serves as the input data, and the second (validation) program dynamically generates the expected results. The problem with using traditional double programming to validate a shared macro is that a given study's clinical database may not fully test the macro. Double programming a shared macro using synthetic input data designed to fully exercise the macro, however, could be a powerful means to unit test particularly complex macros.

Now imagine the following scenario: a programmer spends 4 hours testing a macro, when a test uncovers a bug. Famously, Glen Myers defined testing as "the process of executing a program with the intent of finding errors." Therefore, the test was successful. But the programmer faces a dilemma: how do they know that the changes they make to the program to fix the error do not introduce new bugs? The solution is regression testing: re-testing portions of software to test whether the program has deteriorated. But if re-testing requires tedious, time-consuming manual review of the results, programmers may be discouraged from writing good tests; that is, tests which are likely to find errors. Moreover, if regression testing is difficult, programmers may be afraid to enhance existing macros. The solution is to automate unit testing through the use of a unit testing framework. In the context of this paper, a unit testing framework consists of two components:

1. A library of macros which standardize test programs, automatically compare expected results to actual results, and provide various useful utilities to make common testing tasks easier

2. A driver program which runs all test programs, automatically detects the results of each test case, and reports the results of all tests to a formatted report

Standardized test programs that use a unit test framework take less effort to write once a programmer learns the standard pattern. Standardizing also makes the results available to a driver program. The driver program, in turn, makes regression testing much easier by running multiple tests and reporting the results without manual intervention. The automatically generated test report can serve as documentation of validation activities to meet regulatory requirements.

The concept of a unit testing framework is far from novel. Open source frameworks are available for the unit testing of programs written in languages such as Java and C#. Even within the SAS community, frameworks such as FUTS [8] and SASUnit [3] are freely available. For many reasons, however, a company might be reluctant to adopt a third party unit testing framework. Company SOPs may require the validation of all installed third party software. Some frameworks [8] require software in addition to SAS to function, which may be undesirable for some organizations. A custom unit testing framework can be written entirely in SAS with ODS output, leveraging the development know-how and ODS templates already present in a clinical programming environment. The framework can be kept very simple, thereby making it simple to learn, use, and validate. This paper will illustrate the simplicity of the basic concepts behind unit testing frameworks and discuss how a custom testing framework that is robust yet simple can be written entirely in SAS in very few lines of code.

DESIGN OF A SIMPLE UNIT TEST FRAMEWORK

As described earlier, a simple unit test framework consists of two essential components: a library of macros and a driver program. The library of macros can be organized into three broad categories:

- Test description macros write messages to the log which identify the beginning and end of a test, describe the test's purpose, and describe the expected results.
- Utility macros carry out common testing tasks to make the test writer's life easier; these are strictly optional additions to the framework.
- Assertion macros compare the test's expected results to actual results and write a message to the log indicating whether the test passed or failed. The framework must contain a variety of assertion macros to compare different types of results (data sets, macro variables, external files, etc.).

New assertion macros and utility macros can be added to the framework as the need arises. Tying the framework together is the driver program, which executes all unit test programs sequentially in batch mode, detects the results of all tests contained within the called programs, summarizes the test results, and writes a report of the test results. Figure 1 below provides a high-level graphical view of this simple design.

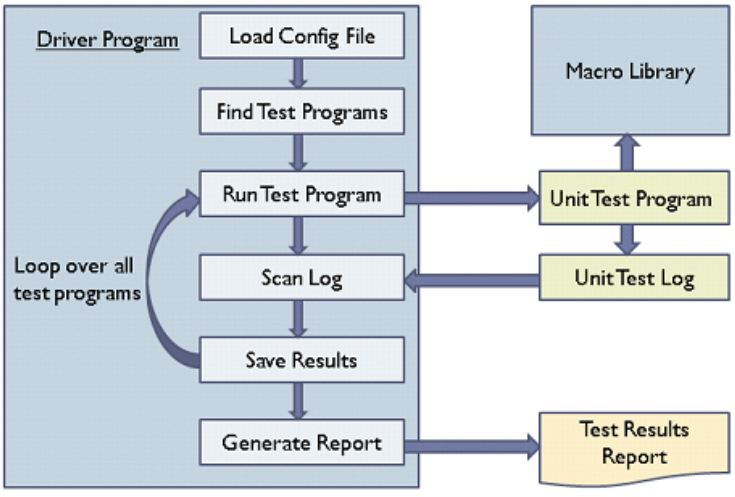

Figure 1: Schematic of simple automated test framework. Items shaded in blue are the framework proper.

