PharmaSUG 2017 - Paper AD16

Migration to SAS Grid: Steps, Successes, and Obstacles for Performance Qualification Script Testing

Amanda Lopuski, Chiltern, King of Prussia, PA
Yongxuan Mike Tan, Chiltern, King of Prussia, PA

ABSTRACT

Through detailed planning and preparation, we set our sights on a successful migration of our global macros and Unix utilities to a Linux based system on SAS Grid. But how did our planning and preparations fare when it was go time?

In this paper we will share the steps taken to prepare for performance qualification testing of our global macro and utility scripts. We will reflect back on the successes and obstacles faced in this new environment during testing, the methods we developed to mitigate issues as they occurred, and how we prepared for clean execution in the VAL environment, then conclude with takeaways from this experience.

This paper will be useful to any group with sights similar to ours: migration to SAS Grid. It will provide valuable, first-hand insight into the migration with emphasis on performance qualification (PQ) testing of global macros and utilities.

INTRODUCTION

To ensure accurate and consistent functionality of macros and utilities, code needs to be tested in incremental steps. The tests are documented and scripts are archived for future reference. For migration to SAS Grid, these global macros and utilities need to be retested and, where needed, updated to match environmental changes. For us, that required collaboration across levels of experience. Chiltern's PQ script testing involved successes that increased the efficiency of the migration process as well as obstacles that required multi-faceted approaches to overcome.

This paper is broken down into three sections. First, the paper outlines steps taken to prepare and align components for the planned performance qualification testing activities. Next, we go into detail about items that allowed for efficiency and effective communications across international teams. Lastly, the paper explains challenges faced during the script execution phase and mitigation efforts that turned the challenges around for the positive. We conclude with takeaway highlights from the experience.

STEPS AND PREPARATIONS

In order for our new environment to meet the user requirements of our current UNIX system, test data was selected to recreate datasets and output tables, listings, and figures. This would ensure that as programmers transitioned from UNIX into Grid operating LINUX, the new environment would be ready for the programmers. We assembled a list of validated macros and utilities and assessed the current status of their test scripts. The macro or utility's functionality was paired with system user requirements to ensure the Grid environment would meet the company's required needs for operation. All supporting files necessary to test the macros and utilities were added to our testing environment.

For the assessment of existing test scripts, we reset the version control number to 1.0 in order to mark the migration from Unix to Linux. All Unix versioned documents remain archived and the version change clarified Linux validation documentation being referenced. This also serves to differentiate future update requests on the operationalized LINUX OS macro or utility code.

The final step of the preparation puzzle: testing resources. These were the people who will conduct the PQ testing that connect the determination of user requirements with the updates and revisions made to testing scripts. We assembled a group of programmers from beginner to advanced positions. And to make the testers' life a little easier from the start, testing trackers were created, log tracker spreadsheets to record issues, and contact groups were assigned so the team could stay in clear communication to make the process as efficient as possible.

SUCCESSES AND VICTORIES

As part of the macro and utility installation protocol, the PQ test scripts already existed, archived, showing initial installation, and any consecutive updates made while in use within the UNIX environment. The PQ test scripts did not need to be written from scratch in preparation for the migration. This saved the team substantial start-up time and allowed for resources to also devote time to other projects simultaneously during the start-up phase.

Along with having installation protocols and test scripts in advance, the scripts were easy to follow and written with detailed instructions. The scripts listed all components necessary to conduct the complete PQ process for that macro or utility. The script established the input files needed to complete the test script, so the programmer would know which files to open prior to start. The PQ document explained the test step objective, its execution criteria, and the expected result including screen shots and sample code.

The step's objective introduced a programmer to the macro or utility being tested, to the parameters it will need as input and could accept, and an accompanying checklist to fill out for each test step. The log files were labeled using TITLE and FOOTNOTE test code statements to distinguish each test in the log file. The layout made it easy to track down each test and log result as you scrolled through the log. The checklist was kept simple and used to record validation status clearly with a consolidated tracker as shown in Figure 1.

	Pass/Fail	Explanation
▪ Test 1	▪ Pass	
▪ Test 2	▪ Fail	Program terminated with unexpected error. See Log file at Line 45.
▪ Test 3	▪ Pass	

Figure 1. Example of consolidated result tracker

The ease of use allowed the team to broaden its resourcing pool, as mentioned earlier. The largest group of testers could include beginner programmers with a few experienced resources available for questions, debugging, and any additional training. The testers with the least experience executed the test scripts following along with the detailed steps, while more advanced programmers would investigate issues to determine what fixes were required. Organizing testing execution in this manner made use of all allotted resources and left minimal persons in limbo waiting for work to be assigned. This was also a major success for allotment of personnel and left time for the advanced programmers to continue to participate in alternate projects.

