Paper AD19

Fast and Efficient Updates to Project Deliverables: The Unix/GNU Make Facility as a Cross-Platform Batch Controller for SAS®

Brian Fairfield-Carter, ICON Clinical Research, Redwood City, CA
Stephen Hunt, ICON Clinical Research, Redwood City, CA

ABSTRACT

Concurrent development of interdependent parts is a common feature of complex programming projects. In clinical reporting, statistical summaries rely on derived datasets (which in turn rely on raw data), and are generated by programs often developed simultaneously with derivation programs. This concurrent development pushes components out of sync: programs must be re-run in sequence before updates to raw data and revisions to analysis datasets are reflected in statistical output. As projects grow in scope and complexity, the need to manage file dependencies increases. Lack of synchronization hampers validation and QC activities, imposes redundancy on the production of output, and threatens project timelines.

The Unix 'Make' facility (and its cross-platform analogue, the GNU 'Make' facility) is a generalized 'build system' designed to manage dependencies in coding projects. Described in general terms, the Make facility examines relationships among project files, and sequentially rebuilds 'target' files whenever pre-requisites to those files have changed. This paper introduces the potential of Make for managing dependencies in SAS® programming projects, and describes how to adapt Make as a SAS batch-controller under both Unix and Windows, to efficiently refresh output files (derived datasets and statistical summaries) when any pre-requisites (programs and datasets) to those output files have changed. Automation techniques for detecting dependencies, and for generating and testing 'makefiles' (specialized script files interpreted by Make) are also discussed.

The techniques and tools described should be adaptable to any SAS version and any platform, and will hopefully be of interest to SAS users of all levels of experience.

INTRODUCTION

In virtually any programming project there exists a set of relationships between the various components that can be loosely described as a 'dependencies hierarchy'. In SAS programming projects in the pharmaceutical industry, one form this hierarchy takes is that in which lower-order analysis datasets depend on raw/extracted data, and both higher-order analysis datasets and statistical output depend on lower-order analysis datasets. SAS code acts as the interface between the various levels: programs read data from lower tiers, and write output to higher tiers. These relationships are illustrated in Figure 1.

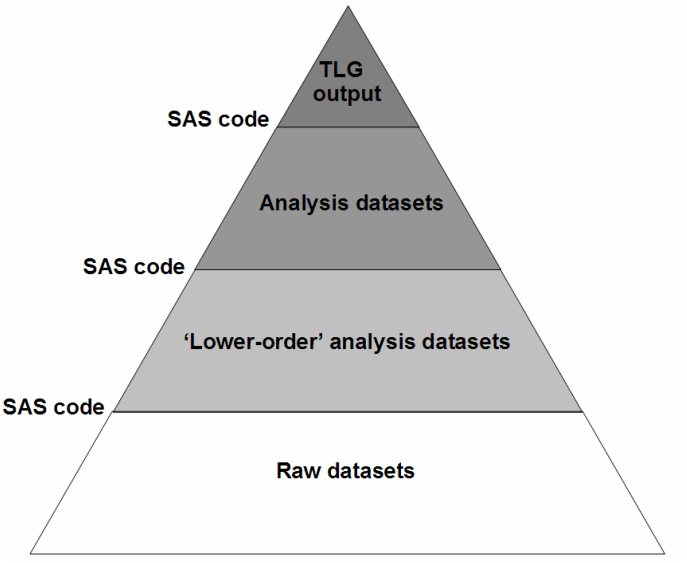


Figure 1. A typical 'dependencies hierarchy' in clinical reporting (TLG='Table, Listing and Graph').

At some point during the life a project, as the number of programs and the volume of output increase, the question will invariably arise how best to run 'batch updates'. How should the entire suite of programs be run, respecting built-

in dependencies and associated run-sequence? The ability to run 'batch updates' is important for a number of reasons:

- QC/Validation is hampered by different 'vintages' of output; if 'production' output is stale (produced from old analysis datasets and/or old raw data), discrepancies in QC output are confounded, since they may reflect genuine programming errors, or they may simply reflect differences in source data
- The time available to produce a deliverable is often limited, so output must be refreshed in a short period of time
- Refreshing output must not rely on individual programmers' knowledge of the required suite of programs or the order in which they must be run

SAS-BASED BATCH-UPDATE SYSTEMS

SAS provides a number of ways to sequentially run programs, the most common being the '%include' directive, which can be employed to create simple batch-update systems:

```
/*Update derived datasets*/
   %include 'd_demog.sas';
   %include 'd_ae.sas';
/*Update tables*/
   %include 't_demog.sas';
   %include 't_ae.sas';
```