ANATOMY OF A TEST PROGRAM

With the above schematic in mind, this section will begin describing the components of the framework by examining a unit test program for a simple macro named %target_day(). The %target_day() macro computes the target visit day within a chemotherapy cycle based on the cycle week. The target day for week 1 is day 1, the target day for week 2 is day 8, the target day for week 3 is day 15, and so on.

```
%macro target_day(week=, outvar=);
   if &week >= 1 then &outvar = (int(&week) * 7) - 6;
   else &outvar = .;
%mend target_day;
```

To be executed properly by the framework's driver program, the test program must call framework macros and be written according to a framework-specific pattern. The test program must begin by pointing to a configuration file which identifies the location of the macro library. A specific test case begins with a call to the %test_begin() macro. This macro writes a line to the log which indicates to the driver program that a new test is beginning along with the name of the macro under test and a unique identification number for the test.

```
%include "..\config.sas";
%test_begin(unit=target_day, id=1);
```

This section of code writes the following to the SAS log:

```
TEST: target_day [1]
```

Next, the test program calls a couple macros to describe the purpose of the test and the expected results. The descriptions will be written to the log and read by the driver program. The driver program will store the descriptions along with the test results so that they may be displayed in the test report. If the description of a test or its expected result is long, the macros may be called multiple times. All descriptions printed to the log for a given test case will be concatenated. Long descriptions within a single macro call will be broken into lines no longer than 200 characters each and then later reassembled by the driver program.

```
%test_describe(desc=Test the target_day macro for several values);
%test_expected(desc=OUTVAR equals (WEEK*7)-6 when WEEK is >= 1. );
%test_expected(desc=OUTVAR is missing otherwise);
```

The above lines write the following to the SAS log:

```
DESC: Test the target_day macro for several values
EXPECT: OUTVAR equals (WEEK*7)-6 when WEEK is >= 1.
EXPECT: OUTVAR is missing otherwise
```

Essentially, test description macros such as %test_begin() and %test_describe() are just wrappers around %PUT statements. The next step is to set-up the test's input data and expected results. Input data and expected results may take the form of data sets, macro variables, external files, and so on. Data sets may be stored externally or they may be defined through a DATA step within the test. Note, multiple tests could share the same input data and so this section could occur at the start of a test program containing multiple related tests. For the %target_day() test program, the input data and expected results take the form of data sets. The %target_day() macro is supposed to return ((WEEK * 7) – 6) if WEEK is greater than or equal to 1 and should return a missing value otherwise. Hence, the below input data will exercise the macro to ensure that it correctly handles missing, negative, and positive values.

```
data input (keep=week)
     expected;
   infile cards;
   input week tgtday;
   datalines;
. .
-1 .
0 .
1 1
2 8
3 15
3.6 15
run;
```

The unit test program must now call %target_day() using the input data and save the actual results:

```
data actual;
   set input;
   %target_day(week=week, outvar=tgtday)
run;
```

Data sets of expected and actual results now exist. The next step is to compare the actual data set to the expected data set and to report the results of the comparison to the log. The comparison is done through an assertion macro. An assertion macro is a macro which asserts that a given condition is true. If the condition is in fact true, the assertion macro prints a success message to the log. If the condition is false, the assertion macro prints a failure message to the log. The types of conditions to be tested determine the assertion macros which must be written. However, the vast majority of tests for SAS macros can be written using two types of assertions:

- Asserting that two data sets are identical
- Asserting that two macro variables have the same value, or equivalently that two strings of text are identical

If test results can be loaded into a data set or stored in a macro variable, then one of the above assertions are all that is needed for the test. That is not to say that other assertion types are not necessary or useful for some tests. Perusing Wright's paper on the FUTS framework [8] provides a sense of the possibilities. Other possible assertion types include asserting that a data set is empty, that a file does (or does not) exist, that external files match, and so forth. The next section of the paper will discuss the two primary assertion macros in more detail, but the next step of the test program is to call %assert_data_equal() to assert that the actual data set matches the expected data set:

```
%assert_data_equal(ds1=actual, ds2=expected);
```

Since the %target_day() macro produces the expected results, %assert_data_equal() writes the following to the log:

```
RESULT: Pass
```

A failing test would have written "RESULT: Fail" to the log. A test requiring manual review (designated by calling %assert_manual) would have written "RESULT: Manual" to the log. The test concludes with a call to %test_end():

```
%test_end;
```

This macro writes the following line to the log:

```
ENDTEST
```

The ENDTEST line notifies the driver program that the test is complete. The driver program can then determine the overall result for the test case, write the result to an output data set, and clear all variables related to the test case.

