PharmaSUG 2023 - Paper AP-253

Creating The cxtf Native SAS Test Framework

Magnus Mengelbier, Limelogic AB

ABSTRACT

The natural evolution of business processes will eventually gravitate towards standardization and further automation to not only improve upon time to delivery but to further quality and standards. SAS macro libraries play an integral part to drive the process forward. As the use of macros evolve from repetitive processing to utilities and shareable process components, the effort of testing requirements and validation approaches change. Although test frameworks that support SAS exist, they frequently require command line or external access, which is not always possible in advanced solutions or embedded systems. The design, evolution, and learnings from creating a native SAS test framework is discussed through practical examples, including the benefits and drawbacks of testing of SAS code and macros in situ.

INTRODUCTION

The use of SAS macros within programming provides simple means to execute repetitive tasks, support the use of standards or create small useful utilities to simplify programs. SAS environments, and inherently those programs and macros, that are used with GxP processes and workloads or business critical functions may require some level of testing or validation.

Each organization has most likely standards and processes for testing and validation, whether that is based on test programs combined with log scanners, manual log reviews or existing test frameworks like SASUnit and FUTS (if you can find a copy). Either or a combination of those approaches can be fully adequate, but if your environment is a solution with SAS deeply embedded or where command line access is disabled or not permitted, the ability to execute tests and report results from within a single SAS session may be required.

The cxtf test framework is a native SAS utility inspired by the R package *testthat* and lessons from using unit testing approaches for SAS utilities. Set with a simple goal to execute a set of tests and report the result in different formats, we consider how SAS program and macro code influences the design and minimum feature set of the test framework.

A TEST FRAMEWORK

A test framework is simply a tool or utility that allows you to execute different test scenarios and capture and record the results of each in a structured consistent way. The test scenario is a challenge that you would like to perform to demonstrate that a feature or function works and/or behaves as expected.

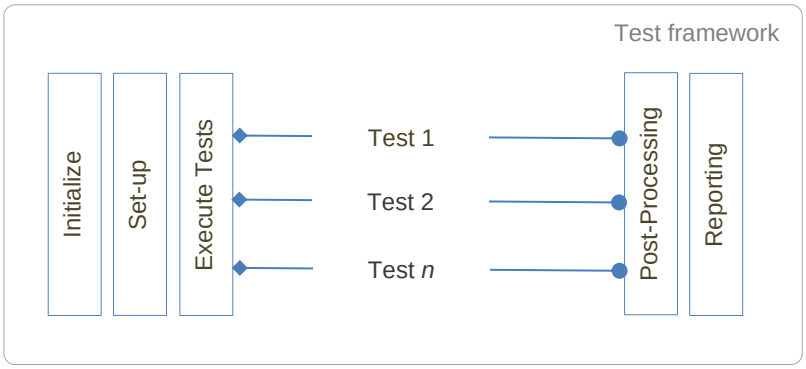

Figure 1 Structure of a test framework

Most often you may need several, and that can mean many, test scenarios (Figure 1) to verify that you have fulfilled a specific requirement as each scenario is designed to challenge a pre-defined or specific part of the requirement. Although, discussing topics like the use of the S.M.A.R.T. criteria and conventions for requirements or tests is beyond the scope of this paper, we should consider that a test framework should accommodate well-defined requirements and their tests that are designed and programmed to be specific, measurable, and testable.

A test scenario, sometimes interchangeably referred to as simply a *test* for brevity, has a few distinct properties. The test is a self-contained set of steps, regardless of if it is entirely stand-alone (Unit Test) or tests an integration with external resources (Integration Test), both types often programmatic in nature. Testing a longer sequence of steps that form part of a business process is commonly referred to as User Acceptance Tests (UAT) and are not the low tests that test frameworks are well suited for, although the framework can be used for specific use cases.

The challenge set forth is thus to create a simple framework that allows the user to write almost any test and provide a convention where the structuring, execution and reporting of results is as simple and flexible as possible.

CXTF DESIGN

The cxtf test framework takes inspiration from test frameworks in other languages, most notably testthat in R:

```
test_that( "Some test I wrote", {

  a <- 1
  b <- 2

  c <- function_under_test( a, b )

  expect_equal( c, 3 )

  rm( a, b, c )

})
```

Even though the above example may seem simple, it fully reflects a valid test to verify that the function function_under_test(…) correctly calculates the value 3 from the two input values. In this case, the rm(…) function is not really necessary, but including it demonstrates the four basic parts of a test; test setup, testing our target function, any assertions that demonstrates the function has performed correctly, and finally the explicit teardown and/or clean-up.

