Stanford Exploration Project, Report 89, November 12, 1997, pages 217–226

Stanford Exploration Project, Report 89, November 12, 1997, pages 217–226

Reproducible research documents using GNU make

Matthias Schwab and Joel Schroeder[1]

keywords: *processing, computing*

ABSTRACT

The Stanford Exploration Project (SEP) publishes scores of reproducible electronic research documents each year. Traditionally, researchers at SEP have used the makefile dialect `cake` to maintain their documents. Because of `cake`'s limited distribution, we decided to replace it by GNU `make`. GNU `make` is an alternative make dialect that is widely used in software maintenance and is well documented. To allow an author to cope with document maintenance as opposed to software maintenance, Richard Stallman amended the original GNU `make` source code: a new GNU `make` special built-in target, `.SECONDARY`, enables the author to specify how to deal with missing files when checking if a target is up-to-date. Furthermore, the maintenance rules common to all reproducible documents were rethought when we translated them from `cake` to GNU `make`. The most significant change is the introduction of three result lists that distinguish between easily-reproducible, conditionally-reproducible, and non-reproducible results. A declaration of the results' reproducibility in the makefile simplified most of the crucial reproducibility rules (such as `make burn`, which removes all easily-reproducible results). We tested the new GNU `make` rules by translating Claerbout's three most recent textbooks. SEP plans to publish all its future reproducible documents using GNU `make`.

INTRODUCTION

For several years, SEP has published reproducible electronic documents. Such reproducible documents maintain all the software pertaining to a specific research project. A reader is supplied with a set of simple commands that enable him to remove all result files, to recompute all result files from source files, and to clean all secondary files, (i.e., those which are neither source nor result files). This set of commands helps authors to keep their documents up-to-date by recomputing their results regularly. It also helps readers by allowing them to isolate the source files and to observe the

[1]**email:** matt@sep.Stanford.EDU, schroedr@sccm.Stanford.EDU

recomputation of the results.

At the center of a reproducible electronic document is a makefile. All SEP makefiles share common include files[2] which define a set of standard rules and variables. Consequently, a reader can rely on these standard rules when interacting with any SEP document.

In the past, SEP used the make dialect `cake` (Nichols and Cole, 1989), which Somogyi specifically developed for document maintenance (**?**).

The cake language uses the C-preprocessor for variable and macro definitions. The author of a cakefile writes Prolog-like rules. Dependencies with expressions such as `cando target1` or `exist target2` make rules very brief and easy to read. Unfortunately, `cake` is not widely used and only very sparingly documented.

We decided to replace `cake` with GNU `make`, which is another make dialect. GNU `make` is widely used as a public domain software maintenance tool. It is well documented (Stallman and McGrath, 1991) and supported by a very active newsgroup, `gnu.utils.bug`. GNU `make` has its own internal variable substitution mechanism. Its rules tend to be more serial than `cake`'s, which is partially due to the lack of dependency features such as `cando` and `exist`.

Most of the standard GNU `make` rules, which are contained in the include files, are translations of the former `cake` version. One highly visible change is the new mechanism that declares a result file either easily-, conditionally-, or non-reproducible. According to the GNU `make` rules, the name-stem of all result files are listed in one of three lists: `RESULTSER` for easily-reproducible result files, `RESULTSCR` for conditionally-reproducible result files, and `RESULTSNR` for non-reproducible result files. When the author lists a target in one of these categories, he makes a promise to the reader about the reproducibility of that result file. The reader can verify the promise by attempting to remove and to recompute a result file using standard commands.

To improve GNU `make`'s suitability for document maintenance, Richard Stallman added the special built-in target `.SECONDARY`. When one deals with electronic documents, secondary files can be large and expensive to compute. A reader may want to check if a result is up-to-date with respect to its source files, even when some secondary files are missing. Standard GNU `make` considers a target always out-of-date if a secondary file, which is listed explicitly as a dependency, is missing. However, if the author includes the special built-in target `.SECONDARY`, then all targets are considered up-to-date even if any of their secondary files are missing. Thus, a target is only out-of-date when it is older than its source files.