'%include' is 'synchronous' (subsequent SAS statements are not executed until the '%include'd program completes), and simply runs each specified program in the current SAS session. This means that run-sequence can be enforced by hard-coding it into the %include list (in the above example, tables are not refreshed until all derived datasets have been refreshed), but since all programs are run in the same session it also opens up the possibility of 'artifacts' (work datasets and global macro variables producing unintended results in later '%include's). A comparable batch-update system, but one where each program is run in an independent SAS session, can be developed under Unix using 'SYSTASK COMMAND':

```
SYSTASK COMMAND 'sas d_demog.sas' WAIT;
SYSTASK COMMAND 'sas d_ae.sas' WAIT;
SYSTASK COMMAND 'sas t_demog.sas' WAIT;
SYSTASK COMMAND 'sas t_ae.sas';
```

(Under Windows, 'CALL SYSTEM' serves a comparable function). The 'WAIT' keyword prevents subsequent SYSTASK calls from being made until the preceding call has completed, and therefore enforces run-sequence in much the same way as '%include' does.

Systems like these are attractive because of their simplicity, and tend to work well for reasonably small projects. However, for large projects, and to achieve rapid and efficient batch-updates, a number of deficiencies exist, including:

- Updates that fail at some intermediate step must either be re-started from the beginning, or re-started from the fail-point; all subsequent code is re-run regardless of whether or not it was affected by the intermediate failure-point
- Files can not be refreshed individually (for instance, if all that is required is an update to a single table, there is no mechanism to trace back through the dependencies and update pre-requisites to the table)
- The system must be updated and maintained manually as the project grows and develops

BATCH-UPDATE SCENARIOS

Consider the following hypothetical collection of datasets and output files:

Raw Data	Analysis $Data_1$	Analysis $Data_2$	Analysis $Data_3$	Output Tables
DEMOG	DEMOG	LABS	ENDPOINT	DEMOG.out
LABS		VITALS		LABS1.out, LABS2.out
VITALS		ECG		AE1.out, AE2.out
ECG		AE		CONMEDS.out
AE		CONMEDS		VITALS.out
CONMEDS				ENDPOINT.out

Three levels of analysis datasets exist: the 'DEMOG' analysis dataset is created solely from raw data, while secondary analysis datasets such as 'LABS' and 'VITALS' depend on some combination of raw data and the lower-level 'DEMOG' analysis dataset, and the tertiary analysis dataset 'ENDPOINT' depends on some combination of raw data and both primary and secondary analysis datasets. Output tables may depend on analysis datasets from any of the three levels.

If a change is made to the 'DEMOG' analysis dataset, all other analysis datasets and all output tables must be refreshed, since all depend on the 'DEMOG' dataset. If, however, a change is made to any secondary or tertiary analysis dataset, the programmer must either refresh all files at all higher levels in the hierarchy (which may be very time-consuming), or else determine which specific higher-level files depend on the file in question. In some cases, validation documentation is associated with a specific file system date on the validated output file; performing 'targeted' updates limits the number of output files that are actually refreshed, and therefore limits repeated/redundant QC.

Conversely, if a request is made to "confirm that the 'ENDPOINT.out' file is up to date", the programmer has the option of refreshing all output (which, again, may be very time-consuming and inefficient), or tracing back through all the files on which 'ENDPOINT.out' depends and refreshing only those that serve as pre-requisites (or more specifically, refreshing only those pre-requisites that have lower-level pre-requisites that have changed since the last time 'ENDPOINT.out' was refreshed).

It's fairly clear that a trade-off exists: make no attempt at 'targeted' updates, and risk running a lot of code redundantly, or target specific files, and have to accommodate all the associated dependencies. A batch-update approach that makes use of a list of '%include' (or 'SYSTASK', or 'CALL SYSTEM') statements is amenable to the former of these; the question is, how can we achieve the latter using a similarly-automated system?

THE 'MAKE' FACILITY

The problems and trade-offs described above are by no means unique to the SAS world. A typical desktop application is created from multiple source files, often numbering in the hundreds, which must be compiled in a carefully-controlled 'build sequence' before eventually being 'linked' into an executable file. The need to manage these relationships and dependencies inspired the development of various utilities which can be loosely described as 'build systems'. Among these is the venerable Unix 'Make' facility which, very generally, updates files when any files upon which the 'target' files depend have been updated.

Relationships between source and output files, and commands required to create output files, are provided to the Make facility in what is for all intents and purposes a script, called a 'makefile'. In Make terminology, the makefile lists 'targets' (files to be created), pre-requisites on which the targets depend, and shell commands used to build the targets. Make executes these shell commands for targets where the file system date on any prerequisite file is later than that of the target file.

