Paper 011-2013

Automated Testing of Your SAS® Code, and Collation of Results (Using Hash Tables)

Andrew Ratcliffe, RTSL.eu, United Kingdom

ABSTRACT

Testing is an undeniably important part of the development process, but its multiple phases and approaches can be under-valued. I describe some of the principles I apply to the testing phases of my projects and then show some useful macros that I have developed to aid the re-use of tests and to collate their results automatically. Tests should be used time and again for regression testing. The collation of the results hinges on the use of hash tables, and the paper gives detail on the coding techniques employed. The small macro suite can be used for testing of SAS® code written in a variety of tools including SAS® Enterprise Guide®, SAS® Data Integration Studio, and the traditional SAS Display Manager Environment.

INTRODUCTION

This paper briefly describes different types of testing, positive and negative testing, the need for a test strategy to decide which types of testing you will use, and how to structure written tests. Tests should be used time and again for regression testing and this paper describes why you should consider your tests and your testing work to be an investment.

The focus of the paper is the steps that I took to build a test framework of SAS macros that make regression testing quick and easy, provide automatic collection and collation of results, and culminate in a red/green indication of 100% success (or fail). The collation of the results hinges on the use of hash tables, and the paper gives detail on the coding techniques employed.

The small macro suite can be used for testing of SAS code written in a variety of tools including SAS Enterprise Guide, SAS Data Integration Studio, and the traditional SAS Display Manager Environment.

WHAT IS TESTING, AND WHY DO IT?

My dad says "if a job's worth doing, it's worth doing well". I say "if a bit of code is worth writing, it's worth testing it properly". Maybe I'm stretching the old saying a little, but the principle remains true.

Software testing is a very large subject area; I'm not going to try to reproduce a text book here. I'm simply going to describe some of the principles I apply to the testing phases of my projects and then show some useful macros that I have developed to aid the re-use of tests. There are many different types of test phase, each with different objectives. Some of these were briefly covered in my "SAS Software Development with the V-Model" paper at SAS Global Forum (SGF) 2011. I'll revisit some of those later in this paper. Right now, I'll offer my main principles of testing:

TESTING PRINCIPLES

- To test something, you need to know what it should do, in all circumstances. This means you need to have established an agreed set of requirements and/or specifications.
- There are a number of reasons why you might need to re-run a test - because the test failed, or for regression testing. For this reason, and for others, automated tests are preferable to manual tests.
- Look upon your tests as an investment. Firstly, finding bugs before go-live is always "a good thing" for a number of reasons. But secondly, tests invariably need to be re-run, so the more effort you put into them the more they'll repay you when you have to re-run them. A library of re-usable tests is an asset.
- Don't just test the "happy path" for your system. Test that the system rejects bad input and handles unexpected situations elegantly. This is called "Negative Testing". In simple terms this might mean testing with values of zero, one, minus one, negative, non-integer, and very large numbers.
- Only rarely can testing offer a guarantee that the product is 100% bug free and works as intended. Most often, testing can only offer a degree of confidence that the product is bug-free and works as intended. The precise degree of confidence that your test process is intended to provide is your decision and will depend on the nature of the project and how much time and money are available overall. This is a strong influence on your test strategy.

- Document your test strategy. This includes stating which testing method & tools will be used for each different type of system element, e.g. data entry screens, report-generation wizards, small files, big files, important reports (to be sent to regulatory authorities, for example), less important reports (for internal information only, for example). Specify how you will compare expected results with actual results when the output is (for example) a large data set or report. Will you do an automated 100% comparison of all cells, or will you do a manual spot check of 1% of the cells?
- Document your test plan and test cases, i.e. the individual steps (and expected results) that the tester should follow. Documenting your test steps means that they can reliably be re-run if the tests have to be done again.
- Capture your input data, or create it programmatically each time you run test. Either way, your tests will be entirely repeatable.
- With regard to documentation, I always preach the "barely adequate" approach, i.e. do what needs to be done ("adequate") but don't go beyond ("barely"). In order to do this, you need to clearly understand the objectives of each document and the intended audience(s). Sometimes you need separate documents; sometimes you can put all of the content into one document.