A similar approach for SAS is usually executing each scenario as a standalone program in batch mode and interpreting the log and results, a design that most SAS testing frameworks incorporates, but a valid question is if a similar approach to testthat can be created for use with SAS.

Assuming SAS macros being the closest representation to R functions, the above example from R can be written as:

```
%macro test_some_test_i_wrote();

    %let a = 1;
    %let b = 2;

    %let c = ;
    %macro_under_test( &a, &b, return = c );

    %cxtf_expect_equal( &c, 3 );

%mend;
```

Selecting SAS macros as the convention was primarily driven by simplifying test code development as it is a commonly available skillset for a well-established architecture. At the same time, the choice inherits many positive technical attributes that greatly simplifies the internals of the framework.

It is worth noting that the `return` macro parameter is just one of several different defensive programming approaches that can be used when a macro returns a value and is not specific to the test framework.

A complete discussion of the cxtf framework is beyond the scope of this paper, but we shall highlight some of the key programming techniques used to make the above example work. Interestingly, most techniques are well-known and widely applied to SAS programming tasks, but not obvious in association with validating SAS macros or compiled functions.

DOSUBL TO THE RESCUE

A key consideration in any test framework design is the underlying mechanism, essentially how tests can be executed so that we can capture the result. Each test is in principle a concise independent identifiable block of code that represent a predefined test scenario, i.e. our test requirement or specification.

We can readily assume that a validation effort, of say a SAS macro or compiled function, is not represented by one single test but a collection. The mechanism will therefore need to be able to independently execute each test in such a way that one test does not affect subsequent tests (unless designed to do so) and that each can be treated as a standalone independent unit.

The test block should also be identifiable, as in our test macro example, since we need to be able to identify to what test a particular result is associated with.

Different conventions and approaches can be chosen, with %include or CALL EXECUTE statements within an administrative wrapper being early considerations. Both techniques became unmanageable as any inadvertent or test scenario driven errors and warnings would affect the overall cxtf process, possibly resulting in an unrecoverable error. The result was frequently incomplete and/or unintelligible logs with missing or incomplete records as a result.

The DOSUBL function, called from within a DATA step or the %SYSFUNC() macro function, is however more ideal as it will immediately execute the specified code in parallel through a *side session*, or *SAS executive* as noted in the SAS documentation.

In brief, a side session shares the resources of the parent session, such as WORK library and any SAS library and file references already assigned, but any SAS library and file references created within the side session will be discarded and not available to the calling session.

The macro scope of the parent session, including the scopes associated with nested macros up to and including the global macro variable scope, are copied to the side session.

```
%macro inside_DOSUBL;

  %put Inside DOSUBL before   &=global_var    &=outer_a ;

  %let global_var = 100000;
  %let outer_a = 1000;

  %put Inside DOSUBL after   &=global_var    &=outer_a ;

%mend;


%macro inner_with_DOSUBL;
    %let rc = %sysfunc( DOSUBL( '%inside_DOSUBL;' ) );
%mend;


%macro outer();
```

```
    %global global_var;

    %let global_var = 1;
    %let outer_a = 1;

    %put Before   &=global_var    &=outer_a ;
    %inner_with_DOSUBL;
    %put After    &=global_var    &=outer_a ;

%mend;


%outer();
```

with the SAS log containing:

```
Before   GLOBAL_VAR=1    OUTER_A=1
Inside DOSUBL before   GLOBAL_VAR=1    OUTER_A=1
Inside DOSUBL after   GLOBAL_VAR=100000    OUTER_A=1000
After   GLOBAL_VAR=100000    OUTER_A=1
```

Note that the value for the global macro variable `global_var` that is set within the DOSUBL call is updated and retained, while the updated value to the macro variable `outer_a` in the scope of macro `%outer()` is discarded. This is a feature that the test framework relies heavily on so needs to be monitored if it is due to a SAS design feature for or defect in DOSUBL.

Also, since the cxtf framework is in itself constructed as a collection of nested macros, this would imply that any communication from within DOSUBL with the test framework needs to be performed through data sets and in such a way that the test framework data sets do not impact tests using the WORK or any other user defined libraries. We avoid this conflict by creating a work library `_CXTFWRK` that simply points to a sub-directory of the WORK library.