'MAKE' IS CROSS-PLATFORM: THE GNU SYSTEM, GNU MAKE, AND MINGW

GNU ("GNU's Not Unix") is an initiative to reproduce the functionality of Unix in an Open Source, General Public License (GPL) distribution. Probably the most widely recognized form of GNU is in the various GNU/Linux operating system 'distributions' (Red Hat Linux, etc.). GNU Make is a central component of the GNU initiative.

GNU Make, in turn, is not restricted to GNU/Linux distributions. For Windows systems, GNU Make is provided as part of MinGW (Minimalist GNU for Windows; see www.mingw.org), a Windows-specific subset of the GNU tool set. A simple command interface to MinGW is provided by MSYS (Minimal System; see www.mingw.org/msys.shtml). (Installation files for both MinGW and MSYS are available on the MinGW site).

GETTING STARTED: THE SIMPLEST POSSIBLE MAKEFILE

A makefile can be conveniently thought of as a 'script', but it's probably more accurate to describe it as a list of instructions describing how to create a script, where the script itself is the collection of executable commands.

TARGETS, PREREQUISITES, AND RULES

Each makefile contains, at a minimum, three key ingredients: targets, which are the names of files to be created or updated, prerequisites, which are all the files upon which each target file depends, and rules, which are the shell commands run to create the targets. Makefiles represent target/pre-requisite syntax in the following way:

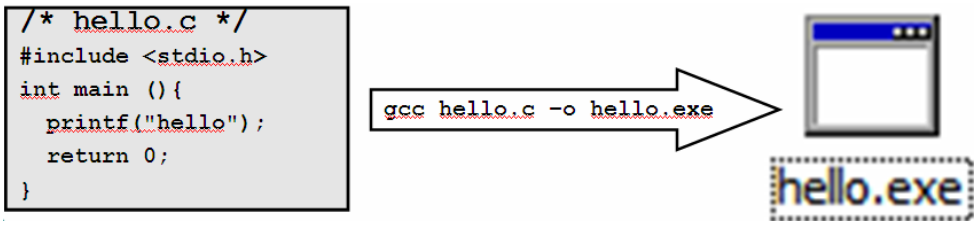
```
target : pre-requisite1 pre-requisite2 ... pre-requisite<n>
```

The name of the target file is given, follow by a full colon, followed by a list of pre-requisites. When a target is 'run', Make compares the file system dates on the target to that on all pre-requisite files, and if any pre-requisites have been modified more recently than the target, a command is executed to re-build the target. In order to 'run' a target, a shell command ('rule') must be supplied:

```
target : pre-requisite1 pre-requisite2 ... pre-requisite<n>
    COMMAND
```

The command must be preceded by a tab character, and can be any allowable shell command under the host operating system.

At its inception, Make was designed as a build system for compiled applications. For example, consider the following 'hello world' application in c:

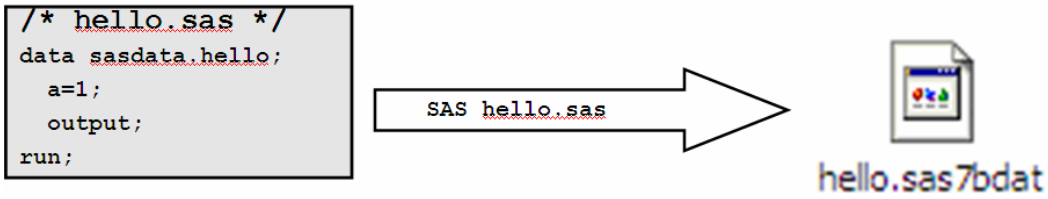


The source file 'hello.c' is compiled via the command 'gcc hello.c –o hello.exe' into the executable file 'hello.exe'. From this example we can infer that 'hello.exe' is a 'target' (output) file, that 'hello.c' is a pre-requisite file (actually, 'stdio.h' is also a pre-requisite, but we can probably assume that file remains static), and that 'gcc hello.c –o hello.exe' is the rule that builds the target. A makefile for this 'application' would therefore look like:

```
hello.exe : hello.c
    gcc hello.c -o hello.exe
```

ADAPTING MAKE TO A SAS ENVIRONMENT

This 'build process' might seem unfamiliar, but it's actually perfectly analogous to generating SAS output. Consider the following 'hello world' SAS program:



The source file 'hello.sas' is run in a batch SAS session, via the command 'SAS hello.sas', to produce the SAS dataset 'hello.sas7bdat'. As with the preceding example, we can infer that 'hello.sas7bdat' is a 'target' (output) file, that 'hello.sas' is a pre-requisite file, and that 'SAS hello.sas' is the rule that builds the target, so a makefile for this 'application' would look like:

```
hello.sas7bdat : hello.sas
    SAS hello.sas
```

This small but perfectly legitimate makefile can now be run by opening a command interface, navigating to the directory containing the makefile, and running the single command 'make'. This will invoke the Make facility, which will then by default look for and read a file called 'makefile' in the current directory (Make can also receive the name of the makefile as an argument, to over-ride this default, using the '-f' option: '`make -f <filename>`'), and interpret the first target. If the pre-requisite is newer than the target, the command is run.

