Stanford Exploration Project, Report 129, May 6, 2007, pages 247–252

Short Note

A New Build Environment for SEP

Ben Witten, Bill Curry, and Jeff Shragge

INTRODUCTION

The Stanford Exploration Project (SEP) has been at the forefront of computational reproducible research for many years. Beginning with the introduction of *active* (a-doc) and *interactive* (i-doc) documents by Claerbout (1990), SEP progressed to reproducible scripting using *cake* (Nichols and Cole, 1989) and archiving on CD-ROMS to ensure that entire reports, along with original codes and processing flows, could be distributed easily and cheaply. An additional change occurred around SEP-89 when the more flexible GNU Make software was introduced at SEP (Schwab and Schroeder, 1995). Testing of the reproducible workflows to ensure complete repeatability became common practice by SEP-77 (Prucha et al., 1999).

In the summer 2006, SEP began transitioning from the Make system to SCons (Softwar Constrution), an open-source software construction based on the Python scripting language. The transition was largely inspired by the work of (Fomel and Hennenfent, 2007), who released the RSF/Madagascar package for generating geophysical (and more general processing) work flows. A main goal of this project was harnessing the scripting power of Python into a package for generating and checking the processing flow rules. Using the RSF/Madagascar as a model, SEP adapted and reengineered certain aspects of the package that were not completely tooled for SEP's research, development and computer environment. One of the main goals was to facilitate an easy transition to SCons for all SEP personnel by retaining a lot of the Make functionality. However, due to the use of a different scripting language and reproducibility philosophy, we have included a number of new features reported herein.

INTRODUCTION TO SCONS

Recently, SEP decided to transition from Make to SCons build tool. Part of the motivation for this change was that SCons has numerous advantages over GnuMake:

- Build rules are written in the more flexible Python scripts;
- SConstruct files containing build rules are easy to learn and edit;
- Improved ability to check for dependencies and parameter changes;

- SCons has the functionality to create multiple build environments (e.g. 32 and 64-bit); and
- SCons can execute build rules in parallel.

The adapted RSF package also contains:

- Built-in *pdflatex* software that generates .pdf reports with the up-to-date SEG standard;
- An improved and efficient method for transforming LaTeX documents to html and wiki formats using the modern *latex2html* and *latex2wiki* programs.

We discuss each of these points in the sections below.

Python

Because SCons is written in Python, it has numerous advantages when writing processing flows, such as:

- Users can write their own build rules, tailored to their specific needs;
- Loops, conditions and string substitions are easily incorporated into the build file; and
- Lists can be used in build rules.

Dependency Checking

SCons uses MD5 signatures to check dependencies rather than timestamps. This improves dependency checking by:

- Ensuring updated build if the MD5 signatures changes, which occurs when any apcect of a file changes; and
- Including the build rules in the dependency check, so changing parameters in the build rule updates the MD5 signature.

COMPILING WITH SCONS

Because SCons is designed for software development, the amount of code written to generate the SEP SCons compiling enviroment is relatively small. However, in order to remain consistent with the philosophy of automatic generation of build rules for programs that is present in the SEP make environment, some deviation from a standard SCons environment was necessary.

The SEP SCons enviroment scans through C, Fortran77/90/95, Loptran, and Ratfor77/90 files present in the source directory. The source files are then scanned for files which they depend upon (listed in use or include statements) and build rules are automatically generated for all programs and objects in that directory. As such, no manual writing of compiling rules in the SConstruct is necessary, although the user may opt to and turn off the automatic build rule generation.

The current implementation of the build enviroment is contained in two files, *SEPDefs.py* and *SEPProg.py*. The first file contains definitions of variables in a build environment that are site-specific, such as the location of the compiler and compile options. The second file contains several subroutine calls that automatically generate the build rules for the code in the source directory.

Current limitations with the scanning of source files mean that there can currently only be one module per file, and the file name must match the name of the module contained in the file. This will change with time, of course.