IMPLEMENTATION OF ASSERTION MACROS

As discussed in the previous section, assertion macros compare expected results to actual results by asserting that a given condition is true. Assertion macros may be kept extremely simple. In fact, some assertion macros may be so simple that it seems an unnecessary bother to write a macro. However, by creating a family of assertion macros, all with similar interfaces and behavior, the chore of writing test programs is made easier. More importantly, the family of assertion macros can all write the test result in a consistent manner to the log so that the driver program can correctly detect the test result. If the framework is updated such that the test result message needs to change, only the assertion macros and not the test programs themselves will need updates.

One of the most common assertions tests whether two symbols (i.e. two strings of text) are equivalent. Below is an implementation of an assertion macro, named %assert_sym_equal(), to test this condition:

```
%macro assert_sym_equal(sym1=, sym2=);
   %if %superq(sym1) eq %superq(sym2) %then %put RESULT: Pass;
   %else %put RESULT: Fail;
%mend assert_sym_equal;
```

Another extremely common assertion is to test whether two data sets are equivalent. Below is an implementation of a macro named %assert_data_equal() to test this condition. Recall that this macro was used in the example test

program above. The %assert_data_equal() macro is little more than a wrapper around PROC COMPARE. In addition to reporting the results of a comparison to the listing, PROC COMPARE sets a global macro variable named SYSINFO. A value of 0 for &SYSINFO indicates that two data sets are exactly equivalent down to the data set labels. The %assert_data_equal() macro as implemented below requires that level of equivalence to report a passing test. The assertion macro provides a parameter, FUZZ, to allow for small differences due to machine precision or rounding. PROC COMPARE prints a warning to the log if either data set has zero observations. To avoid this warning, the macro requires that both data sets exist and have observations. This is accomplished via the %any_obs() utility macro. A separate assertion macro could be used to assert that a data set is empty.

```
%macro assert_data_equal(
   ds1=,    /* Data set one */
   ds2=,    /* Data set two */
   id=,     /* Optional: ID variables to align data sets */
   fuzz=0   /* Vars x and y are reported as identical if abs(x - y) <= &fuzz */
);
   %if %any_obs(&ds1) and %any_obs(&ds2) %then %do;
      proc compare base=&ds1 comp=&ds2 listall method=absolute criterion=&fuzz;
         %if &id ne %str() %then id &id;;
      run;

      %if &sysinfo = 0 %then %put RESULT: Pass;
      %else %put RESULT: Fail;
   %end;
   %else %put RESULT: Fail;  %** Test fails for empty or missing data sets;
%mend assert_data_equal;
```

TESTING FOR ERROR CONDITIONS

In addition to testing the behavior of a macro under expected conditions, it is advisable to exercise the code with invalid parameters, out of range values, etc. in order to test how the macro handles errors. In some cases, the actual objective of the macro is to generate an error. For example, one might write a macro to abort the SAS session if a certain condition is met. SAS provides a number of mechanisms, in particular a number of macro variables, to automatically detect errors. However, knowing which error macro variable to check within a given context can be confusing. It is much easier to inspect the log to determine if any errors or warnings are present. This too can be automated: SAS provides PROC PRINTTO to redirect the log to an external file. The following procedure tests the log messages produced by a section of SAS code:

1. Turn off page numbers and dates in the log (options NODATE NONUMBER). Optionally set NONOTES, NOSOURCE, NOSOURCE2, NOMPRINT, NOMLOGIC, and NOSYMBOLGEN.
2. Redirect the log to an external file via PROC PRINTTO (ideally write the log to the WORK directory)
3. Execute the macro to be tested
4. Redirect the log back to the default location. Restore options turned off in step 1.
5. Read the external file created in step 2 into a SAS data set
6. Compare the data set produced in step 5 to an expected data set using %assert_data_equal()