TEST PHASES & TYPES

There are many different types of test phase, each with different objectives. Some of these were briefly covered in my "SAS Software Development with the V-Model" paper at SAS Global Forum (SGF) 2011. I'll revisit some of those here and add some more phases and types

Unit Test Phase	A unit is an individual unit of code, e.g. a macro, or a DI Studio job. The objective of the unit test phase is to check that each feature in the unit specifications has been implemented successfully in the module. Unit testing is usually white box testing, i.e. it requires knowledge of the code. This should be done by a different programmer but is often done by the coder themselves. If a unit is reliant on one or more other units then the tester may prepare dummy units so that the emphasis is on testing just the one real unit, or it may be pragmatic to skip to integration testing for some units.
Integration Test Phase	Integration testing consists of executing groups of units in order to test their interaction and to see if they behave as predicted by the Design Specification. Units should only be integration tested once they have successfully been unit tested. Integration tests can be planned in a layered fashion so that they test increasingly large groups of units until the full system is constructed and tested. Integration testing is focused upon the communication between the units, not their individual behaviour. We wish to test that the units co-exist and cooperate as planned, under error conditions as well as the "happy path"
System Test Phase	In the system test phase we check the system as a whole; we do not pay attention to the individual parts of the design. This test phase treats the whole system as one big unit. System testing can involve a number of specialist types of test to see if all the functional and non-functional requirements have been met. In addition to functional requirements these may include the following types of testing for the non-functional requirements: • Performance - Are the performance criteria met? For example, batch turnaround time or interactive response times • Volume - Can large volumes of information be handled? For example, do you get out of memory issues? • Stress - Can peak volumes of information be handled? For example, peak hour of the day, or year-end • Documentation - Is the documentation usable for the system? • Robustness - Does the system remain stable under adverse circumstances? • Security – Is each application role limited to appropriate access to each system artifact (in each dev/test/prod environment? For example, user/admin/support roles, business data/logs/code

User Acceptance Test Phase	User acceptance testing (UAT) is for the customer to check that the system does what they need. It may sound like system testing but there's a key difference: Systems testing checks that the system that was specified has been delivered, UAT checks that the system delivers what was requested. As its name implies, UAT is done by the users. The users should check that the system fits into the business processes that they choose to employ, and they should check that the documentation and training is adequate and fit for purpose.
Operational Acceptance Test Phase	When a system has completed its development and is ready for go-live, the users need to confirm that it is fit for their purposes and that they can use it, but the system may also be on the point of being handed-over to an operations team who will check and remediate overnight runs and/or deal with users' questions about the interactive elements of the system. If this is the case, the operations team will need to do their own "UAT" to confirm that they are capable of operating and supporting it. The operations team should check that they know how to start, stop, and re-start some or all of the system, that they understand the schedule and its dependencies, and that the Run Book or Operations Guide contain suitable/sufficient troubleshooting information

Test types:

Peer Review	Most of the preceding tests are dynamic in nature, i.e. they involve the execution of the code. Peer review is a static test. As its name suggests, it is a review of the code and other artefacts by fellow programmers who haven't written the code themselves. The objectives of the review are listed below. It is important to note that criticising your fellow programmers is not one of the objectives, it should be a positive exercise. • Confirm that the code meets documented Coding Guidelines • Confirm that the code appears to meet the Design Specification • Share knowledge of the system with the reviewers • Learn lessons that can be applied in future coding exercises
Mutation Testing	A test of your tests, mutation testing involves making copies of your code, changing each copy in a small way such that it no longer functions correctly, and then running your tests against each copy. If your tests are sufficiently comprehensive then each mutant copy of your code will fail one or more of the tests. As I said earlier, testing is about establishing an appropriate degree of confidence in your system. Mutation testing goes beyond the measure of "appropriate" in many cases
Negative Tests	Negative tests check that the system behaves as designed/desired when unexpected things happen. This can range from a user typing-in an invalid date (is the value accepted, causing subsequent failure, or is a clear message issued?) through to the server failing in the middle of a complex batch suite (can the batch jobs be restarted without a great deal of manual intervention and/or rolling-back of data?). As ever, the "appropriate confidence" measure must be applied; testing every possible negative event is unlikely to be appropriate.