We have tested the new GNU `make` document rules by translating Claerbout's three most recent books (Claerbout, 1992, 1994, 1995). Based on the success of these tests, SEP plans to switch to GNU `make` for its future reproducible documents.

---

[2] In this paper we refer as make to the general utility while we use GNU `make` or `cake` to refer to its particular dialects.

REPRODUCIBILITY

Motivation

With the development of the computer, science and engineering research has changed fundamentally. Computers are used to simulate experiments and process empirical data. These simulation and processing procedures are themselves researched intensely. But how should such complex computational processes be published and communicated effectively?

A dozen researchers at SEP develop and test methods to image the subsurface from seismic data. For years, SEP has published scores of research papers to record their findings and communicate them to the interested geophysical community. These written scientific reports often lack the implementation details a fellow researcher would need to reproduce and verify the published results effectively. Often the author himself cannot reproduce his own findings that he produced a few months earlier because parameter values committed to memory are forgotten or his processing sequences or program versions have changed. On the other hand, it is hard for an autor to formulate and maintain a scrutinizing listing of all implementation details. Additionally such a listing makes dire reading material for humans.

Solution

As a solution, SEP has developed a framework for electronic documents: a Unix file system that is organized so that a reader can regenerate all of a document's research results from a minimum set of source files using a few simple commands. This reproducibility is distinct from other forms of interactivity such as setting or changing a parameter of a particular process. The reproducibility of a document only comprises the ability to remove and subsequently reproduce the document's result files.

Concept

At the center of the reproducibility concept are the definitions of result, source, and secondary files:

- **Result files** are all files that the author uses to document his findings. At SEP, result files are often figures in various graphic formats, such as postscript.
- **Source files** are the minimum set of files required to reproduce all the result files. This usually comprises C or FORTRAN source code, TEX files, makefiles, and data input files.

- **Secondary files** are all intermediate[3] files which a reader creates when reproducing the result files from the source files.

A makefile – usually a source file itself – contains rules for isolating each set of files and for recreating the result files after their removal.[4] Ultimately, a reader can dissect the document and observe or alter its reassembly.

Basic functionality

A set of variable definitions and rules are included in every makefile to insure that a reader can rely on standard rules in any makefiles. Besides these inclusion statements, the makefile contains a list of the name-stems of all result files and a rule to create each result file.

The included standard definitions comprise three basic reader commands:

- `make burn` removes all result files mentioned in the local makefile's result list.
- `make build` recomputes all result files by exercising the local rules about how each result file is built.
- `make clean` removes all secondary files. The included standard makefiles contain the rule `jclean`, which identifies secondary files according to certain suffix naming conventions. Because of the crucial role of cleaning and the potential damage if an insufficient clean rule is supplied, every makefile author has to supply his own clean rule, which he can easily default to `jclean`.

These three commands define the interface a reader can expect to work for any electronic document. A reader can exercise these three basic tasks without knowing any details about the particular computations described in the local application makefile.

Usage

When inspecting an unknown electronic document, we usually burn all result files, clean the document of potential secondary files, and rebuild the result files. Having isolated the source files (after `make burn; make clean`), we then watch as the makefile accomplishes the reproduction of the document's results (`make build`).

[3]GNU `make` defines *intermediate* slightly differently. See **Secondary targets** below.

[4]Besides the document specific files mentioned in the makefile, the electronic document basically expects a standard Unix environment. Thus, C and FORTRAN compilers, X Window libraries, and GNU `make` are not part of the result, secondary, and source classifications. The makefile expects them to be available to a reader and always up-to-date.

The same mechanism allows an author to test his own electronic document frequently. He can burn and rebuild the entire document even without remembering a certain executable invocation or a particular parameter setting.

Furthermore, the ready-to-use makefiles of an electronic document invite a reader to change source files and to observe the effects on the result files after burning and building them.[5]

The reproducible electronic document creates an environment in which the author shares his research to the fullest extent with a reader. He provides a general description of his research in the form of a (often online) report. Additionally, he provides a complete set of files which allow a reader to reproduce his results in a transparent and standardized fashion: no secret parameters, no incomplete set of source files, and no forgotten processing sequence. Overall, the reproducible electronic document facilitates an increased efficiency in maintaining, communicating, and reusing an author's research.