Notice that testing for log anomalies has been reduced to a comparison of data sets. There are some scenarios in which the above procedure will not work. For example, if a session is aborted (e.g. by a macro designed for that purpose), portions of the log will be inaccessible to the aborted session. Similarly, testing for certain fatal error conditions can put the SAS session into a state that is difficult or impossible to recover from. For these more extreme testing scenarios, the test program can dynamically write the section of code to be tested to an external file, execute that code via a child SAS session using the SYSTEM() function, and then read in the child SAS session's log as in step 5 above. This approach completely insulates the main test program from the fatal error condition that it is trying to detect. Both approaches can be implemented in a single utility macro which takes as parameters the code to be executed (stored in a macro variable) and a flag to indicate whether the code should be executed in a child session or as part of the current session (the latter approach will capture the log using PROC PRINTTO).

GLUING THE FRAMEWORK TOGETHER: THE DRIVER PROGRAM

To reap the benefits of a unit testing framework, there must be a means to run multiple tests at once, detect the results of those tests, and report the results in an automated manner. This is achieved through a driver program. One implementation of a driver program is shown below.

```
%include config;
%run_all_tests(dir=&test_dir, results=results, logscan=logscan);
%summarize_results(data=results, out=sumry);
%write_reports(summary=sumry, detail=results, logscan=logscan);
```

This implementation consists of four parts:

1. A configuration file, which is brought into the program via an %INCLUDE statement. The configuration file identifies the location of directories, sets up autocall libraries, and sets global options and macro variables.
2. A call to a macro (%run_all_tests) which identifies the test programs to run, runs them, parses the log to identify test results and log issues, and saves the test results to a data set.
3. A call to a macro (%summarize_results) which summarizes the test results.
4. A call to a macro (%write_reports) which writes the test summary, detailed test results, and log scan results via three PROC REPORT steps to an RTF file.

The %run_all_tests() macro is the most complex part of the entire unit test framework. A simple yet robust implementation in less than 60 (non-whitespace, non-comment) lines of code is provided in the Appendix. The following pseudo code illustrates the %run_all_tests() algorithm:

```
00 BEGIN DATA STEP
01    INITIALIZE VARIABLE LENGTHS
02    CREATE A FILEREF AND OPEN THE DIRECTORY
03
04    LOOP OVER ALL FILES IN THE DIRECTORY
05       READ FILE NAME (SKIPPING NON-TEST FILES)
06       EXECUTE TEST PROGRAM (WRITE LOG TO THE WORK DIRECTORY)
07       INITIALIZE LOG FAILURE FLAG TO 0
08
09       LOOP OVER ALL LINES IN THE LOG
10          READ LOG LINE
11          PARSE LOG LINE INTO PART1 AND PART2 DELIMITED BY A COLON
12
13          IF PART1 EQUALS
14             'TEST'    THEN READ UNIT NAME AND TEST ID
15             'DESC'    THEN READ TEST DESCRIPTION
16             'EXPECT'  THEN READ EXPECTED RESULTS DESCRIPTION
17             'RESULT'  THEN COMPUTE OVERALL RESULT
18             'ENDTEST' THEN OUTPUT TEST RESULT RECORD
19          ELSE IF LINE MATCHES REGULAR EXPRESSION THEN
20             INCREMENT LOG FAILURE FLAG
21             OUTPUT DIRTY LOG SCAN RECORD
22          END IF
23
24          IF END OF FILE AND NO LOG ISSUES THEN OUTPUT CLEAN LOG SCAN RECORD
25          IF END OF FILE OR END OF TEST THEN RESET TEST CASE VARIABLES
26       END LOOP OVER LINES IN THE LOG
27    END LOOP OVER ALL FILES IN DIRECTORY
28
29    CLOSE DIRECTORY
30 END DATA STEP
```

Though the full implementation is provided in the Appendix, it may be helpful to discuss the innermost loop in more detail. The following three lines correspond to lines 10 and 11 of the pseudo code. The INFILE statement uses the FILEVAR option to identify the log file to open. In this manner, once the inner loop completes and a new test

program is run by the outer loop, the INFILE statement will close the previous log file and open the new log file. Notice also that the INFILE statement specifies that a colon will delimit records. The INPUT statement on the next line uses modified list input (via the colon operator between PART1 and $UPCASE200.) to honor the delimiter when reading PART1; but, formatted input is used when reading PART2 and so any colons in the remainder of the log line are ignored. The third line glues PART1 and PART2 back together (removing control characters, if any) so that the log scanning section has a complete line to parse.

```
infile logref end=eof filevar=log truncover dlm=':';
input part1 : $upcase200. part2 $200.; ** READ LINE IN TWO PARTS;
line = compress(catx(': ', part1, part2), , 'c');
```