BUILD RULES

It was important to keep the SCons syntax similar to the Make syntax so that everyone is comfortable using it. We have created rules for different types of files and figures. Below is an example of a Make rule and the same rule in SCons. The Makefile rule is given by,

```
RESDIR=./Fig
%.v: %.H
        <$*.H  Grey >/dev/null gainpanel=a pclip=100 title=$* out=$@
${RESDIR}/images.v: image1.v image2.v image3.v
        vp_SideBySideAniso image1.v image2.v image3.v > $@
```

We assume all ".H" files are already made. In the Make rule, we have the targets followed by a colon, then the dependencies. The next lines are indented and give the commands. Here we have two rules: one to make ".v" files from ".H" files and a second to combine the .v files into a single figure. The `%` are used as command line wild cards, `$*` call dependencies, such as RESDIR, `$@` is the target. Now look at the same rule in SCons.

```
RESDIR= ''./Fig''
ResultER(RESDIR+'images.v',['image1.H','image2.H','image3.H'],
         '''
         <${SOURCES[0]} Grey >/dev/null gainpanel=a pclip=100 title="Image 1" out=image1.v;
         <${SOURCES[1]} Grey >/dev/null gainpanel=a pclip=100 title="Image 2" out=image2.v;
         <${SOURCES[2]} Grey >/dev/null gainpanel=a pclip=100 title="Image 3" out=image3.v;
         vp_SideBySideAniso image1.v image2.v image3.v > ${TARGET}
         ''')
```

In the SCons rule, we have combined both of the previous Make rules. In SCons, first you give the target (or list of targets) then the dependency (or list of dependencies) and then the commands. The triple quotes mean that the command is broken up into more than one line. We can replace a string with a variable. In this case,"./Fig" has been replaced by RESDIR. Targets and sources can be referred to as `${SOURCES[i]}$` or `${TARGETS[i]}$` if there are multiple sources or targets, where i is the position in the list on the first line. If there is only on source or target they can be called in the build rule as `${SOURCE}` or `${TARGET}`.

In Make there was no way to distinguish between different build objects except with the RESULTSER/RESULTSCR/RESULTSNR listed at the top of the Makefile. In SCons we adapted rules from RSF for different objects:

- Flow is used to build intermediate files;
- Plot used to view intermediate figure files; and
- ResultER, ResultCR, ResultNR, which are easily-, conditionally-, and con-reproducible results.

The Result commands are used to create anything that should not be removed when a clean is used. ResultER is an easily reproducible result. If there is any change in the rule or the dependency that figure will be rebuilt. ResultCR is a conditionally reproducible result that has one of the following properties:

- require large or proprietary data set
- require parallel processing or special software
- or take 20 minutes or more to build

SCons handles a CR result by first checking to see if the file exists. If the file does not exist, then it is created using the provided build rule. If the file does exist, then it checks to make sure there are rules to build all of the dependencies for that file. If so, then nothing is done. If at least one rule to build a dependency is missing then the build aborts and gives an error message to that affect. ResultNR serves to let the "tex" portion know there is a figure and as a place to put commands for figures that could be build, but the data has been lost.

CONCLUDING REMARKS

In this paper, we describe the benefits of switching build environments from Make to SCons. We have also shown the basic framework of how SCons works for SEP and how a user can take advantage of SEP's SCons environment.

REFERENCES

Claerbout, J. F., 1990, Active documents and reproducible results: SEP–**67**, 139–144.

Fomel, S. and G. Hennenfent, 2007, Reproducible computational experiments using scons: 32nd IEEE International Conference on Acoustics, Speech, and Signal Processing.

Nichols, D. and S. Cole, 1989, Device independent software installation with CAKE: SEP–**61**, 341–344.

Prucha, M. L., R. G. Clapp, et al., 1999, Reproducible research - results from sep-100: SEP–**102**, 249–252.

Schwab, M. and J. Schroeder, 1995, Reproducible research documents using GNUmake: SEP–**89**, 217–226.