A makefile usually consists of a list of targets and commands. For example, if we had several programs creating analysis datasets (where the 'ae' and 'vitals' datasets depend on 'demog'), the makefile might consist of the following:

```
demog.sas7bdat : demog.sas raw_demog.sas7bdat
    SAS demog.sas
ae.sas7bdat : ae.sas raw_ae.sas7bdat demog.sas7bdat
    SAS ae.sas
vitals.sas7bdat : vitals.sas raw_vitals.sas7bdat demog.sas7bdat
    SAS vitals.sas
```

Make treats the first target it finds as the 'default' target; with this makefile, the Make facility will stop executing once it has run (or at least checked) the 'demog.sas7bdat' target, since nothing in the default target has told it to look at any other targets. How is Make instructed to examine other targets? The simplest way is to create a default target that lists all other targets:

```
ALL : demog.sas7bdat ae.sas7bdat vitals.sas7bdat
```

With this as the first/default target, Make will check/run the 'demog.sas7bdat' target, and then carry on and do the same with 'ae.sas7bdat' and 'vitals.sas7bdat'.

While the targets in this simple makefile are listed in an order that matches the dependencies relationships ('ae' and 'vitals' depend on 'demog', meaning that the 'demog' target must be run before the 'ae' and 'vitals' targets), this order does not have to be hard-coded into the makefile. The tremendous power of Make lies at least in part in its ability to detect dependencies regardless of how the targets are ordered. Consider if the preceding makefile was written in this way:

```
ALL : ae.sas7bdat vitals.sas7bdat demog.sas7bdat
ae.sas7bdat : ae.sas raw_ae.sas7bdat demog.sas7bdat
    SAS ae.sas
vitals.sas7bdat : vitals.sas raw_vitals.sas7bdat demog.sas7bdat
    SAS vitals.sas
demog.sas7bdat : demog.sas raw_demog.sas7bdat
    SAS demog.sas
```

The first target that Make would ***test*** would be 'ae.sas7bdat', but this is not necessarily the first target that would be ***run***. Before running the target, Make would examine each of the pre-requisites, and if it determined that a pre-requisite was itself also a target (as is the case with 'demog.sas7bdat'), it would first test (and, if necessary, run) that target before returning to the original target. This ability to trace back through the dependencies hierarchy has a cascade effect: if Make tests the 'ae.sas7bdat' target and determines that it does not need to be run, but in the process determines that 'demog.sas7bdat' does need to be run (if, for example, the raw dataset 'raw_demog.sas7bdat' had been updated), once 'demog.sas7bdat' has been updated the dependency of 'ae.sas7bdat' on 'demog.sas7bdat' will cause Make to run the 'ae.sas7bdat' target (and, in fact, all targets that depend on 'demog.sas7bdat').

Although the 'ALL' default target is very useful, it can also be over-ridden. This would be done if, for example, only a specific file needed to be updated. If the 'vitals.sas7bdat' dataset had to updated for some reason, but it was not necessary to check or update any other datasets or output, Make could be invoked as follows:

```
make vitals.sas7bdat
```

The name of a specific target (in this case 'vitals.sas7bdat') is passed to Make as an argument. Make checks (and if necessary runs) the target (first checking and if necessary running any of its pre-requisites). Note that when Make is invoked in this way, no subsequent targets are run. This means that even though some pre-requisites to the given target may be run, and other targets may depend on those same pre-requisites, no other targets are run until Make receives an unqualified invocation.

In summary, even with a relatively unsophisticated makefile we are able to target specific files for update and at the same time accommodate all the associated dependencies.

ADDITIONAL COMPLEXITIES: VARIABLES, MULTIPLE DIRECTORIES, SPECIALIZED TARGETS, AND CLEANUP

The examples above are simplified for illustrative purposes; in reality targets and prerequisites may not all be stored in the same directory (or even stored under the same root directory). This means that the Make facility needs some way of finding files not contained in the 'current directory' from which the makefile is read.