Having read a line from the log, the next step is to identify messages from the test programs. Recall from the example program to test the %target_day() macro that the test description macros such as %test_begin() and %test_describe() write lines to the log in the form of "<TYPE>: <MESSAGE>". These log lines serve as a means for the test program to communicate test information to the driver program as the driver program parses the test program's log. There are five types of test messages that this implementation of the driver program expects:

- TEST: <unit> [<id>]
- DESC: <description>
- EXPECT: <description>
- RESULTS: <Pass | Fail | Manual>
- ENDTEST

The driver program therefore checks the current line against each of these possibilities. If the message type is "TEST", the driver program reads the unit name and test id:

```
if part1 = 'TEST' then do;
   unit = left(scan(right(part2), 1, '[]'));
   id   = left(scan(right(part2), 2, '[]'));
end;
```

If the message type is "DESC" or "EXPECT", the driver program reads the description and concatenates it to any previous descriptions provided for the same test. In this way, multiple calls to %test_describe() or %test_expected() can be used to provide long descriptions.

```
else if part1 = 'DESC' then desc = catx(' ', desc, part2);
else if part1 = 'EXPECT' then expect = catx(' ', expect, part2);
```

A given test case could require multiple assertions, each writing a separate RESULT: message to the log. Therefore, the driver program needs to determine the overall result for the test case by considering multiple individual results. If at least one of the test's results is "Fail", the entire test case fails. If none of the results are failures, but at least one result is "Manual", the overall test result is "Manual." Finally, if all of the results are "Pass", the test case as a whole passes. The informat used in the below code maps "Fail" to 3, "Manual" to 2, and "Pass" to 1. As a result, computing the maximum of the individual results produces the correct overall result.

```
else if part1 = 'RESULT' then do;
   resultn = max(input(upcase(part2), result.), resultn);
   result = put(resultn, result.);
end;
```

If the message type is "ENDTEST", the driver program writes the test result to an output data set.

```
else if part1 = 'ENDTEST' then output &results;
```

Finally, if the log line does not contain a message from the test program, the driver program compares the line to a Perl regular expression defined in the configuration file in order to identify errors, warnings, and unexpected notes. If the line matches the regular expression, it is written to a separate log scan data set.

```
else if prxmatch(cats("/&search/i"), line) > 0 then do;
   logfail+1;
   output &logscan;
end;
```

A simple setting for the SEARCH macro variable is below. The below setting is far from adequate to identify the variety of log notes that might indicate a problem. A more complete implementation would include more than 20 common log strings. However, the below example provides the basic form that this variable must take.

```
%let SEARCH = (^WARNING:)|(^ERROR:)|(^[^\d].*_ERROR_)|(^NOTE: MERGE);
```

An explanation of Perl regular expressions is beyond the scope of this paper, but the Base SAS documentation discusses the PRX family of functions and provides basic information on Perl regular expressions.

SUMMARIZING AND REPORTING OUTPUT

After running all test programs and scanning each log, the %run_all_tests() macro produces two output data sets:

- A test results data set containing one record per test case
- A log scan data set containing every problematic log line identified by the Perl regular expression discussed above, along with one record for each clean log indicating that the log has no issues.

The %write_reports() macro outputs the test results data set to an RTF file via PROC REPORT (using an ODS template already available for standard TLF reporting), producing output similar to the following:

Test Results by Test Case

Macro: relday
Test File: test_relday.sas

Test ID	Test Description	Expected Result	Actual Result
1	Compute relative day when prior to treatment start	Relative Day equals Analysis Date minus Treatment Start Date	Pass
2	Compute relative day when on or after treatment start	Relative Day equals Analysis Date minus Treatment Start Date plus 1	**Fail**

The second output data set produced by the %run_all_tests() macro contains all problematic log lines identified during the log scanning. If a file's log was clean, the data set contains a single record for the file indicating that the log was clean. The %write_reports() macro outputs the log scan data set via PROC REPORT. An example log scan report is shown below.

