Paper 080-2009

Keeping Up to Date: Using Build Tools with SAS®

Robert Burnham, Tuck School of Business, Hanover, NH

ABSTRACT

Complex SAS projects can have many program files, datasets and outputs that need to be managed. Build tools, such as Make, automate the updating of inter-dependent files and are used extensively in software development. Build tools function by observing when files are modified and applying internal rules to decide which programs need to be run in order to keep project results current. This tutorial starts with a simple example and then introduces new features that show how build tools can make SAS programmers more efficient at managing large projects. Versions of Make are available on almost all Unix/GNU Linux systems and instructions for setting up GNU Make on a Windows PC are included.

INTRODUCTION: USING BUILD TOOLS WITH SAS

Build tools work by looking at the times when files were modified and taking actions to update the files that are dependent on those changes. Take the simplest possible example, a three line SAS program named *hello.sas*:

```
data _null_;
  put "hello, world";
  run;
```

This program will write *hello, world* to the program's log file, which SAS names *hello.log* by default. The program's output is dependent on the instructions recorded in *hello.sas*. If the program is changed such so that it says *hello, joe* then it will need to be run again.

To use a build tool, such as GNU Make, we need to describe the relationships between the outputs (also called the *targets*), the programs, and any external data that is used. The build tool also needs to know the actions required to update the outputs. When using GNU Make, this information is written in a *Makefile*. Here is a first Makefile for the project:

```
hello.log: hello.sas
        sas hello.sas
```

The first line indentifies the target (*hello.log*) and its dependency (*hello.sas*) with a colon delimiting the two. The second line, indented by a tab character, is the action needed to bring the target up to date. To update the project the user types *make* at the prompt and sees the action, *sas hello.sas*, echoed back as SAS starts up and runs the program. If *make* is invoked again a different message appears:

```
$ make
make: `hello.log' is up to date.
```

A MORE REALISTIC EXAMPLE

Build tools become more helpful as a project grows. Consider a large file of financial data, e.g. daily stock returns from many firms, which are analyzed and reported on monthly. There are initially two files:

```
monthly_report.sas
stockreturns.csv
```

The program, *monthly_report.sas*, reads in *stockreturns.csv* and produces a report in PDF format called *stock_report.pdf*. In the parlance of the build tool, *stock_report.pdf* has two dependencies: *monthly_report.sas* and *stockreturns.csv*. This is expressed in a *Makefile* as:

```
stock_report.pdf: monthly_report.sas stockreturns.csv
        sas monthly_report.sas
```

Invoking *make* runs SAS and generates *stock_report.pdf* along with the expected *.lst* and *.log* files. The build tool checks the timestamps of both dependencies to see if either was updated more recently than the target and starts the action to update the report if necessary.

In addition to automatically checking the timestamps, the build tool has another benefit. Having a *Makefile* means that a user, who may not know SAS at all, can drop in new data and generate updated reports without knowing any of the steps involved.

The current system works but every month the data file becomes larger, processing time increases and business units send in additional requests for different types of analysis. Instead of reading and processing the raw *.csv* file each time that an analysis is required it now makes sense to load the data into a permanent SAS dataset, perhaps with indices, and have separate programs for the analysis. After refactoring a few things you have the following files in your directory:

```
Makefile
finance_report.sas
president_report.sas
process_stockreturns.sas
stockreturns.csv
```

The new process has a program called *process_stockreturns.sas* which reads, sorts and indexes the raw data and saves it as a SAS dataset named *stockreturns.sas7bdat*. Two additional SAS programs, *president_report.sas* and *finance_report.sas* generate PDF and RTF files respectively for the different consumers of the analysis. The *Makefile* has expanded to encompass all of these relationships:

```
stockreturns.sas7bdat: process_stockreturns.sas
        sas process_stockreturns.sas

president_report.pdf: president_report.sas stockreturns.sas7bdat
        sas president_report.sas

finance_report.rtf: finance_report.sas stockreturns.sas7bdat
        sas finance_report.sas
```

The new dependencies are represented, but now typing the *make* command only updates *stockreturns.sas7bdat* and not the other targets. A quick search through the *make* documentation indicates that targets can be specified as arguments, e.g. *make finance_report.rtf*, but that re-introduces some complexity. The solution is to define a *default* target, a list of outputs that the build tool should always update. Adding this line to the top of the *Makefile* solves the issue:

```
default: president_report.pdf finance_report.rtf

stockreturns.sas7bdat: process_stockreturns.sas
        sas process_stockreturns.sas

president_report.pdf: president_report.sas stockreturns.sas7bdat
        sas president_report.sas

finance_report.rtf: finance_report.sas stockreturns.sas7bdat
        sas finance_report.sas
```

Updating the reports monthly is very now straightforward. A change in a particular report will tell make to run only the program that creates that report while a change in the underlying data will trigger updates for all of the targets.