Other small victories had a big impact on keeping our timelines and progress moving forward. Because of an internal programming SOP that specifies the use of standardized file accessing code, only minor updates were needed to adjust functionality of the macro or utility testing program code. Here is an example using MYPATH as the global variable:

```
%global mypath;
  %let mypath = C:\SAS\;
     libname dataset "&mypath.study1\sasdata";
     filename program "&mypath.study1\program";
  %include program(testpgm1.sas);
```

This code demonstrates that all file references start with C:\SAS\. For migration to SAS Grid, we only needed to replace C:\SAS\ with our new system path. Wording of the test scripts also only needed small edits to match log textual differences. If the expected result looked like this:

```
Note: macro %helloworld is located in C:\SAS\study1\macro;
```

For approval using the Linux OS PQ test script with version reset at 1.0, the tester made a small change in the expected result to account for the system path:

```
Note: macro %helloworld is located in /MYSERVER/study1/macro;
```

This would show that the macro functions as expected, given the new environment. To make this change to the test plan, the programmer would confirm that the substituted value, MYPATH, is equal to the expected value from the Unix PQ test script, C:\SAS\. Below

```
Note: macro %helloworld is located in $MYPATHstudy1\macro;
```

The programmer can do that by including the following code at the end of a test program:

```
%put MYPATH=%sysget(MYPATH);
```

The put statement displays for the programmer to see what pathway update needs to be made in the PQ test script expected results to account for changes.

OBSTACLES TO OVERCOME

A majority of the PQ testing continued as expected and our successes certainly kept progress turning. We did run into a few tricks to watch out for. As part of migration, we upgraded to the latest versions of SAS and applications like Enterprise Guide (EG). Operating within the new upgraded versions produced unexpected issues: environmental application content and format differ between versions. The issues we encountered were with the ABORT statement and the CATALOG procedure.

From our legacy environment, the use of an ABORT statement within EG typically terminated the SAS *process*, however we found the ABORT statement would terminate the SAS *session* entirely. This issue primarily affected the expected results section of our PQ test scripts. Programs were expected to terminate with errors when testing for missing parameters, within a macro call. The test scripts needed textual updates to make the expected outputs match the exact output of the log.

The CATALOG procedure on the other hand required complete code manipulation. As a part of our validation and documentation process, we used the below code to create an output listing all macros called within the test program:

```
proc catalog catalog=work.sasmacr;
   contents;
run;
```

However, we discovered that within SASHELP.VCATALG, the MEMNAME changed from SASMACR to SASMAC1 in with the release of SAS v9.2. We could not update the catalog name within our code, because when the test program was executed in Linux, WORK.SASMACR was valid and produced expected output. We needed a new course of action to bypass the PROC CATALOG statement!

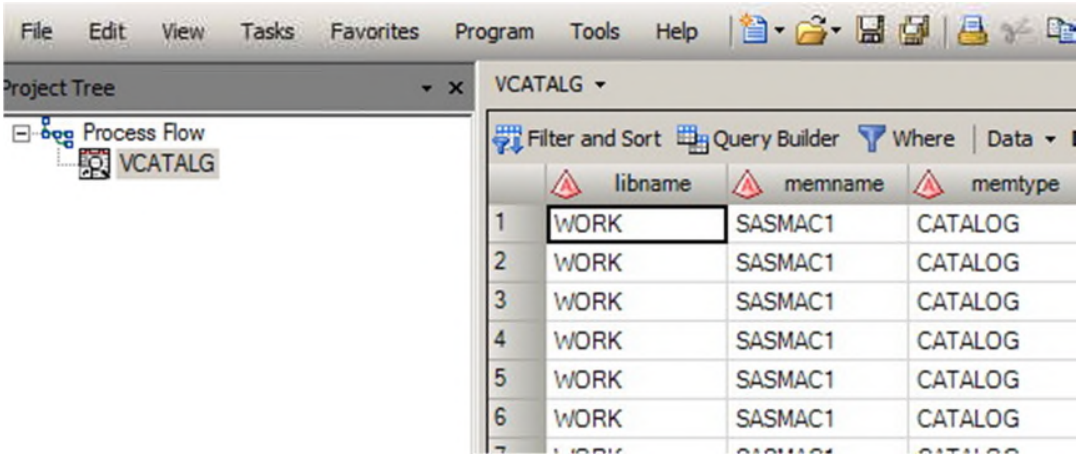

Image 1. Screen shot of SASHELP.VCATALG showing memname as SASMAC1

We rescued this mission by replacing the PROC CATALOG statement with the one below:

```
proc sql ;
  create table work.macros as
   select libname,memname,objname,created from dictionary.catalogs
   where objtype='MACRO' and libname='WORK'
   order by objname
   ;
quit;
```

The use of PROC CATALOG in our specific testing case was outputting an unexpected error that was corrected when we replaced that code with the PROC SQL code above. While determining how to fix the unexpected error, we were able to rule out that it stemmed from the macro malfunctioning. The specific code calling the PROC CATALOG was used to compile a list of macros used within the program for audit purposes. After this discovery, we could temporarily overlook this error until a resolution was discovered and continue testing as planned.