Log Scan Results

Test File Name	Line Number	Log Line	Condition Code
test_calcpfs.sas	42	WARNING: Apparent invocation of macro CALCPFS not resolved.	2
	52	ERROR: Errors printed on page 1.ERROR: Errors printed on page 1.ERROR: Errors printed on page 1.	2
test_relday.sas	142	Clean	0
test_target_day.sas	102	Clean	0

Finally, the %summarize_tests() macro, called before %write_reports(), summarizes the test results for each macro and across all macros. This simple summary is achieved via four steps: PROC FREQ, PROC TRANSPOSE, and two DATA steps. Summaries of other information contained in the test results and log scan data sets could easily be added as well. The %write_reports() macro then outputs the summarized results:

Summary of Test Results

Macro under Test	Number of Passing Tests	Number of Failing Tests	Number of Tests Requiring Manual Review	Number of Tests With Missing Results
Overall	2	1	0	1
calcpfs	0	0	0	1
relday	1	1	0	0
target_day	1	0	0	0

The reports that are provided by the driver program provide two key benefits. First, they provide immediate feedback on all test cases. Second, the reports serve as permanent documentation of testing activities in support of SOPs and regulatory requirements.

RELATED WORK

The literature on unit testing in the broader software development community is vast; freely available frameworks are available for the unit testing of programs written in languages such as Java and C#. Within the SAS community, there is no de facto standard unit testing framework. However, there are two open source frameworks tailored to SAS: FUTS and SASUnit. FUTS was developed at Thotwave around 2006 and described in a SUGI paper by Jeff Wright [8]. It is released under the Eclipse Public License and is available at sascommunity.org. FUTS is beautifully simple: it consists only of a library of SAS assertion macros along with a Perl script to serve as a driver program. Perl must be installed to use FUTS. Test results are reported as text to the DOS command window rather than to a formatted report. SASUnit was developed at HMS Analytical Software GmbH around 2008 and is described in multiple PhUSE papers [3]. It is actively supported and in continued development at sourceforge.net. It is released under the GNU General Public License version 2.0. SASUnit produces impressive and professional HTML output; as a result, SASUnit's implementation is more complicated than the implementation of FUTS.

There is a growing body of literature on SAS unit testing, much of it originating in Europe. Di Tommaso provides an excellent discussion of unit testing for SAS programs [1]. His paper describes the use of a testing macro, %PASSFAIL, to standardize the design and reporting of individual test cases. Di Tommaso's paper suggests test suite automation as a future area of work. As discussed in the introduction, most clinical research organizations utilize double programming to validate study-specific data sets, tables, listings, and figures. Farrugia describes a fascinating pilot project at Roche to use unit testing as the primary validation method for a complex analysis data set [2]. He reports that unit testing uncovered bugs in the production program which were missed by double programming, underscoring some of the shortcomings of the traditional double programming paradigm.

CONCLUSION

Automating aspects of unit testing through the use of a unit testing framework provides several advantages. Regression testing is much easier, which could alleviate fears associated with breaking existing macros. This in turn encourages the writing of better tests—tests likely to find errors—as well as the enhancement and refactoring of macros. By standardizing the manner in which programmers write tests, unit test frameworks can make the job of writing new tests simpler. And, by automatically reporting the results of all test runs in a formatted document, unit test frameworks provide immediate feedback on programming changes as well as permanent documentation of validation activities. This paper discussed the basic concepts of a simple unit testing framework and explained how to write a basic framework entirely in SAS. Due to its simplicity, a basic unit testing framework can be easily developed and validated, taught to new staff, and used in the validation of clinical SAS macros. And, by developing a simple framework in-house, an organization can more easily expand upon the framework, adding new features and library macros as the need arises.

REFERENCES

[1] Di Tommaso, Dante. "Unit Testing as a Cornerstone of SAS Application Development." PhUSE 2011, Paper AD04.

[2] Farrugia, Ross. "A Case Study in Using Unit Testing as a Method of Primary Validation." PhUSE 2010, Paper RG01.

[3] Mangold, Andreas and Dr. Patrick Warnat. "Automatic Unit Testing of SAS Programs with SASUnit." PhUSE 2008, Paper RA07.

[4] Matthews, Carol and Brian Shilling. *Validating Clinical Trial Data Reporting with SAS®*. SAS Institute Inc. 2008.

[5] McConnell, Steve. *Code Complete, 2nd Edition.* Microsoft Press, 2004.

