Making scientific computations reproducible

Matthias Schwab, Martin Karrenbach, Jon Claerbout [1]

ABSTRACT

Commonly research involving scientific computations are reproducible in principle, but not in practice. The published documents are merely the advertisement of scholarship whereas the computer programs, input data, parameter values, etc. embody the scholarship itself. Consequently authors are usually unable to reproduce their own work after a few months or years. At university laboratories such as ours, junior students are delayed in their own creative research by attempting to reproduce and leverage a former students research. A reader interface when accompanied by a complete set of source files, overcomes these difficulties: In the electronic version of such a document, a reader can reproduce and verify the document's results by invoking a set of makefile targets: the `burn` target removes the result files (usually figures), `build` recomputes them, `view` displays the figures, and `clean` removes any intermediate files, that were created when computing the result files. A reader can study the interaction of the various source files while recomputing the result files. An author of a research document, who uses a GNU `make` file to maintain his project software, can easily adopt our reader interface: The author needs to adhere to some naming conventions and to include a small set of files (150 lines of code which we distribute on our World Wide Web site). Since five years we have successfully used this reader interface in our daily research work at our laboratory, in 12 theses, and in 3 textbooks.

INTRODUCTION

Review

In our laboratory (the *Stanford Exploration Project* or *SEP*) we noticed that after a few months or years, researchers were usually unable to reproduce their own work

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

without considerable agony. Furthermore in average two PhD students graduate each year at SEP. After their graduation, new students who wanted to leverage their work, often spent a considerable amount of time to merely reproduce their colleague's results. We believe that similar inefficiencies plague most research involving complex scientific computations, such as image processing, operational research, or seismic data processing. It is this lack of reproducibility that impedes those who seek unselfish cooperation in research.

Motivated to overcome this impedance to cooperation our laboratory developed the concept of a reproducible electronic document (*red*): We added three crucial components to our computational repertoire: makefiles, naming conventions for figures, and a small set of commands respectively `make` targets.

Makefiles are the standard method of maintaining software on UNIX computers. A makefile contains the commands to build the different components of a software package. However, a makefile is more powerful than a simple script of commands since it has a notion of result files (*targets*) being up-to-date when they are younger than their corresponding source files (*dependencies*) that they are derived from. Since maintaining a reproducible research project essentially equals maintaining software, we find the make utility crucial to solving our problem elegantly. Fortunately, fine tutorial books about the `make` language (Stallman and McGrath, 1991; Oram and Talbott, 1991) exist and students learn `make` easily even without such tutorial books.

The second component in making scientific computations reproducible is to declare a name for each result file (which at SEP is invariably a figure) such as `myresult.ps`. For each result file the stem of the result name, `myresult`, and a rule to build the result file is listed in the application's makefile.

The third step in making scientific computations reproducible is to define a small set of standard commands that allow a reader to interact with the document without knowing any of the underlying application specific commands or files. This article discusses SEP's design for such a `reader interface`.

In the past, SEP used the make dialect `cake` (Nichols and Cole, 1989), which Somogyi specifically developed for document maintenance (Somogyi, 1984). We decided to replace `cake` with GNU `make`, which is another make dialect. GNU `make` is widely used as a freely available, platform independent 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` (?).

Having learnt about Jon Claerbout's reproducible research ideas, Buckheit and Donoho(1996) at the Stanford Statistics Department developed a reproducible research environment *Wavelab*. Wavelab is a library of *Matlab* routines for wavelet research. In contrast to SEP's reader interface, Wavelab is confined to Matlab and does not define a set of universal reader commands. However, by operating entirely within Matlab their reproducible research is portable to any computational environment that supports Matlab, such as PC, Mac, or UNIX computers.

Claim

We claim in this article that additionally to a complete set of application files, a small set of standard commands is needed to give a reader access to an author's research. This small set of standard commands enables a reader to remove and recompute a document's results without knowing the diverse, underlying computational details.

SEP developed such a set of standard commands for a UNIX environment using GNU `make`. Researchers who already use `make` to store their processing commands can adopt SEP's reader interface by adhering to certain file naming conventions and by including a few additional rules, which SEP distributes freely at his World Wide Website.

