Paper 002-2010

A Robust and Flexible Approach to Automating SAS® Jobs under UNIX

Mike Atkinson
and
Ministry of Health Services
Government of British Columbia, Canada

Abstract

This paper describes how SAS® jobs are automated under UNIX at the BC Ministry of Health Services. A job is a set of SAS program(s) and/or Perl modules to be run in sequence. Job steps can be listed in a text file for the Perl script to run.

Having a well-tested perl script run the show has some advantages. Notification of job success or failure is reliable. Jobs can be restarted at the point of error. Logs are automatically date- and time-stamped. It is easy to migrate SAS code from development to production.

Introduction

The Unix server is a great platform for automating SAS jobs. It is very reliable, and capable of handling big jobs. Single programs can easily be scheduled to run when desired using crontab, built right into Unix. When it comes to multiple steps, however, crontab doesn't get you all the way there.

One approach to handling multiple SAS steps might be to use SAS itself to run other SAS programs. We have taken a different approach, using perl to handle the execution of multiple SAS steps.

This paper describes the use of a perl script that has been developed at the BC Ministry of Health Services to schedule SAS jobs on Unix. Here the term "job" refers to a set of SAS program(s) and/or perl module(s) to be run in sequence. Job steps can easily be specified in a text file for the perl script to run.

Some of the strengths in this approach to scheduling jobs are that:

- Multi-step jobs can easily be scheduled and run as a unit.
- With perl running the show, SAS steps have less to do and can be a bit simpler. Jobs can easily be organised into small, distinct SAS steps that are relatively easy to debug and maintain.

- Each SAS program is run as a new, separate task; programs start clean, with no datasets or macro variables left behind by SAS code that may have executed previously.
- The entire log of each SAS program is written to a separate file, right from SAS start-up messages through to the final note indicating the total CPU and memory used. File names of logs automatically include the date and time so that logs are not overwritten.
- Failure of a SAS step will always result in a Failure email message being sent; the failure message is sent by perl, which keeps executing after a SAS error. (The perl script receives the return code from each SAS step it executes, and sends a failure email after the first non-zero return code received.)
- Multi-step jobs can easily be restarted at any step within the job. This is very handy for those (rare!) occasions when a step has failed.
- Changes to SAS code (or entire new subsystems) can easily be migrated from development to production.

Directory Tree

Files on Unix are organised into directory trees, which can be effectively used to organise SAS code and associated logs, data, reports, et cetera. In our system, the path to the effective root of the tree indicates the environment: devl, prod, test, et cetera. Under the root for each environment (devl, prod, etc) is an identical structure for storing all of the items related to SAS jobs.

Here's a sample tree structure for an automated system producing monthly and daily reports. The trees for the development and production environments are shown side by side to demonstrate that the only difference is in the path to the tree root.

```
/path/to/devl
    /parms
    /bin
    /sascode
        /daily_reports
        /month_end
        /shared_code
            /macros
    /logs
        /daily_reports
        /month_end
        /shared_code
    /reports
        /daily_reports
        /month_end
        /shared_code
    /sasdata
        /daily_reports
        /month_end
```

```
/path/to/prod
    /parms
    /bin
    /sascode
        /daily_reports
        /month_end
        /shared_code
            /macros
    /logs
        /daily_reports
        /month_end
        /shared_code
    /reports
        /daily_reports
        /month_end
        /shared_code
    /sasdata
        /daily_reports
        /month_end
```

Some of these sub-directories (shown in bold above) are required by our job scheduling system; others can be created as needed based on system-specific requirements.

The `/bin` sub-directory contains all perl code, including the scripts that run jobs. Other useful perl modules (such as a module designed to FTP files) and any custom perl modules are stored here as well.

Our `/parms` sub-directory contains at least two files related to each job. A job requires a .run file and a .mail file in the /parms sub-directory, for instance: `daily_reports.run` and `daily_reports.mail`. The .run file lists the steps to be run; one step per line. Most steps are SAS programs, but perl modules can also be included as steps within a job. Parameter values can optionally be specified for SAS programs (or perl modules) on the line along with the name of the program.