IMPLEMENTATION

Interface

A set of standard rules and variables are included to facilitate a standard behavior for all of SEP's electronic documents. The standard rules are grouped in a set of files according to their functionality. The most important ones are

- `SEP.obj.rules` and `SEP.prog.rules`, which contain standard compilation and link rules.
- `SEP.tex.rules`, which contains rules to maintain and display TEX documents.
- `SEP.action.rules`, which defines a basic interaction interface for authors.
- `SEP.fig.rules` and `SEP.clean.rules`, which include the rules fundamental to reproducibility, such as `burn`, `build` and `clean`.

An author may choose to include individual files in his makefile. In general, however, an author need only include `SEP.top` at the top and `SEP.bottom` at the bottom. These two makefiles, in turn, include all the other standard makefiles. The order of precedence for rules and variables in GNU `make` is complex (Stallman and McGrath, 1991). Therefore, we have carefully structured our system of standard include files so that the author of the local application makefile can overwrite any of the included default rules.

[5] A change in the source code or the default parameter values is where reproducibility ends and interactivity starts.

Following GNU `make` recommendations an author should use only lower case letters for the variable names in his makefiles. To avoid conflicts between SEP's standard variables and an author's local variables, we use stricly upper case variable names in the standard makefiles. An author should use lower case letters for his own variable names, reserving upper case to reference or deliberately override one of our supplied standard variables. Unlike some other `make` dialects, GNU `make` variables are easy to identify since they are of the form `${variable-name}` everywhere except in their definition statements. Therefore, we believe that restricting oneself to lower case variables should be overly constraining.

Degree of reproducibility

Instead of a single result list, GNU `makefiles` of SEP documents typically define three result list variables. These variables differ from other standard variables in that they are *meant* to be defined by the author. These variables, any of which the author may omit if empty, are `RESULTSER`, `RESULTSCR`, and `RESULTSNR`. The endings `ER`, `CR`, and `NR` indicate the degree of reproducibility to the reader:

- **easily-reproducible** describes result files which the reader can expect to regenerate within less than ten minutes on a standard workstation.
- **non-reproducible** are all the result files which a reader cannot reproduce, such as hand-drawn illustrations or scanned-in figures.[6]
- **conditionally-reproducible** indicates results which require special software or a lavish amount of time for recomputation. In both cases, the author nevertheless supplies a complete set of source files and rules to ensure that the results are reproducible. However, he accompanies the conditionally-reproducible result file (e.g., `myres.ps`) with a warning file (`myres.warning`) in which he states why he classified the result as conditionally-reproducible. In applied geophysical research, many interesting results fall into this category.

The standard document command `make burn` removes all easily reproducible results exclusively. The invocation of `make burnall` removes easily and conditionally-reproducible result files. The distinction between the three types of result files enables a reader to choose his burning and rebuilding according to his computer environment's special software and his own time commitment.

The standard targets `burn` and `jclean` remove files based on naming conventions. An author needs to inspect these carefully before using them. If the GNU `makefile` contains `RESULTSER = myres`, then `make burn` attempts to remove a series of potential result files `${RESDIR}/myres.ps ${RESDIR}/myres.v ${RESDIR}/myres.v3`,

[6]Non-reproducible figures are listed mostly for administrative purposes. It can be very useful to find all result files of a specific document.

where RESDIR is set to the directory in which the result files reside. An invocation of the target jclean will remove any file ending in .H .v .v3 or any file name beginning with the phrase junk in the current directory. Consequently, if ${RESDIR} is defined as ".", and the makefile author uses jclean in his clean rule, make clean will accidentally remove all result files.[7] Thus, if jclean is to be used, the result files should reside in a directory different from that of the makefile. Any author of makefiles will have to study the burn and clean rules as well as the associated naming conventions to avoid removing valuable files accidentally.

The build rule in SEP's standard include files attempts to update all result files: easily-reproducible, conditionally-reproducible, as well as non-reproducible result files. This can have the surprising effect of recomputing files which one has assumed are existing and up-to-date. Such surprising recomputations can be caused by common dependencies of rules about conditionally- and easily-reproducible result files such as,

```
cr.ps : easy.v difficult.v
er.ps : easy.v
```

If er.ps is built, and a secondary file easy.v is created, then cr.ps is out-of-date since its dependency easy.v is younger.