Preview

This article presents SEP's reader interface for reproducible electronic documents. The first section, *Problem*, describes the lack of reproducibility in scientific computational research. The following section *The Reader's gain* introduces the four commands that constitute the interface, their usage, and benefit from a reader's point of view. The section *The Author's Effort* outlines the naming conventions and GNU `make` rules an author of a reproducible electronic document needs to supply. Additionally, the section elaborates briefly on potential difficulties caused by differing compute environments of the reader and author. The section `Example` illustrates the reader interface and its implementation at a small but realistic example. The example concerns a finite-difference approximation of the wave equation. This section details SEP's implementation of the four basic reader commands. Finally the section `Guidelines for shared rules` introduces three design principles that we used when implementing the standard reproducible electronic document `red` rules.

PROBLEM

The development of a common reader interface at SEP was motivated by a loss of research and programming effort every time a graduate student left the group after graduation. Such a student publishes his thesis which carefully outlines his approach and illustrates its success with a series of figures. Unfortunately, he usually leaves an *electronic battlefield* neatly stored within a directory tree: Various input data sets, each preprocessed slightly differently, half a dozen versions of some crucial subroutines, maybe a few makefiles with some compilation rules, if lucky some `README` files with some cryptic notes, a bundle of precious result files mixed with a horde of unimportant intermediate and temporary files. A junior student who wants to refine or apply his colleague's method faces the Sherlock-Holmes-like task to organize the inherited files: He needs to identify valuable source or result files and remove their insignificant brethren. He has to reconstruct the command sequence that produced a specific result by combining particular input data, parameters, and program versions. Such a reconstruction of the software of the former student is often a significant effort during which the junior student is not creatively researching new ideas. We believe the difficulties to regenerate a colleagues results trouble any field of scientific computations, such as geophysics, image processing, or operation research.

A researcher who resumes a project after some considerable time faces a similar problem. If he has not maintained a careful record, he probably has forgotten many application specific details. Consequently he will have to laboriously exhume his command sequences, parameter settings etc.

Another version of the same dilemma appears during the collaboration of two researchers. When the first researcher hands a project to his colleague he often has to explain tedious details on how to execute various programs to attain the desired result and which files contain the significant parts of his contribution to the project.

The situation is even worse when results in a field of scientific computations are published. Traditionally the audience is treated to a fundamental outline of the processing sequence, but is spared the complex details of data preparation, parameter values, source code, etc. While such a report informs the audience about the overall strategy used to attack the problem, it does not help the reader to leverage the work of the author: To check or refine the published work, a reader has to painfully repeat the author's implementation and testing sequence. If the originally used data is not available, or if the description of the overall strategy misses essential details, a reader may find it impossible to reproduce the given results at all. The value of reproducibility probably does not lie in detecting the rare cases of fraud, but more importantly in an enhanced efficiency of cooperation and technology transfer within the scientific community.

THE READER'S GAIN

Traditional documents in fields involving scientific computations contain only the advertisement of research, but not the research itself. The paper documents usually comprise the description of the research and do not contain the complex details needed to reproduce the published results. The files and directories containing these often voluminous information are usually unorganized and cannot be exploited without in-depth knowledge of the application files involved. The lack of reproducibility of computational results hinders the cooperation and communication of progress within research groups as well as the general scientific communities.

The missing link is a reader interface: a small set of defined commands which allows a reader to interact with the document without knowing its details. At SEP we implemented following set of commands, called *reproducible electronic document rules* or `red` rules, using GNU `make`:

- `make burn` removes all result files mentioned in the local makefile's result list. The author needs to supply this result list. However, the rule implementing the target `burn` is supplied by SEP's `red` rules.

- `make build` recomputes all result files by exercising the local rules about how each result file is built. The author needs to implement the rules for recomputing the result files. Again the target `build` is part of SEP's `red` rules.
- `make view` displays the result files to the reader. The rule implementing the target `build` is supplied by SEP's `red` rules.
- `make clean` removes all secondary files. In most cases the author does not have to specify this rule but uses a default `red` rule.

These commands are based on the definitions of result, source, and secondary files[2]:

- **Result files** are all files that the author uses to document his findings. At SEP, result files are usually figures in various graphic formats, such as postscript.
- **Source files** are the minimum set of files required to reproduce all the result files. This can comprise C or FORTRAN source code, shell scripts, parameter files, makefiles, or data input files.
- **Secondary files** are all intermediate[3] files which a reader creates when reproducing the result files from the source files.