In essence, and from the perspective of the test framework, the DOSUBL function with its characteristic attributes and behavior is a simple means to execute tests as independent and standalone code blocks that fit well with the conventions of a test framework.

SAS ERRORS AND WARNINGS

One other critical aspect that the test framework mechanism needs to perform well is capturing and identifying any errors or warnings, whether they are inadvertent failures or part of test scenarios specifically designed to ensure that said SAS macros and compiled functions return an error or warning when they should, i.e. negative testing.

Error management in SAS, or rather the lack of exception handling similar to which we can be find in other languages like R and Python, is really a tribute to how we use SAS within our organizations. A simplistic and generalized perspective is that SAS by design is a scripting language for analyzing data with error management designed to best suit that application.

Most SAS users simply rely on a log statement starting with ERROR: or WARNING: to recognize an error or warning, respectively. SAS does however also include status codes, one such being the SAS System Condition Code, the automatic global macro variable SYSCC. There exist other automatic macro variables such as SYSRC, SQLRC, etc., that are specific to certain SAS functions or procedures, but it is SYSCC that is consistent for all applications.

The condition code can take any positive value equal to or greater than 0, where 0 represents no errors or warnings. From the perspective of the test framework, we do not have to consider the individual values, just simply if the condition code is equal to 0 (no errors, warnings, etc.) or not, i.e. a test failed.

A key aspect of SYSCC is that it is write-enabled, meaning you can reset the SAS session condition to a state with no errors. Keep in mind that other variables, such as SYSMSG, work in conjunction with SYSCC.

One interesting aspect of DOSUBL is that the value of SYSCC is not passed from the side session to the calling parent session as we noted previously for global macro variables. SYSCC represents the condition code of the *current* session and not all SAS sessions, so SYSCC defined in the side session is only applicable that specific side session.

To demonstrate this, consider the following:

```
%macro inside();

   %put Inside before &=syscc;

    data _null_ ;
      set work.i_do_not_exist ;
    run;

   %put Inside after &=syscc;

%mend;


%macro outer();

   %put Outside before &=syscc;
   %let rc = %sysfunc( DOSUBL( '%inside()' ) );
   %put Outside after &=syscc;

%mend;

%outer();
```

would result in the log:

```
Outside before SYSCC=0
Inside before SYSCC=0
ERROR: File WORK.I_DO_NOT_EXIST.DATA does not exist.
NOTE: The SAS System stopped processing this step because of errors.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


Inside after SYSCC=1012
Outside after SYSCC=0
```

This is an important notion as we can use DOSUBL as a simple form of exception handling, much like try-catch or try-except blocks in other languages. At the same time, it also highlights that an alternative method than SYSCC is needed to detect the occurrence of any errors or warnings as a non-zero value of SYSCC may be masked.

Considering the previous notion to simply look for lines starting with ERROR: and WARNING: to detect SAS execution errors and warnings, respectively, the framework only needs to ensure that detection is only performed in association with the current test. A simple approach is to use the PRINTTO procedure to temporarily redirect the entire log to a temporary file, call it the test log, for the duration of a single test and perform a read-back using a DATA step and PUT statements to ensure the captured log entries are available in the main program log as a matter of record. The test log files, one for each test scenario, can then be inspected for execution errors and warnings associated with that specific test as noted above.

The test framework assumes that any error or warning in the test log is indicative that the test code unexpectedly failed, which would fail the test. The framework was extended to support annotations that would note that an error or warning is expected to occur:

```
%*@test expecterr ;
```

```
%*@test expectwarn ;
```

In the above case, if both annotations are defined for a test, both an unspecified error and unspecified warning would be expected. If either or both were not detected, the test would fail as the test code did not result in the expected error and warning.

The annotations can include a message part, following the example at the top of this section:

```
%*@test expecterr File WORK.I_DO_NOT_EXIST.DATA does not exist;
```

Note that the *ERROR:* string is not required as it is assumed for the `expecterr` annotation.

The use of DOSUBL provides the framework with the necessary controls to execute a given arbitrary test and, at the same time, capture any errors or warnings, whether they are inadvertent failures or part of test scenarios.

IDENTIFYING TESTS

A great deal of flexibility on how test code blocks can be defined and programmed is inherited from DOSUBL as it can essentially execute any SAS code that you would regularly submit through a normal SAS session. This includes the framework convention of test scenario macros. As code does not automatically submit itself, the framework also needs to define a convention for storing the code associated with the test scenarios.