As mentioned earlier, a makefile is essentially a script file, and in keeping with this the Make facility allows for a number of script-like characteristics, probably the most important of which is the ability to declare variables and do text substitution, much like with SAS macro variables.

Make has a reserved variable name "VPATH", analogous to the "PATH" system variable under Windows, which can be used to provide a list of directories to search for targets. If targets are stored in directories "../rawdata", "../derived" and "../tables", under a common root directory, VPATH can be set as follows:

```
VPATH = ../rawdata:../derived:../tables
```

When Make reads a target reference from the makefile, it searches through the directories specified in VPATH until it locates the file. This means that path references do not have to be hard-coded into target names.

Make variables can be used for text substitution, so that makefiles are easier to read and maintain. A very common application is in specifying rules (the shell commands that build targets). In the following example, a specialized shell script called 'CSAS913' (which can itself receive command-line arguments) is used to customize the SAS submission environment. Instead of hard-coding "CSAS913 -c" into the command for each target, a variable can be declared instead:

```
COMMAND = csas913 -c
```

This variable can then be resolved, using the '$' operator:

```
d_ae.sas7bdat : d_patfile.sas7bdat  \
          ae.sas7bdat  \
          d1_ae.sas
    $(COMMAND) d1_ae.sas
```

(Note also the use of the '\' character, which allows lengthy statements to be split onto multiple lines, also making the makefile more readable). Since the shell command is provided through text substitution, it can easily be changed. Depending on your environment, if instead of using the 'CSAS913' script you simply wanted to call SAS directly, all that would be required would be to change the variable 'COMMAND' to

```
COMMAND = SAS
```

Another simple but useful application of make variables is in 'cleanup' operations. In some situations it's preferable to remove all output files, or all output files of a particular type, before running Make. For example, a target such as

```
cleanfiles :
    -rm *.lst
```

simply removes all listing (.lst) files from the current directory. Since the various types of output files are usually not stored in the same directory, path names must usually be supplied in cleanup targets; declaring path names as make variables near the top of the makefile helps in makefile maintenance:

```
DERIVEDPATH = ../derived
TABLEPATH = ../tables
LISTINGPATH = ../listings

-------<Makefile body>-------

cleandata :
    -rm $(DERIVEDPATH)/*.sas7bdat

cleanoutput :
    -rm $(TABLEPATH)/*.rtf $(TABLEPATH)/*.out
    -rm $(LISTINGPATH)/*.rtf $(LISTINGPATH)/*.out
```

Invoking Make using the command '`make cleandata`' removes all SAS datasets from the 'derived' directory, and using the command '`make cleanoutput`' removes all output (.rtf and .out) files from the 'tables' and 'listings' directories.

Similar to the 'ALL' target, individual cleanup targets can be tied together in a 'cleanall' target; however, if any of the individual cleanup targets fails (for example, if no '.out' files exist in the 'TABLEPATH' directory), no further targets will be run. To work around this, Make allows for a special type of target, called a 'phony' target:

```
.PHONY : cleanall
cleanall : cleanloglst cleandata cleanoutput
```

Declaring ‘cleanall’ as a ‘phony’ target allows Make to execute subsequent targets in the ‘cleanall’ list after an individual target has failed.

It is beyond the scope of this paper to describe all the functionality supported by Make, but a final useful script-like aspect of Make is the reserved ‘subsystem’ target, which allows ‘external’ makefiles to be run in a Make session, analogous to the ‘%include’ directive in SAS:

```
subsystem:
    cd combined && $(MAKE)
```

In this example, the current directory is set to a sub-directory called ‘combined’, and the makefile stored there is then run.

TESTING MAKEFILES

When developing a makefile, it is usually preferable (and more efficient) to test targets and other functions without actually modifying any files, and without actually running any commands. Two simple techniques are introduced here that allow for this: the ‘touch’ utility, and the ‘-n’ Make option.

THE ‘TOUCH’ UTILITY

The ‘touch’ utility allows you to modify file attributes (i.e. modification date & time) without actually modifying the file. This means that ‘touch’ is very useful when testing a makefile, since it allows you to simulate file-modification scenarios, and then see how Make responds. As with Make, ‘touch’ was originally a Unix utility, but is now cross-platform, thanks to GNU and MinGW. Touch recognizes a number of options, and can modify file attribute information in specific ways (for example, to set the ‘last modified’ date-time to match that of another file), but for purposes of testing makefiles, the simplest syntax suffices.

```
touch ae.sas
```

This command sets the file system date-time for the file ‘ae.sas’ to the current system date-time. This means that if ‘ae.sas’ creates a target file (SAS dataset) ‘ae.sas7bdat’, it will appear to Make as though ‘ae.sas’ was modified more recently than the last time ‘ae.sas7bdat’ was refreshed, even though no modifications were actually made. To force an update to a specific target file, simply ‘touch’ one of the file’s pre-requisites and run Make.