Ultimately, a reader using these commands can dissect the document to inspect the result and source files. Furthermore, he can remove the result files and observe their recomputation from its source files. Having analyzed the executed make rules of result file's recomputation, a reader can usually identify the parameter that he wants to modify, the input data that he wants to exchange, or the source code that he wants to inspect.

The reader interface targets `burn, build,` and `view` are paired with targets that act on individual result files: For example `make myresult.burn` removes a result file *myresult*, but leaves all other result files intact.

Degree of reproducibility

GNU `makefiles` of SEP documents typically define three result list variables rather than a single one as implied in the paragraphs above. These three variables, any of which the author may omit if empty, are `RESULTSER`, `RESULTSCR`, and `RESULTSNR`. The endings `ER`, `CR`, and `NR` indicate to the reader the degree of reproducibility:

- **ER: easily-reproducible** describes result files which the reader can expect to regenerate within less than ten minutes on a standard workstation.

[2]The files related to the author's article (e.g. a TEX, device independent, or postscript format) do not entirely fit into the file classification above. Since the article in its final display format (e.g. postscript, or dvi) is often created from some source file (e.g. LATEXor nroff) they could be considered result and source file pair. However, the transformation of the text document into various formats is trivial: neither the author nor his reader have any scientific interest in it. Consequently we do not consider the article related files as part of the reproducibility scheme.

[3]GNU `make` defines *intermediate* slightly differently. See the section **Secondary files**.

- **NR: non-reproducible** are all the result files which a reader cannot reproduce, such as hand-drawn illustrations or scanned-in figures.[4]

- **CR: conditionally-reproducible** indicates results which require proprietary data, licensed software, or a lavish amount of time for recomputation. In all 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., `myresult.ps`) with a warning file (`myresult.warning`) in which he states why he classified the result as conditionally-reproducible. In industrial-scale geophysical research, many interesting results fall into this category.

The standard make targets `burn` and `build` are complemented by targets `burnER, burnCR, burnNR, burnall` and `buildER, buildCR, buildNR, buildall`. For example `burnNR` burns all non-reproducible result files (which is probably a very bad idea since they are non-reproducible). The target `burn` is defaulted to `burnER` to restrict the standard removal of result files to easily reproducible ones. The target `build` is defaulted to `buildER` to recompute all result files that `make burn` may have removed.

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.

Usage

When inspecting an 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`). Often we prefer to execute `make myresult.build` to learn about the specific result file `myresult`.

A concrete test of a document's reproducibility is a cycle of burning and rebuilding its results. Research stocked in such a document can be interrupted for an indetermined time to be later resumed by the author or a colleague. At SEP we often burn and rebuild entire articles, sponsor reports, and books to ensure their validity and portability. Since all authors collectively adhere to a single set of commands, a general *test suite* can verify the reproducibility of all completed documents. We use such a test suite of removal and recomputation to ensure the completeness of an author's document before publication. Additionally we benchmark new computer equipment by comparing the time to complete such a test suite. These benchmarks are especially valuable since they are based on a typical representation of our own work.

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

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

building them.[5] Beginning students often start their first project at SEP by exploring a senior colleague's results in this manner.

The reader interface for reproducible research is only one component of SEP's current computational research environment: A research document at SEP is written in LaTeX (Lamport, 1986). Using SEP's own LaTeXmacros, a push-button in each figure caption invokes a graphic user interface (written in a script language called xtpanel (Cole and Nichols, 1993)). The graphic user interface enables a reader to interactively execute the `burn, build, clean,` and `view` commands for each individual figure. SEP's GNU `make` rules allow the author to easily extend the interactivity of a result figure to additional, application-specific actions. Unfortunately these features are beyond the scope of this article. However we bundle our entire software and the theses of our research group in CDROMs which we distribute.

THE AUTHOR'S EFFORT

Having specified what rules we want to supply to every reader of a reproducible document (`burn, build, clean` and `view`), we discuss in this chapter what rules we expect from an author of an application makefile.

A primitive implementation could require each author to explicitly write a `burn, build, clean,` and `view` rule for each figure. But this would incur a massive overhead of duplicated code which an author needs to supply with each makefile. Instead our laboratory supplies the author a set of common include files, the `red` rules, that implement a standard `burn, build, clean,` and `view` rule. The entire GNU `make` code for SEP's implementation comprises only 150 lines. The author complements the `red` rules with the bare minimum of application–specific rule and variable definitions. The author:

- lists all result files in one of the result lists `RESULTSER, RESULTSCR, RESULTSNR`,
- writes a rule to build each file mentioned in the result lists,
- states a clean rule which removes all secondary files (which he can default),
- includes all the shared files containing the default red rule and variable definitions.