Document levels

An electronic document can comprise a single directory with one or two result files, a scientific paper containing a dozen figures (i.e., result files) with several sections organized in subdirectories, or an entire book containing scores of figures divided into chapters and sections.

Ideally, each section of a document contains its own GNU makefile and offers a reader the standard reproducibility commands. If an author wants to minimize his effort, he is only required to supply the most low-level makefiles needed. All higher level makefiles are created by a script, makemake, which summarizes the information of all makefiles in the immediate subdirectories. For example, makemake can create a book-level makefile from chapter-level makefiles in subdirectories which themselves may have been generated from section-level makefiles. These script-generated makefiles update themselves when a reader invokes any of their rules. This hierarchical document organization facilitates an easy separation and merger of document parts.

[7] In the near future, we hope to replace the wildcard formulation of the clean rule with a formulation based on GNU make's notion of secondary files.

Secondary targets

The reproducibility of an electronic documents requires the source files. The representation of a document is composed of the result files. The secondary files are used only in the process of generating the result files from the source files.

Secondary files can be prohibitively large, and they generally clutter the system. For these reasons, they are typically removed soon after they perform their function of helping to create result files. If a result file were considered out-of-date when one of its secondary files was missing, we would need to retain the secondary files to avoid redundant recomputation of the result files. Therefore, we wanted GNU `make` to consider result files with missing secondary files to be up-to-date. This would make it possible to freely remove secondary files.

Standard GNU `make` distinguishes between secondary files associated with pattern rules (*intermediate targets*) and secondary files associated with non-pattern rules (i.e., rules that do not involve pattern matching). In the case of a missing secondary which is the target of a pattern rule, GNU `make` considers an associated result file to be up-to-date. On the other hand, for a missing non-pattern secondary target, such a result file is out-of-date.

At our request, Richard Stallman added a special built-in target to GNU `make`, `.SECONDARY`, which allows the author to choose the aforementioned behavior of GNU `make` with respect to its missing secondary targets. Any target listed as a dependency of `.SECONDARY` is assumed up-to-date when missing. Listing `.SECONDARY` without any dependency ensures that every missing secondary file is presumed up-to-date. At SEP, the special built-in target `.SECONDARY` is included with the `SEP.defs.top` file, causing GNU `make` to assume a target with any missing secondary files is up-to-date. In short, `.SECONDARY` makes the behavior of pattern and non-pattern rules the same with respect to secondary files.

CONCLUSION

We successfully equipped three of Jon Claerbout's books with makefiles. Most of the translation from `cake` to `make` was done by a shell script which left only little to hand-editing. All three books have been tested by frequent burning and rebuilding on different platforms. We hope that soon all future SEP documents will use GNU `make`.

ACKNOWLEDGMENTS

We are continuing to work with Richard Stallman to adapt GNU `make` even more to the needs of electronic document maintenance. We appreciate his advice and his implementation of the special built-in target `.SECONDARY`. Martin Karrenbach implemented a TEX macro which extracts the reproducibility information `(ER,CR,NR)` from the makefile and indicates it in each figure caption. Steve Cole and Dave Nichols implemented a mechanism variable in their xtpanel.builder which made it a breeze to use GNU `make` in the standard xtpanel document interface. We thank Mihai Popovici for his courage in testing an early version of the GNU `make` SEP rules in his thesis. Finally, we want to acknowledge the pioneering work of Jon Claerbout, Martin Karrenbach, Dave Nichols, and Steve Cole which made this research project *almost* a pure translation from SEP's original cake rules.

REFERENCES

Claerbout, J. F., 1992, Earth Soundings Analysis: Processing Versus Inversion: Blackwell Scientific Publications.

Claerbout, J. F., 1994, Applications of Three-Dimensional Filtering: Stanford Exploration Project.

Claerbout, J. F., 1995, Basic Earth Imaging: Stanford Exploration Project.

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

Stallman, R. M., and McGrath, R., 1991, GNU Make: Free Software Foundation.