The `/sascode` sub-directory has within it a separate sub-directory for each distinct system being automated. Each of the sub-directories can store a group of related SAS programs, which might represent a complete system or a sub-system of a larger system. Our convention has been to also create a sub-directory of /sascode named /shared_code, used to store SAS code shared between sub-systems. We put utility macros in a sub-directory beneath /shared_code named /macros.

The `/logs` sub-directory must contain the same set of sub-directories as the /sascode sub-directory; there will be a log directory for each system or sub-system. When a job is run, the log for each executed program is automatically placed into the appropriate /logs sub-directory, with the job start date and time both contained within the file name of the log.

The `/reports` sub-directory should also contain this same set of sub-directories. Reports will be stored in the appropriate sub-directory under `/reports`, again date and time stamped.

We have a bit more flexibility around the `/sasdata` sub-directory. Programs can choose whether to store each SAS dataset directly under the /sasdata directory, or under the appropriate sub-directory of /sasdata.

We make our production SAS programs completely portable between environments. It is really easy to migrate from development to production when migration does not require a single change to the code. When programs are run, they find out from the perl script running them whether their current environment is devl, prod, or whatever.

Scheduling jobs

The crontab facility is built-in to Unix, and allows you to indicate the date and time at which to execute commands. A convention we follow for scheduling jobs (under crontab) is that a change directory command precedes the call to the job scheduling

script. The cd command makes /bin the current directory, and is followed on the same line (after a semi-colon) by the command to run the perl script.

For instance, crontab might include the following (where the first line is a comment):

```
# minute hour day_of_month month day_of_week(0=Sunday)
# Run month end reports at 5am on the 1st of Jan, Apr, Jul, Oct
00 05 01 1,4,7,10 *    cd /path/to/prod/bin; run_job month_end_reports
```

When the run_job perl script is executing, it can easily determine the current directory – which in this case is /path/to/prod/bin. The perl script would be expecting the current directory to be of the form /path/to/*environment*/bin, and in this case would find it is running in the prod environment.

Given the current directory, the perl script will look in the directory /path/to/prod/parms for files named month_end_reports.run and month_end_reports.mail. The file named month_end_reports.run indicates the SAS programs to be run (and which sub-directory each SAS program belongs to).

This job could be executed directly at the command prompt (instead of using crontab) by entering the same run_job command while the current directory is /path/to/prod/bin.

Defining Jobs

As mentioned above, each job we run requires two files in the /parms directory, named *job_name*.run and *job_name*.mail where *job_name* is replaced by the name of a job. The .run file lists the steps of the job, in the order they are to be run. The .mail file lists the email addresses of people to receive an email notifying of job success or failure, and of people to receive emails with reports as attachments.

The .run file contains one line for each step in a job. Each line simply names the program to be run and the name of the sub-directory where the program will be found. Optionally, parameter/value pairs can be passed to a step. Either a SAS program or perl module can be called in each step: the file type (after the dot) is either sas or pm (perl module). Comments start with a pound sign. Here's a short sample job .run file that has been named month_end_reports.run:

```
# Sample job for producing month end reports
#
month_end/compile_data.sas    \  # Prepare data for reporting steps
         date_order=DESCENDING
month_end/daily_activity.sas     # Report transactions for the day
month_end/daily_ranges.sas       # Report showing range for each day
month_end/monthly_summary.sas    # Report totals by month
month_end/mail_reports.pm        # Mail reports created by this job run
```

You may have noticed that each line in this job starts with `month_end/`, prior to the name of the program. The part before the slash names the sub-system. The SAS code for the first step would be expected to be in the directory:
`/path/to/prod/sascode/`**`month_end`**`/compile_data.sas`

The trailing backslash (\) is a continuation character, indicating that anything on the next line (prior to #) should be considered as if it was part of the line with the continuation. The sample code above uses a continued second line to pass a value for the parameter date_order to the compile_data.sas program.