This list constitutes the author interface of SEP's reproducible electronic document.

We base our reader interface on `make` since it excels in the efficient recomputation of result files by a notion of a files dependencies and up-to-dateness. An author who already uses GNU `make` to store his processing commands can adopt SEP's reader interface by merely adhering to certain file naming conventions.

[5]A change in the source code or the default parameter values is where we draw the line between reproducibility and interactivity.

Electronic Publication

The tremendous increase in connectivity and information exchange (especially by the Internet and through CDROMs) enable us to complement most computational research publications by a corresponding complete set of application files. These files comprise input data, source code for processing programs, and parameters, and command scripts such as makefiles.

Even when such a reproducible electronic document and the accompanying `red` rules ensure effortless reproduction of the result files within the author's compute environment, a reader at a remote site may encounter difficulties since his compute environment may differ. However, there exist many de-facto standards within any scientific community. Many computer science publications do not shy away from accompanying there book with a CDROM containing software. And as more people attempt to share software the quicker the industry will define standards.

We cannot give a general recipe on how an author should deliver software to his audience: it will depend on the computer environment their community uses. Donoho and Buckheit (1996) deliver Matlab scripts and expect their audience to have access to Matlab. At SEP we publish our reproducible electronic documents on CDROMs: on such a CDROM we supply besides the author's application specific programs, our own seismic data processing software, and publically available `Freeware`. We expect our audience to have access to a UNIX system, including a C and FORTRAN compiler, GNU `make`, and X11 Window System. In the future, we hope that the World Wide Web and software such as Java tremendously ease the distribution and sharing of software.

But even if the author and reader do not share the same environment, the reader benefits from having access to the author's application specific files. He will be able to study the author's implementation and if sufficiently interested, adopt the author's work for his own compute environment.

EXAMPLE

The document you are reading is in its electronic version accompanied by a subdirectory called `frog`. The subdirectory `frog` contains a complete albeit small example of a reproducible electronic document: a finite–difference approximation of the 2-D surface waves caused by a frog hopping around a rectangular pond. The files `paper.latex` and `paper.ps` hold two formats of the accompanying article. The set of `RATFOR`[6] files implement a 2-D wave propagation code. The `Fig` directory contains the result file: a postscript figure of the pond after some wild hops by the frog.

The central makefile in the `frog` directory contains:

```
SEPINC = ../Rules
include ${SEPINC}/Doc.defs.top
```

[6] `RATFOR`is a preprocessor for FORTRAN that provides control flow constructs essentially identical to those in C. SEP distributes `RATFOR` on its World Wide Web server.

```
RESULTSER = frog dot

col = 0.,0.,0.-1.,1.,1.
${RESDIR}/frog.ps ${RESDIR}/frog.gif: frog.x
        frog.x                     > junk.pgm
        pgmtoppm ${col} junk.pgm > junk.ppm
        ppmtogif        junk.ppm > ${RESDIR}/frog.gif
        pnmtops         junk.pgm > ${RESDIR}/frog.ps

objs =  copy.o adjnull.o rand01.o wavecat.o \
        pressure.o velocity.o viscosity.o wavesteps.o
frog.x: ${objs}

dot.build ${RESDIR}/dot.txt : dot.x
        dot.x dummy > ${RESDIR}/dot.txt
dot.view: ${RESDIR}/dot.txt
        cat ${RESDIR}/dot.txt
dot.burn:
        rm ${RESDIR}/dot.txt

dot.x : ${objs}

clean: jclean

include ${SEPINC}/Doc.rules.red
include ${SEPINC}/Doc.rules.idoc
include ${SEPINC}/Prg.rules.std
```

What does the author of this particular reproducible document supply in his makefile? The include directives at the top and bottom of the application makefile pull-in the `red` rules, default variable and rule definitions that all application makefiles share. These shared definitions allow the author to exclusively write the application-specific rules and definitions seen above. At SEP additional include files contain compilation rules which in this example are listed explicitly below the comment at the center of the makefile.