During the first round of testing, some of our macro tests failed with the same errors. We initially recorded these issues as a long list of bugs to work out. However, when we looked closely into the problems, many errors originated from the same code used within various core macros. To correct this issue, we created a hierarchy that indicated which macros depend on the successful completion of another. We also restructured our resourcing such that as inner macros reach a “pass” status, the resource could move on to the next inner macro until all inner macros reached “pass”. The primary macro was then tested and afterwards the entire group of macros could be deemed as “pass”.

As illustrated in Figure 2 below, in order for the top level to function properly, all the inner macros need to be functioning properly. When an item passes PQ testing, we marked it green to signal the next items can begin validation. When an item fails or is waiting for another macro to pass, we marked it red to suspend its testing and the testing of any items dependent on it. This helped to more precisely narrow down what issues were needs for code updates and what issues needed more in depth analysis.

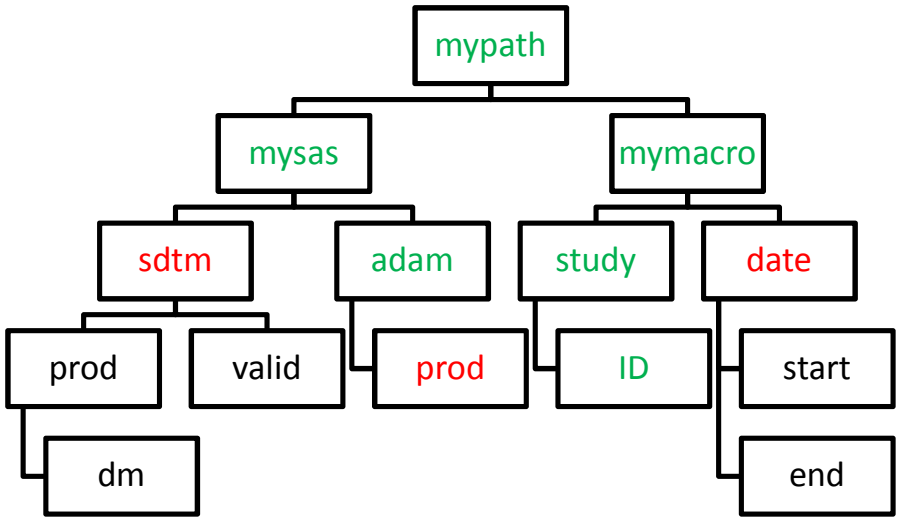

Figure 2. Hierarchy diagram indicates the interconnected relationship of global macros and an example of how we prioritized PQ testing based on general to specific use.

In summary, there were three obstacles we faced and the resolutions we implemented to overcome them. The first issue required a change to the expected output within our PQ test scripts. The second was a global change that affected all macros we tested. And the final issue we faced was restructuring the test script execution. As we learned, we adapted and responded with updates and code fixes to put us back on track.

CONCLUSION

To conclude this paper, Mike and I have created a few takeaway bullets from our experience. These were overarching topics that were central to the successful execution of our Performance Qualification test scripts. The first was the clarity of our test scripts. This clarity allowed us to broaden our resources pool and beginner programmers were able to perform testing independent of outside assistance and efficiently follow the detailed plan for validation. More importantly, the straight forward instructions greatly increased our flexibility to add additional testers quickly when needed. Since the majority of our testing resources could include beginner programmers, we successfully applied a “divide and conquer” approach for testing macros and utilities.

The utilization of a comprehensive tracker not only helped facilitate and organize a coherent validation strategy, but also integrated programmers with varying levels of experience to work in harmony. Despite the few hurdles we encountered, we were able to overcome them with ease and completed our PQ script testing in a timely manner.

Lastly, by recognizing an issue as systemic like the usage of PROC CATALOG in our specific case, we could identify that while our logs contained an unexpected error message, the macros themselves were functioning as expected. We could continue testing any parent macros or other items as planned to meet our deadlines.

CONTACT INFORMATION

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

Amanda Lopuski
Chiltern
Amanda.Lopuski@Chiltern.com

Yongxuan Mike Tan
Chiltern
Yongxuan.Tan@Chiltern.com