GOOD PRACTICE AND RECOMMENDATIONS

It is very rare to find a test that only gets run once. Even if the code works first time, it may need updating later in its life and you will need to re-run the test to make sure that the existing functionality hasn't been broken by the addition of new functionality. This is known as regression testing. **Creating and documenting tests is a real investment that will pay dividends for sure.**

For each test, you should specify clear steps alongside the expected results from the steps. The steps need to be written clearly and unambiguously, and so do the expected results. If you're expecting to re-run the tests (you are, right?) then you need to **be confident that the very same test will be run each time** and hence the very same results will be produced.

Think about the inputs to the test process. Document the access rights and software tools that your tester will need, e.g. read/write access to /my/team/data, and the use of SAS Enterprise Guide.

Think about the data inputs too. Ideally your tests will include code/steps to create test data and/or you will store your input test data alongside your test documents so that the tester doesn't have to go find some new data themselves – thereby undermining the aim of repeatability.

If we accept the likelihood and value of re-running tests, and we accept that running the very same test every time is a good thing, then it seems a logical conclusion that **automated testing** offers advantages over manual testing. When we talk of automated testing we're referring to the programmatic scripting of tests so that they can be run and re-run with one command. A fully automated test will extract its own test data from the test library (or create the test data programmatically), run the test(s), store the outputs and results, and then remove the test data.

AUTOMATED TESTING

So, having stressed the importance of testing, let me give you some hints on how I keep the test phase efficient and effective on my projects. My number one tip is to automate your tests, and I'll describe a simple macro that you can use to highlight test results in your log. I'll then describe how to enhance the macro and easily add a reporting system to summarise your test results in one place.

Let's assume you have a set of tests scripts, i.e. steps for the tester to take, accompanied by expected results for each step. Let's take a simplified example:

1. Run the customer load. Inspect the SAS log. Expect no error or warning messages.
2. Count the rows in the input customer file (CSV) and the warehouse customer table (SAS data set). Expect the number of rows in each to match. I said it's a simplified example!

We can automate the second test (the first too, but that's for another time). The benefit of automating is that we can re-run it quickly and effectively (after test failure, and for regression testing).

Our automated code might look like this (assuming we've already run code to create whouse.cust from cust.csv):

```
%let infile=/a/b/cust.csv;
%let outdata=whouse.cust;

/* Pipe output from unix command via fileref */
filename in pipe "wc -l &infile";

data test1_result;
  /* Not interested in reading observations, just want nobs */
  set &outdata nobs=outcount;
  /* Get line count from unix command */
  length incount_c $20;
  infile in;
  input incount_c $20;
  incount = input(incount_c,best.);
  /* Compare the two figures */
  if incount eq outcount then
    put 'Test 1 was passed';
  else
    put 'Test 1 was failed: ' incount= outcount=;
  stop;
run;
```

This code will use the unix "wc -l" command to return a line count of the input file via the piped fileref, and the nobs parameter on the SET statement to return a row count of the warehouse table. It then compares the two values and reports the success/failure of the comparison into the log.

WITH BASIC MACROS

You can see that we might do a number of similar comparisons in our whole test suite, so it would be useful to report them to the log in a consistent, easy to find fashion. We can turn the assertion into a macro as follows:

```
%macro assert_condition(left,operator,right,tag=);
  if &left &operator &right then
    put "TESTING: &sysmacroname: TAG=&tag, OUTCOME=PASS";
  else
    put "TESTING: &sysmacroname: TAG=&tag, OUTCOME=FAIL";
%mend assert_condition;
```

I've introduced the tag parameter for use as a unique identifier for each test. We can use the macro within our DATA step as follows:

```
data test1_result;
  /* Not interested in reading observations, just want nobs */
  set &outdata nobs=outcount;
  /* Get line count from unix command */
  length incount_c $20;
  infile in;
  input incount_c $20;
  incount = input(incount_c,best.);
  /* Compare the two figures */
  %assert_comparison(incount,eq,outcount,tag=Test 1);
  stop;
run;
```

And we can create a higher-level testing macro that allows us to compare any raw file with a SAS data set and assert that the row counts will be equal:

```
%macro assert_EqualRowCount(infile=,outdata=,tag=);
  /* Pipe output from unix command via fileref */
  filename in pipe "wc -l &infile";

  data _null_;
    /* Not interested in reading observations, just want nobs */
    set &outdata nobs=outcount;
    /* Get line count from unix command */
    length incount_c $20;
    infile in;
    input incount_c $20;
    incount = input(incount_c,best.);
    /* Compare the two figures */
    %assert_comparison(incount,eq,outcount,tag=&tag);
    stop;
  run;
%mend assert_EqualRowCount;
```

Armed with our %assert_EqualRowCount macro we can easily automate a range of similar tests, and get consistent and easy-to-find results in the log.