The first half of the makefile are the rules needed to complement the included reproducibility rules. The `RESDIR` variable points to a directory where the author stores his result files. The `RESDIR` variable definition at the top of the makefile overwrites the included default value `./../Fig`. The next variable `RESULTSER` contains the list of all easily reproducible result files of a document. By default the variable `RESULTSER` is empty. In this case the only result figure is `frogpond`. The document does not contain any conditional or non-reproducible result files.

The next rule contains the commands to build the postscript version of the result file `frogpond`. Such a rule is application specific and cannot be supplied by included default rules. The target name comprises the directory `RESDIR` in which the results reside and a file suffix `.ps` which indicates the files postscript format. The rule depends on an executable `frogs.x` which it executes during the computations of the result.

Generally, default rules for compilation and linking of executables such as `frogs.x` are supplied by shared include files. To ensure portability I included extremely simple compilation and linking rules at the bottom of this makefile. However, the dependency of the executable on its subroutine object files, as in the case of `frogs.x`, needs to be defined by the author of the makefile, since it depends on the application's executables.

Finally, the target `clean` invokes the included `red` targets `jclean` and `texclean`. The targets identify secondary files based on certain naming conventions of our laboratory and remove them.

What does the reader interface offer a reader? In a standard UNIX environment, a reader of the electronic version of the document can interact with it using the default rules of the reader interface:

- `make burn` removes the result file `frogpond.ps` from the result directory `Fig`.
- `make build` recomputes the result file `frogpond.ps` by compiling the FORTRAN executable `frogs.x`, executing it, and converting a bitmap output of `frogs.x` to postscript using some publically available X11 routines.
- `make view` up-dates the result file `frogpond.ps` and displays it using the postscript viewer `ghostview` (if available on your system).
- `make clean` removes secondary files that were created during the execution of the build command It actually invokes a the included default targets `jclean` and `texclean`. The targets identify secondary files based on certain naming conventions of our laboratory and remove them. (For example object files with suffix `.o` or temporary files with the name stem `junk`).

A reader is able to execute these commands without understanding any application specific details. Furthermore, he can start changing any of the available files and observe the effect on the documents results.

Burn

Here is a template for the `burn` target which is included in every application makefile as part of the `red` rules:

```
burn: burnER

burnER: ${addsuffix .burn, ${RESULTSER}}
burnCR: ${addsuffix .burn, ${RESULTSCR}}

%.burn:
        this shell command removes the file designated by the stem ``%''
```

`burn` is what GNU `make` refers to as a *phony* target: the target does not relate to an actual file called `burn`. `burnER, burnCR` and `%.burn` are other examples for phony targets. The dependencies to `burnER` are generated by one of GNU `make`'s built-in functions `addsuffix`. The dependency list contains all files the author listed in his `RESULTSER` list. We use this trick to spawn off an up-date of a set of GNU `make` targets. In turn each individual target `myresult.burn` is up-dated by executing a command which removes the resultfile `myresult`.

At SEP result files are usually figures of various formats: often postscript and SEP's *vplot* format. An author at SEP does not actually list a particular file in the list of result files, but the stem of a result file. Usually an entire family of files with various suffixes relate to a single stem. As in the example above: `RESULTSER = frogpond` may refer to a file `frogpond.ps, frogpond.v, or frogpond.v3`. The burn command at SEP searches for each stem an entire list of possible suffixes (`RES_SUFFIXES = .v3 .v .ps`) and removes the existing files:

```
%.burn : FORCE
        ${foreach sfx, ${RES_SUFFIXES} , \
            if ${EXIST} ${RESDIR}/$*${sfx} ; then \
                 ${RM}  ${RESDIR}/$*${sfx} ;  fi; \
        }
```

This pattern rule is also invoked when a reader removes an individual result file `myresult.ps` by executing `make myresult.burn`. The standard `burn` rule exclusively removes the easily reproducible result files. A reader can remove the conditionally reproducible result files by explicitely invoking `make burnCR` for example.

Build

The template for the `build` rule is almost identical to the `burn` rule:

```
build:   buildER
buildER: ${addsuffix .build, ${RESULTSER}}
buildCR: ${addsuffix .build, ${RESULTSCR}}

%.build:
        this shell command invokes the author's rule for making the file
        designated by the stem ``%''
```

A very general way to utilize the `build` rule, is to require an author of an application file to submit a rule `myresult.build` for all files `myresult` listed in his makefile's result lists.