THE ‘-N’ OPTION

For purposes of testing a makefile, the ‘touch’ utility is more aptly used in conjunction with Make’s ‘-n’ option. The ‘-n’ option echoes commands back to the console window without actually executing them, meaning it gives an indication of what Make thinks needs to be run in order to re-build target files. For a specific target a command like

```
make –n ae.sas7bdat
```

will write to the console window a list of commands necessary to update the target, but no commands will actually be run, and the target itself will not be updated. The ‘touch’ utility and the ‘-n’ option can provide quick insight into file dependencies: simply ‘touch’ a pre-requisite file, and run ‘`make -n`’ to see what targets are affected.

DEPENDENCIES IN SAS CODE & AUTOMATED MAKEFILE GENERATION

Using the basic ingredients shown so far, the process of writing a makefile for a relatively small and simple project wouldn’t pose a significant task. However, writing and maintaining a makefile for a large and complex project could make for a substantial undertaking. Consider a project with dozens, or perhaps even hundreds of source files being written and modified by several programmers. Each time code was added, removed, or modified in any way that altered the project’s dependencies profile, targets and prerequisite lists in the makefile would also have to be modified. Not only would this add programming overhead to the project as a whole, but it would pose a significant risk for error.

By defining rules for identifying dependencies among project components, a makefile-generating system can be developed that traverses the study directory structure, compiles a list of source files, and from these files determines the collection of dependencies and prerequisites. The ‘dependencies list’ can then be used to automatically generate a makefile.

DEPENDENCIES IN SAS CODE

From Figure 1, we can conclude that dependencies may exist between a number of different project file types:

- Permanent datasets
- Output files (.lst, .txt, .rtf, etc.)
- Macros
- '%include' files
- Format catalogs

The first question to consider is which of these dependencies need to be captured in a makefile; types of dependencies that need to be identified will depend on the nature of the SAS environment and specific project; for instance, for the sake of simplicity it may be possible to ignore autocall macros, under the assumption that they will remain static. If such an assumption cannot be made, it may be necessary to make autocall macros pre-requisites to certain types of targets.

Once the types of dependencies that need to be captured in a makefile have been determined, rules need to be developed for identifying these dependencies. References to permanent SAS datasets, identified in source code via the two-level '<library>.<dataset>' convention, can be identified according to a few fairly clear syntactical rules:

- `set <library>.<dataset>`
- `proc sort data=<library>.<dataset>`
- `out=<library>.<dataset>`
- `create table <library>.<dataset>`
- `select … from <library>.<dataset>`

These code fragments can be further classified into those that read SAS data (input dependencies, where the datasets serve as pre-requisites), and those that write SAS data (output dependencies, where the datasets serve as targets). For example:

- Input (pre-requisites):
  - set <library>.<data>
  - data=<library>.<data>
  - (proc sql) select ... from <library>.<data>
- Output (targets):
  - data <library>.<data>
  - out=<library>.<data>
  - (proc sql) create table <library>.<data>

How can these dependencies be detected in SAS source code? The simple answer is to use some sort of text pattern-matching, though simply finding strings that match the abstract pattern '<word>.<word>' will not suffice, since this will provide no information on whether the string refers to a pre-requisite or a target. A more complex abstraction must therefore be used, making this a task perfectly suited to the use of Perl regular expressions.

The SAS expression/text string '`set derived.data`' might be described in abstract terms as "a string starting with the word 'set' (or the word 'set' preceded by any number of blank spaces), followed by any number of blank spaces, followed by two words separated by a '.' character, followed either by the termination of the string or by any number of blank spaces". Such a description might then be represented as a regular expression:

```
(^set|\sset)\s\S+\.\S+
```

Using Perl regular expressions for complex pattern-matching makes it possible to parse out the various forms in which input/output dependencies take in SAS source code, and to classify these dependencies as pre-requisites and targets. These 'raw materials' can then be used to write target : pre-requisite lists, and generate a default 'ALL' target.

A simple Perl script, which uses a recursive directory tree search to find SAS programs under a root directory, and then iterates through an array of regular expressions to parse out a dependencies list from SAS source code, is included in the 'shellout' package available on SourceForge (https://sourceforge.net/projects/shellout/, Fairfield-Carter et al, 2006). Note that for the sake of simplicity, the script only attempts to capture references to permanent SAS

datasets, and relies on the assumption that statistical output will have embedded in the filename the name of the program that generated the output (for example, 't_demog(table.1).out' is generated by the program 't_demog.sas'). Dependencies other than those involving SAS datasets tend to show considerable variability between environments, and often involve 'hidden dependencies', so the approach will have to be tailored to the specific environment (techniques for addressing hidden dependencies are discussed in the final section).