In this sample job, suppose that the 2nd, 3rd, and 4th steps (daily_activity.sas, daily_ranges.sas, and monthly_summary.sas) each produce one or more reports – which might be PDF reports, HTML, or simple text reports. The final step, mail_reports.pm, would email these reports to interested recipients.

Mailing Results

The perl module that appears as a job step most frequently (in fact, at the end of most of our jobs) is mail_reports.pm. Like other perl modules and scripts, it resides in the /bin directory.

The mail_reports.pm perl module will only send reports created by this run of the job, if successful. Reports for this run can be identified by the date/timestamp of the run, which appears in each report's file name.

A second filter is applied before reports are sent to each potential recipient. After finding reports with the right date/time stamp, pattern matching (using regular expressions) is applied to determine which of those reports will be sent to each recipient.

Both the list of the mail recipients and the patterns for each recipient are specified in the .mail file. Each line of the .mail file specifies an email address of a potential report recipient, followed by one or more regular expression patterns for that recipient.

In addition to regular expressions, we've implemented a couple of special characters, useful particularly for job administrators: exclamation mark (!) and greater-than sign (>). The exclamation mark indicates that a recipient will receive a notification of success or failure each time the job is executed. The greater-than sign indicates to send an email listing of the reports that were sent to each of the recipients.

Here is a sample .mail file, with comments, which in this case would be named month_end_reports.mail:

```
#
# Send all the reports, a list of who was sent what report,
# and the job summary report to the administrator
IT.Guy@unreal.net        ! > .
#
# Send all reports, but not the job summary or report summary
Samwise.Gamgee@unreal.net    .
#
# Send all PDF reports only
Dorian.Gray@unreal.net       pdf$ PDF$
#
# Send only the report of daily ranges
Jack.Worthing@unreal.net     range
```

SAS programs

When the run_job perl script is run, it will generate a command to run each SAS program. Here's what one of these commands might look like:

```
sas -autoexec /path/to/site/autoexec/autoexec.sas
     -log /path/to/prod/logs/month_end/compile_data.20090702.181835.log
     -print /path/to/prod/reports/month_end /compile_data.20090702.181835.lst
     /path/to/prod/sascode/month_end /compile_data.sas
     -sysparm prod:subdir=month_end:date_stamp=20090702.181835:run_instance=1:da
te_order=DESCENDING
```

The above would be a single, long command – it is only wrapped here because it is too long to fit on one line of this document. The sysparm parameter is one long string of characters with no spaces.

The first thing one of our SAS programs needs to do is to determine which environment it's being run in: development, production, et cetera.

The name of the environment appears before the first colon in the sysparm value, and colons are used as separators between other parm/values pairs, including the date/time stamp. The SAS program can use a single statement to set the macro variable &Env as the environment.

```
* Find whether we are running in DEVL or PROD environment;
%let Env = %scan(&sysparm, 1, %str(:));
```

Once a SAS program has set the macro variable &Env, it can safely %include other SAS code from the correct environment. It might start by including some code to perform additional initialisation (including reading the remaining parameters from &sysparm), such as in:

```
%include "/path/to/&Env/sascode/shared_code/initial.sasinc";
```

Our SAS code initial.sasinc reads the remainder of &sysparm (following the first colon), setting macro variables for each pair of variable=value found. In addition, initial.sasinc

does some generic initialisation, such as allocating macro and template libraries, and determining the current date.

Using the Current Date

Completely automated SAS programs need to do everything by themselves. Typically, an automated SAS program will use the current date to derive any other dates required for processing. The current date might also be used to look up other parameter values (perhaps stored in a table with effective date ranges) to find which values apply on the current date.

Normally the current date is picked up automatically by the SAS code, but it is possible to manually provide a different "current date". This might be done to run a program using dates and/or parameter values effective some time in the past. Note that current date is distinct from the date/time stamp; the date/time stamp is only used in creating file names, while current date is used to set parameters for the run.