[6] U.S. Food and Drug Administration. "Computerized Systems Used in Clinical Trials." FDA Guidance for Industry, April 1999.

[7] U.S. Food and Drug Administration. "General Principles of Software Validation." FDA Final Guidance for Industry and Staff, January 2002.

[8] Wright, Jeff. "Drawkcab Gnimmargorp: Test-Driven Development with FUTS." SUGI 31, Paper 004-31.

ACKNOWLEDGMENTS

I would like to thank the members of the UBC Statistical Programming Department's SAS Macro Team for their encouragement, ideas, and support in the development of our internal unit testing framework. I am very grateful to Carol Matthews and Sundar Srinivasan for their help in the framework's development and validation. And I truly appreciate the help that Ravi Kumar Reddy Konda provided by reviewing a draft of this paper and providing constructive feedback. Finally, I would like to thank my wife, Lauren, for her support and understanding as I worked on this paper.

CONTACT INFORMATION

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

Matthew Nizol
United BioSource Corporation
2200 Commonwealth Blvd, Suite 100
Ann Arbor, MI 48105
Phone: +1 734 994 8940 x1605
Email: matt.nizol@unitedbiosource.com
matthew@nizol.net

APPENDIX: RUN_ALL_TESTS MACRO

```
%macro run_all_tests(
   dir=,      /* Directory containing the tests */
   results=,  /* Output data set for test results */
   logscan=   /* Output data set for log scan */
);
   data &results (keep=unit id file desc expect result)
        &logscan (keep=file linenum line syscc)
      ;
      length file log line $200 dref unit id result $32 desc expect $5000;

      ** CREATE A FILEREF AND OPEN THE DIRECTORY;
      status = filename(dref, "&dir");
      if status=0 then dirid = dopen(dref);

      if dirid <= 0 then do;
         put "ERROR: Directory &dir could not be opened";
      end;
      else do i=1 to dnum(dirid); ** READ ALL FILES IN THE DIRECTORY;
         ** READ FILE NAME (SKIPPING NON-TEST FILES);
         file = dread(dirid, i);
         if prxmatch('/^TEST_.*\.SAS$/i', strip(file)) = 0 then continue;

         ** EXECUTE TEST PROGRAM.  THE LOG IS WRITTEN TO THE WORK DIRECTORY;
         syscc = system("&sasexe -sysin '&dir\" || strip(file) || "'");
         log = cats("&work\", scan(file, 1, '.'), ".log");
         logfail = 0; ** INITIALIZE LOG STATUS;

         if fileexist(log)=0 then do;
            put "ERROR: Log does not exist: " log=;
         end;
         else do linenum=1 by 1 until(eof); ** READ ALL LINES IN THE LOG;
            infile logref end=eof filevar=log truncover dlm=':';
            input part1 : $upcase200. part2 $200.; ** READ LINE IN TWO PARTS;
            line = compress(catx(': ', part1, part2), , 'c');

            ** SEARCH FOR TEST RESULTS AND LOG PROBLEMS;
            if part1 = 'TEST' then do;
               unit = left(scan(right(part2), 1, '[]'));
               id   = left(scan(right(part2), 2, '[]'));
            end;
            else if part1 = 'DESC' then desc = catx(' ', desc, part2);
            else if part1 = 'EXPECT' then expect = catx(' ', expect, part2);
            else if part1 = 'RESULT' then do;
               resultn = max(input(upcase(part2), result.), resultn);
               result = put(resultn, result.);
            end;
            else if part1 = 'ENDTEST' then output &results;
            else if prxmatch(cats("/&search/i"), line) > 0 then do;
               logfail+1;
               output &logscan;
            end;

            ** AT THE END OF THE LOG, INDICATE IF CLEAN;
            if eof and logfail < 1 then do;
               line = "Clean";
               output &logscan;
            end;

            ** RESET VARIABLES;
            if eof or part1='ENDTEST' then do;
               call missing(unit, id, desc, expect, of result:);
```

```
            end;
          end; ** LOOP OVER LINES IN THE LOG;
        end; ** LOOP OVER ALL FILES;

        ** CLOSE DIRECTORY AND STOP THE DATA STEP;
        status = dclose(dirid);
        stop;
    run;

    %** SAVE TEMP LOG/LST;
    %sysexec(move "&work\*.log" "&dir\.");
    %sysexec(move "&work\*.lst" "&dir\.");
%mend run_all_tests;
```