However, the log is not the most convenient place to place our results. Putting them into a data set would be far better - we could have just one row per test result (no need to scan for messages in the log), and we could easily print it out (with traffic lighting for pass/fail if we wanted).

In the next section I'll describe how we can conveniently put our results into a data set.

COLLATING THE RESULTS

Putting the test results into a SAS data set rather than the log is the next step to improve our efforts.

We can use the tag as a unique key for the test results data set so that the data set has the most recent test result for each and every test. We can re-run individual tests and have an individual row in the results data set updated.

To give us the greatest flexibility to add more macros to our test suite, we don't want the process of writing results to a data set to interfere with activities that are external to the macro. So, using a SET statement, for example, would inconveniently require the data set to be named in the DATA statement. This seems a good opportunity to use the OUTPUT method for a hash table. We can load the results data set into the hash table, use the TAG as the key for the table, and add/update a row with the result before outputting the hash table as the updated results data set. Here's the code:

```
%macro assert_condition(left,operator,right,tag=
                       ,resultdata=work.results);
  /* Load results into hash table */
  length Tag $32 Result $4;
  declare hash hrslt(dataset:"&resultdata");
  rc = hrslt.defineKey('TAG');
  rc = hrslt.defineData('TAG','RESULT');
  rc = hrslt.defineDone();
  /* Update the hash table */
```

```
    tag = "&tag";
    if &left &operator &right then
      result="PASS";
    else
      result="FAIL";
    rc=hrslt.replace(); /* Add/update */
    /* Write back the results data set */
    rc = hrslt.output(dataset:"&resultdata");
    rc = hrslt.delete();
  %mend assert_condition;
```

By adding the maintenance of the results data set to our basic assert macro, the functionality gets inherited by any higher-level macro (such as the previously-described %assert_EqualRowCount).

Clearly, the new macro won't work if the results data set doesn't already exist, and we'd like to present the results in a format better than a plain data set. We'll cover that in the next section.

PRESENTING THE RESULTS

We've created a simple, generic macro for testing and recording test results. All that remains now is for us to tidy-up some loose ends.

Firstly, the macro assumes data set WORK.RESULTS already exists. And it also assumes that the data set contains appropriate variables named TAG and RESULT. We can quickly arrange that by being sure to include a call to the following macro in our testing code:

```
  %macro assert_init(resultdata=work.results);
    data &resultdata;
      length Tag $32 Result $4;
      stop;
    run;
  %mend assert_init;
```

Finally, we want to present our results. We can do this easily with a simple call to PROC REPORT:

```
  %macro assert_term(resultdata=work.results);
    title "Test Results";
    proc report data=&resultdata;
      columns tag result;
      define tag / order;
    run;
  %mend assert_term;
```

Equipped thus, we can focus on our testing code, not the mechanics of collating and presenting results. For example, let's imagine we have some new code to test; the purpose of the code is to read a raw file (a.txt), create some computed columns, and write-out a SAS data set (perm.a). One of our tests is to check that the number of rows in the raw file matches the number of rows in the SAS data set. Here's our code to test this functionality:

```
  %assert_init;
  %include "code_to_be_tested.sas";
  %assert_EqualRowCount(infile=a.txt,outdata=perm.a,tag=T01-1);
  %assert_term;
```

We can make the results a tad more visual by colourising the pass/fail values:

```
  %macro assert_term(resultdata=work.results);
    proc format;
      value $bkres 'PASS'='Lime'
                   'FAIL'='Red';
    run;

    title "Test Results";
    proc report data=&resultdata;
      columns tag result;
      define tag / order;
    define result / style(column)={background=$bkres.};
    run;
  %mend assert_term;
```

This assumes you're using SAS Enterprise Guide. If not, you'll need to add some appropriate ODS statements around the PROC REPORT.