We have the convention of including an underscore in all macro variable names – this makes it easy for our %print_macro_vals macro to pick out which global macro variables to write to the log. Here's some sample output from this macro:

```
************ User Macro Variable Values Shown Below ************
Note: Macro variables must contain one of: _

CURRENT_DATE = 20090702
DATE_STAMP = 20090702.182346
FIRST_DATE = 20090401
LAST_DATE = 20090630
RUN_INSTANCE = 1
TODAYS_DATE = 20090702
************ User Macro Variable Values Shown Above ************
```

Failure and Success

Every time a job fails, an email is sent to the appropriate person(s). It's great to get these failure messages! The only thing worse than receiving a message about failure of a job is not receiving a message about the failure of a job. This is where the perl script approach is particularly robust. The failure of a SAS program (even an abend) is captured by the perl script, which will fire off a failure email message and then cleanly exit itself.

If the example job failed, notification in the case of failure would be sent only to IT.Guy@unreal.net (who has the "!" symbol in the .mail file). Presumably he would fix the error and re-submit (and no one else would need to know that the first attempt had failed). The email indicating failure would look like the following:

```
Subject: **run_job month_end_reports FAILED**

run_job month_end_reports FAILED with return code 2

Command:.........run_job month_end_reports
Environment:.....prod
Job Name:........month_end_reports
Date Stamp:......20090702.182346
Start at step:...(beginning)
End at step:.....(end)

Following steps were run:

   Start  Return
    Time    Code  Command
18:23:46       0  month_end_reports/compile_data. sas         date_or\
                  der=DESCENDING
18:34:57 ******2  month_end_reports/daily_activity.sas
18:34:57 Failure Time

To run all steps from the failed step onward, you might use the
following command:

run_job month_end_reports daily_activity.sas 20090702.182346 &
```

The last line of the failure email can be very handy. Once you've corrected an error, you can restart the job at the step that had the error by copying the last line from the failure email and pasting at the command line (while in the /bin directory).

Specifying a date/time stamp like this is important when restarting jobs. Since the mail_reports.pm perl module (usually the last step of the job) will be using the date/time stamp to identify all reports that apply to this job, report files created both in steps before the error and after in steps executed after restarting the job, need to be named with the same date/time stamp. When we pass a date/time-stamp value to the run_job perl script, this date/time-stamp is used instead of setting the value based on the date and time the job was run.

When all the prior steps in the example job have completed successfully, the mail_reports.pm will email the report(s) to the indicated recipients, in addition to sending IT.Guy a message of success. The recipients would be:

IT.Guy@unreal.net	(who receives all reports)
Samwise.Gamgee@unreal.net	(who also receives all reports, but not the failure/success message)
Dorian.Gray@unreal.net	(who receives PDF reports only)
Jack.Worthing@unreal.net	(who receives reports containing "range" in the filename)

All reports created during a run which meet the pattern match criteria for a given recipient will be mailed (as multiple attachments of one email) to that recipient.

Let's assume the job executed successfully all the way through from start to finish, and all the above reports were sent. The IT guy would also receive an email listing all the reports that were sent to each recipient. He would also receive a job summary email such as the following, showing the execution time for each step in the job.

```
Subject: run_job month_end_reports was successful

run_job month_end_reports completed with return code 0

Command:.........run_job month_end_reports
Environment:.....prod
Job Name:........month_end_reports
Date Stamp:......20090702.181835
Start at step:...(beginning)
End at step:.....(end)

Following steps were run:

   Start  Return
    Time    Code  Command
18:18:35       0  month_end/compile_data.sas sas         date_o\
                  rder=DESCENDING
18:31:46       0  month_end/daily_activity.sas
18:37:24       0  month_end/daily_ranges.sas
18:41:46       0  month_end/monthly_summary.sas
18:42:55       0  month_end/mail_reports.pm

18:42:55 Completion Time
```

Our automation tool provides a great way to organise and execute our SAS code. It reliably notifies us of success or failure of jobs, and handles the distribution of reports.

The system has the kind of reliability that helps one relax while at SAS Global Forum.

Contact Information

Mike.Atkinson@gov.bc.ca
working with the British Columbia
Ministry of Health Services
Victoria, British Columbia
(250) 952-2405