Different approaches to store these macros were considered, from one test macro per file for use with the SAS Autocall facility, one single file with all tests, and finally multiple tests defined in one or more program files. The latter was chosen as it would give developers the most flexibility in organizing both tests and the effort to create them, as well as suit the controls that the framework would need to process them.

The scheme was further simplified across multiple iterations. To start, cxtf provides the capability to execute tests defined through a single file or all test programs in a folder. In the latter case, a test program is identified as a SAS program file name with the prefix `test-*` or `test_*`.

Within each test program, the framework will identify all macros with a name that starts with the prefix `test_*` to represent a test scenario as below:

```
%macro test_mytest( a = 0 );
    ...
%mend;

%test_mytest( a = 1 );
%test_mytest( a = 2 );
```

This permits the test developer to freely create utility macros that are used by, but ignored as, test scenarios. Further enhancements were implemented to permit a test macro to be called multiple times, say for parameter driven permutations, beyond the default call. A default call is automatically calling the macro without specifying any parameter values.

Just finding test code to execute is not sufficient if we cannot identify to which particular test the code is associated with. The test framework uses the test macro name, excluding the prefix `test_`, as the default test scenario identifier. Sometimes it may be advantageous to refer to a test using an alias, which can be defined using the *alias* annotation:

```
%*@test alias myothername ;

%macro test_mytest( a = 0 );
    ...
%mend;
```

The alias annotation is associated with a single unique test, such that the test macro and permutations as above can each be assigned a unique alias.

WRITING TESTS

The test framework assumes that tests are written along a few simple conventions. Each test can have one of four outcomes; Pass, Fail, Skip and Not Done. The outcome is derived from the cumulative assertions, e.g. the checks of actual versus expected that are performed as part of the test.

A test will have an outcome of Pass if, and only if, all assertions are true, so all tests and assertions are written that way. A test that is identified in the inventory of tests to execute, not skipped and has no recorded assertions is recorded as Not Done.

Given our first example:

```
%macro test_some_test_i_wrote();

    %let a = 1;
    %let b = 2;

    %let c = ;
    %macro_under_test( &a, &b, return = c );

    %cxtf_expect_equal( &c, 3 );

    %cxtf_expect_equal( &c, &a, not = TRUE );
    %cxtf_expect_equal( &c, &b, not = TRUE );

%mend;
```

The test contains three assertions that must all result in a true comparison. Note the latter two uses a parameter `not` to indicate that the equality should be negated. This option is a standard convention across all cxtf assertion macros, but we could just as easily have created another assertion macro `%cxtf_expect_notequal()`.

For the above test to pass, all three assertions have to be true and both the test macro and macro(s) under test have to execute without an error or warning, since there are no annotations specifying otherwise.

The example does highlight that the logistics of recording the pass or fail result is entirely performed by the test framework.

The ability to force a test to pass, fail or skip based on test code logic is supported through a standard set of cxtf macros (note the different uses of %return and %goto macro statements):

```
%macro test_a_test();

    %if ( &SYSSCP = WIN ) %then %do;
        %cxtf_skip( Cannot run test on Windows );
        %return;
    %end;

    %if ( %sysfunc(exist( work.myinput )) ) %then %do;
        %cxtf_fail( Input data not available );
        %goto cleanup;
    %end;

    %cleanup:

%mend;
```

We can also utilize annotations to skip a test entirely if some prior conditions have, or have not, been met:

```
%*@test skip not test1 ;

%macro test_mytest( a = 0 );
    ...
%mend;
```

The test from a previous example above will in this case be skipped if the test *test1* does not pass, i.e. the keyword `not`.

The main aim of the test framework is to rely on the conventions and standards for SAS macro development and not impose any specific programming constructs beyond standard macros and naming conventions.

CONCLUSION

The concept of writing tests for a SAS macro or compiled function is not new, but to be able to use SAS macros for writing tests to validate said macros and functions provide a flexible framework. The DOSUBL function, and how it provides an independent isolated SAS session, and other more mainstream programming techniques gives us a set of mechanisms that we can utilize to execute tests and capture results without affecting subsequent tests.

ACKNOWLEDGMENTS

The cxtf test framework is an extension of the great research effort started by Martine Dubourgeat.

CONTACT INFORMATION

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

Magnus Mengelbier
Limelogic AB
papers at limelogic.com
limelogic.com

All examples contained in this paper used R package testthat version 3.0.2 and SAS 9.4 TS Level 1 M6 (9.04.01M6P111518) on Microsoft Windows.