CREATING A MAKEFILE FROM A LIST OF INPUT/OUTPUT DEPENDENCIES

As a companion to the Perl script mentioned above, a SAS program is also provided that reads the 'dependencies list', assembles the list of file names according to target/pre-requisite relationships, and writes a Unix/GNU makefile. Since the Perl script writes the dependencies list to 'standard output' (the console window), the SAS program uses a simple inter-process 'pipe', to capture the script output directly into a SAS dataset:

```
*** Pipe 'standard output' from Perl to SAS;
filename makefile "makefile.txt";
x "cd ../make";
filename x pipe 'pmake.pl';
data dpend;
  infile x lrecl=2000 truncover;
  input string $char2000.;
run;
```

In translating the list of dependencies into a makefile, the following general rule is employed: any output file written by a SAS program is a 'target', and any file on which that target file depends is a prerequisite. This means the SAS program itself is a prerequisite to any targets created by the program, and any file read by the program is also a prerequisite to the target. The makefile is written in a data _null_ step. Recall that in a makefile, a 'tab' character must precede each shell command; 'tab' characters are supplied using hexadecimal notation ('09'x):

```
put / '09'x "$(COMMAND) " program /;
```

The following is an excerpt from a generated makefile:

```
#
# Makefile generated by sasmake, 14JUN07, 16:43
#

VPATH=../tables:../listings:../derived

COMMAND = csas913 -c
PROGPATH = ../combined
DERIVED = ../derived
TABLEPATH = ../tables
LISTINGPATH = ../listings

ALL : \
     d_patfile.sas7bdat \
     d_ae.sas7bdat \
     d_conmed.sas7bdat \
     d_ecg.sas7bdat \
     -----(etc.)-----

d_patfile.sas7bdat : elgreg.sas7bdat  \
          demog.sas7bdat  \
          fup.sas7bdat  \
          visdt.sas7bdat  \
          d0_patfile.sas
    $(COMMAND) d0_patfile.sas

d_ae.sas7bdat : d_patfile.sas7bdat  \
          ae.sas7bdat  \
          d1_ae.sas
    $(COMMAND) d1_ae.sas
```

```
d_conmed.sas7bdat : d_infus.sas7bdat  \
          conmed.sas7bdat  \
          d_patfile.sas7bdat  \
          d1_conmed.sas
    $(COMMAND) d1_conmed.sas

d_ecg.sas7bdat : d_patfile.sas7bdat  \
          ecg.sas7bdat  \
          d1_ecg.sas
    $(COMMAND) d1_ecg.sas

-----(etc.)-----

.PHONY : cleanall

cleanall : cleanloglst cleandata cleanoutput

cleanloglst :
    -rm $(PROGPATH)/*.lst $(PROGPATH)/*.log

cleandata :
    -rm $(DERIVED)/*.sas7bdat

cleanoutput :
    -rm $(TABLEPATH)/*.rtf $(TABLEPATH)/*.out
    -rm $(LISTINGPATH)/*.rtf $(LISTINGPATH)/*.out
```

DEALING WITH HIDDEN DEPENDENCIES

'Hidden' dependencies arise when files are not referenced explicitly: where file names are obscured through the use of macro variables and macro calls. For instance, if output files are declared using a form such as

```
proc printto file="\\server\directory\file.out";
or
ODS rtf file="\\server\directory\file.rtf";
```

, then regular expressions and pattern matching can be used to identify and parse out the file name. In practice, output files are often not referenced in this way; 'macrotization' and the use of external metadata files often means that output file references take the form

```
proc printto file="\\server\directory\&pgmname&tablenum..out";
or
ODS rtf file="\\server\directory\&pgmname&tablenum..rtf";
```

The target name '&pgmname&tablenum..rtf' will have no meaning in a makefile.

Even permanent SAS datasets may not be referenced explicitly, but may instead rely on macro variables and text substitution. For example, processing that is repeated across a number of permanent/input datasets may be encapsulated as:

```
proc sort data=derived.&dset out=&dset;
  by &byvars;
run;
```

All the datasets referenced through the '&dset' macro variable are clearly pre-requisites to whatever file the program creates, but again the pre-requisite name '&dset' will have no meaning in a makefile.