BEYOND BUILDING

Creating a *Makefile* for the project has automated the process of updating the monthly reports, but that is not the only task that it can handle. Makefiles are commonly used to automate other data handlings chores, e.g. cleaning out old log files and outputs so that the project is in a pristine state to update all of its targets. Many programmers will code a clean target in their Makefiles with no dependencies, for example:

```
clean:
        rm -f *.lst *.log *.rtf *.pdf *.sas7bdat
```

Typing *make clean* at the prompt will now delete all of the project's targets prompting a complete build the next time that *make* is invoked. In addition to a *clean* target, many *Makefiles* will include targets to install programs, build help files, or run tests.

USING VARIABLES

All of the commands in the *Makefiles* to this point have been hard coded, but in practice most *Makefiles* take advantage of a variable syntax that allows for more flexibility.

The current actions defined in the *Makefile* take the form *sas the_program.sas*, which works well when the version of SAS that we want to use is the first one in the current path. To use a different version of SAS installed in another location, or perhaps pass in command line arguments to SAS when it runs, would require changing each action in the *Makefile*. Variables provide a more flexible option. For example, the project is currently running on a Windows machine and having the splash screen and logo popup every time the project is updated is not desirable. Define a variable, which can be called *SAS* for clarity, at the top of the *Makefile* like this:

```
SAS = sas -nosplash -nologo -icon -sysin
```

The syntax to use a variable in a *Makefile* is $(variable_name), so all of the actions should be modified from this:

```
sas president_report.sas
```

To this:

```
$(SAS) president_report.sas
```

When invoked the build tool now echoes these commands:

```
$ make clean; make
rm -f *.lst *.log *.rtf *.pdf *.sas7bdat
sas -nosplash -nologo -icon -sysin process_stockreturns.sas
sas -nosplash -nologo -icon -sysin president_report.sas
sas -nosplash -nologo -icon -sysin finance_report.sas
```

This is a simple example of the power of variables in *Makefiles*, but it opens up a number of possibilities. For example, perhaps the project has certain key parameters for analysis, e.g. the beginning and start dates for reports. The top of the *Makefile* could be modified like this:

```
STARTD=01JAN2005
ENDD=31DEC2007
SAS=sas -nosplash -nologo -icon -set startd $(STARTD) -set endd $(ENDD) -sysin
```

The *set* command line option creates variables in the SAS environment which can be accessed using the %SYSGET macro. If the programs are modified to use those values in a WHERE clause, for example, then the build tool has become a very easy to use system for running different versions of analyses without ever having to actually edit the underlying SAS code.

PHONY TARGETS

The previous examples all listed outputs, dependencies and the actions needed to update the results. *Makefiles* also allow you to define “phony” rules which do not update a specific target but specify an order for the build process. For example, consider the invocation above:

```
$ make clean; make
```

In order to force a rebuild of the outputs we had to invoke *make* twice; once to build the clean target and delete the existing datasets and the second time to update all of the outputs. If we need to do this frequently then building a phony *Makefile* rule can simplify the process. A phony rule for this could be stated as:

```
force: clean default
```

Invoking *make force* would then be the equivalent of *make clean; make*. Defining phony targets can make the build process more legible and easier to use and maintain over time.

PREVIEWING A MAKEFILE’S ACTIONS

There are some times when it is advantageous to be able to see what steps the build process is going to take before running it. This is particularly true when you are debugging *Makefiles* and when run times are long. Most versions of make have flags such as *-n*, *--just-print*, *--dry-run*, or *--recon* that tells make to print out the actions it would take to update the targets. For example:

```
$ make clean
rm -f *.lst *.log *.rtf *.pdf *.sas7bdat

$ make --just-print
sas -nosplash -nologo -icon -sysin process_stockreturns.sas
sas -nosplash -nologo -icon -sysin president_report.sas
sas -nosplash -nologo -icon –sysin finance_report.sas
```

The output is pretty straightforward; after deleting everything the build system reports that it would need to run all of the programs to update the targets.

CONCLUSION

It is hard to overestimate the utility of using a build tool, particularly as projects get larger and more complicated. Aside from the efficiency that the tools give in terms of only running those programs which need to be executed to bring a project up to date; *Makefiles* serve as living history for projects documenting what they do and all of the inputs and outputs that they generate. In the immediate term *Makefiles* provide power and flexibility, over the long term when you are trying to recreate analyses that were conducted years ago with complex workflows, they are absolutely invaluable.

CONTACT INFORMATION

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

Name: Robert Burnham
Enterprise: Tuck School of Business at Dartmouth
Address: 100 Tuck Hall
City, State ZIP: Hanover, NH 03755
E-mail: robert.a.burnham@dartmouth.edu
Web: http://www.dartmouth.edu/~bburnham/

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.