The downside of the macros as they stand at this point is that the results data set gets recreated every time we run the code. Maybe we don't want that because we want to collate test results from a number of separate bits of test code. So, finally, we can make the creation of the results data set conditional, i.e. if it doesn't exist we'll create, if it already exists then we'll leave it alone:

```
%macro assert_init(resultdata=work.results);
  %if not %sysfunc(exist(&resultdata)) %then
  %do;
    data &resultdata;
      length Tag $32 Result $4;
      stop;
    run;
  %end;
%mend assert_init;
```

VARIATIONS

Some of the foregoing was first delivered through a series of articles that I wrote in my blog at www.NoteColon.info. As my planned series on testing drew to a close, I got an email from Quentin McMullen with some very kind words about the `NOTE:` blog, but also some very erudite comments about my choice of parameters for my testing macros. Rather than paraphrase Quentin's comments, I decided to publish his email verbatim (with his permission). Here's the heart of Quentin's email, followed by a few brief comments from me.

> Just a quick thought:
>
> I have a similar macro to %assert_condition, but it only has one (main) parameter, &CONDITION, instead of three; &LEFT &OPERATOR &RIGHT. So it looks like:
>
> ```
> %macro assert_condition(condition,tag=);
>  if &CONDITION then
>    put "TESTING: &sysmacroname: TAG=&tag, OUTCOME=PASS";
>  else
>    put "TESTING: &sysmacroname: TAG=&tag, OUTCOME=FAIL";
> %mend assert_condition;
> ```
>
> So you can call it like:
>
> ```
> %assert_condition(incount eq outcount)
> ```
>
> or
>
> ```
> %assert_condition (age > 0)
> ```
>
> or
>
> ```
> %assert_condition ( (incount=outcount) )
> ```
>
> I tend to like the one parameter approach.
>
> The only tricky part is if you have an equals sign in the condition, you have to put parentheses around the condition so the macro processor does not interpret the left side as a keyword parameter. The nifty thing is that the parentheses also mask any commas, e.g.:
>
> ```
> %assert_condition(gender IN ("M","F") )
> ```
>
> Do you see benefits to the 3 parameter approach vs 1 parameter?

I do very much see the benefits of Quentin’s approach. His example, using the IN operator, is particularly well chosen. Rest assured I now adopt the McMullen approach!

FUTS – AN AUTOMATED TEST FRAMEWORK

I'll finish this paper by recommending a suite of SAS macros named FUTS (Framework for Unit Testing SAS programs) from Thotwave. These are available for free download after registering with the site (the download includes documentation and some examples of usage too). Developed by Greg Barnes-Nelson and colleagues, the macros are pure gold.

You can read background to the macros in the following SAS conference papers which chart the development and use of the macro (from their original incarnation as SASUnit through to FUTS):

- *Automated Testing and Real-time Event Management: An Enterprise Notification System*, SUGI 29, 2004
- *SASUnit: Automated Testing for SAS*, Phuse, 2004
- *Drawkcab Gnimmargorp: Test-Driven Development with FUTS*, SUGI 31, 2006

There is a degree of overlap between FUTS and the macros that I have described. I recommend FUTS, but some of my clients find its size intimidating, and others don't have authority to download anything (especially "code") from the internet.

CONCLUSION

Whilst new and more sophisticated SAS tools become available as years go by, the value, principles and objectives of testing remain constant. Writing good test cases is an investment; producing automated test cases increases efficiency and reliability. The hash table ias a good vehicle for writing test results from a macro without disturbing the input/output flow of the host DATA step.

AUTHOR BIOGRAPHY

Andrew is Managing Director of RTSL.eu, a leading European SAS specialist consultancy. Having first used SAS in 1983, Andrew's experience and knowledge covers breadth as well as depth. Andrew shares his experience through his www.NoteColon.info blog and has presented conference papers in the UK, Europe and USA on a variety of SAS topics.

CONTACT INFORMATION

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

Name: Andrew Ratcliffe

Enterprise: Ratcliffe Technical Services Limited (RTSL.eu)

Address: 5 Willow Close, Bexley, Kent, DA5 1QY, United Kingdom

Work Phone: +44-1322-525672

Web: www.RTSL.eu / www.NoteColon.info