These issues should be given some consideration when programming strategy is being developed, since the advantages of using Make may justify the deliberate adoption of coding conventions that minimize hidden dependencies. If hidden dependencies can't be avoided, then a number of possibilities exist for working around them, though the approach taken will of course have to be tailored to the specifics of the project and reporting environment. With dataset references made using macro variables, it may be possible to simply ignore the

dependencies completely; this would be true, for example, of a 'post hoc' macro called at the end of each derivation program that simply changed the sort order on derived datasets. In other cases it may be necessary to list a given macro as a pre-requisite to all derived datasets, or to list all derived datasets as pre-requisites to a given macro.

For output files, careful use of file-naming conventions will allow for some corner-cutting. For instance, as described earlier, the practice of embedding the name of the SAS program in the file name of any output file it creates can be quite useful. A list of filenames in the output directory can be generated, and compared against a list of SAS program filenames to determine dependency relationships. However, this again will require the use of regular expressions: consider the following programs and output files:

```
t_ae.sas          t_ae(Table.6).out
                  t_ae(Table.7).out

t_aeall.sas       t_aeall(Table.8).out
                  t_aeall(Table.9).out
```

A simple match between text strings will create the impression that 't_ae.sas' creates all four output tables, since the name 't_ae' appears in the filename of all four.

SAS log files can provide a rich source of information when developing strategies for working around hidden dependencies, since log files report the resolved values of macro variables, and report file and dataset references in predictable ways (in fact, for datasets, the specific phrase that appears in the log will indicate whether the dataset in question is a target or a pre-requisite). For example, a filename alias 'PRT1', which refers to an 'abstract' filename containing embedded macro variables ("&pgmname&tablenum..out"), will produce a log message of this form when the file is actually being read or written:

```
NOTE: The file/infile PRT1 is:
      File Name=/<directory>/t_ae(Table.6).out
```

The log message provides the resolved value of the macro variables, so the file 't_ae(Table.6).out' is recognized as a target produced by the program to which the log file belongs. Similarly, read/write operations on SAS datasets are clearly identified in the log:

```
NOTE: There were 45 observations read from the data set <library>.<dataset>
NOTE: The data set <library>.<dataset> has 180 observations and 5 variables.
```

For datasets referred to using macro variables, the resolved values of the macro variables appear in the log, giving the actual name of the dataset. Further, the wording of the log message is predictable, and makes it possible to distinguish between datasets that are read and those that are written.

Both of these last two approaches (file lists, and parsing log files) have the disadvantage that they require the log and output files to exist prior to generation of the makefile. While this is probably a fair assumption, since programs tend to be run multiple times during development, leaving behind log and output files, it nonetheless represents a potential weak point in makefile generation.

Finally, dependencies information can be maintained as part of a standard program header. This may greatly simplify the task of generating a makefile, but must be weighed against the additional overhead (and possible inaccuracy) involved in keeping program headers up to date.

CONCLUSION

Statistical reporting in SAS involves many of the hallmarks of more 'conventional' software development: source and output files can be numerous, and are related through a complex 'dependencies hierarchy' in which concurrent development occurs across levels. A 'build sequence' must be followed in which raw data is used to produce analysis datasets, and analysis datasets are used to produce statistical output. Validation activities, and the timely production of output require the use of an automated system to regularly refresh intermediate and output files. The Unix/GNU Make facility is a powerful, generalized, and well-developed 'build system' designed to manage interdependencies and build sequence in programming projects, and can be very effectively adapted for use in a SAS environment. Hopefully, by introducing the basics of adapting Make to clinical reporting in SAS, and to dynamic makefile generation, this paper will serve as a useful reference to anyone hoping to adapt Make for use in their own reporting environment.

REFERENCES

Fairfield-Carter, Brian, Stephen Hunt, and Tracy Sherman (2006). SAS, GNU and Open Source: An Introduction to MinGW Development Tools and Sample Applications. Proceedings of the 2006 Pharmaceutical Industry SAS Users Group Conference.

GNU Make documentation: http://www.gnu.org/software/make/

ACKNOWLEDGMENTS

The authors would like to thank ICON Clinical Research for consistently encouraging and supporting conference participation, and all our friends at ICON for all their great ideas and support, and particularly Jackie Lane and Robert Stemplinger for providing review comments and sharing their knowledge of the Make facility. We'd also like to thank Cara, PJ, Nicholas, Cole, and last but not least our good friend Tim, for his sense of humor, and for introducing us to the world of open source.

CONTACT INFORMATION

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

Brian Fairfield-Carter, Stephen Hunt
ICON Clinical Research
Redwood City, CA
Email: fairfieldcarterbrian@gmail.com