At SEP, every result file is a figure and is to exist in postscript format. The default `build` rule is, therefore, defaulted to:

```
%.build: ${RESDIR}/%.ps
```

Since many (but not all) figures at SEP are actually generated as vplot figures[7] it is convenient to supply an additional default rule which converts vplot files into postscript format:

[7] Vplot is SEP's prefered graphics format.

```
${RESDIR}/%.ps: ${RESDIR}/%.v
        pstexpen ${RESDIR}/$*.v ${RESDIR}/$*.ps
```

The author in the frog example does not use this default rule since his application rule directly creates the postscript file[8].

View

The `view` rule updates all result rules and displays them:

```
view  : ${addsuffix  .view, ${RESULTSALL}}

%.view: FORCE
        this shell command displays the contents of the file designated
        by the stem ``%''
```

`RESULTSALL` is a simple concatenation of `RESULTSNR RESULTSCR RESULTSER`.

At SEP the `%.view` rule checks for various result formats and chooses the first one the makefile knows how to generate (the return value of a recursive `gmake -n` call indicates which figure format can be built). Having found the figure's format, the makefile invokes a rule (`view3, view1, viewps`) whose dependency updates the figure and whose command body displays it using the appropiate viewer (SEP's `view` for vplot and GNU's `ghostview` for postscript files):

```
%.view: FORCE
        @\
        if   ${CANDO_V3} ; then         \
          ${MAKE} $*.view3 ;            \
        elif ${CANDO_V}  ; then         \
          ${MAKE} $*.view1 ;            \
        elif ${CANDO_PS} ; then         \
          ${MAKE} $*.viewps ;           \
        else                            \
          echo "make $@: cannot make targets $*.% (% = view3,view1,viewps)" ; \
        fi

%.view3 : ${RESDIR}/%.v3 FORCE
        view      ${RESDIR}/$*.v3

%.view1 : ${RESDIR}/%.v FORCE
        view      ${RESDIR}/$*.v

%.viewps : ${RESDIR}/%.ps FORCE
        ghostview ${RESDIR}/$*.ps
```

In the `frog` example the `make frogpond.view` rule does not find a rule for computing a `.v3` or `.v` vplot version of frogpond. Consequently, it invokes the postscript

[8]We will say more about precedence of competing rule and variable definitions later: however we ensured that the author's definitions always prevail over possible default ones.

alternative `frogpond.viewps` of the command body. In return `frogpond.viewps` executes `ghostview` to display the result file `frogpond.ps`. If your computer system does not support the postscript viewer `ghostview`, then you will need to supplement the `%.viewps` rule with your own display command.

Clean

The clean rule removes all secondary files. In general, the author of an application makefile needs to supply his individual clean rule since it is not possible to foresee what secondary files the author may choose to create when computing his result files. At SEP we, however, supply an author with a default clean rule called `jclean`: `jclean` identifies secondary files based on certain naming conventions. For example any file with the stem `junk` or with the suffix `.o` is a secondary file and is removed by the `jclean` rule:

```
jclean : klean.usual klean.fort ;

KLEANUSUAL := core a.out mon.out paper.log \
              *.o *.x *.H *.A *.M *.ps *.v *.v3 *.V *.trace
klean.usual :
        @-${TOUCH} ${KLEANUSUAL} junk.quiet
        @-${RM}    ${KLEANUSUAL} junk.*

FORT_FILES = $(patsubst %.f,%,$(wildcard *.f)) junk.quiet
klean.fort:
        @\
        for name in ${FORT_FILES} ; do                          \
          if  ${EXIST} $${name}.r                               \
              ${TOUCH} $${name}.f ;                             \
              ${RM}    $${name}.f ;                             \
          fi ;                                                  \
        done
```

If not applied carefully, the clean rule can remove valuable source or result files. SEP's `jclean` definition removes all files with suffix `.v, .v3, .ps` which indicate a file containing a figure. Since all result files at SEP are figure files and have one of these suffixes, the `jclean` command should never be invoked in a directory containing SEP result files (the `RESDIR` directory). By not supplying a default `clean` rule, the author of a makefile has to consciously choose his cleaning mechanism. He should only use the supplied `jclean` rule after he studied which files it will remove.

In the accompanying `frog` example, `make build` creates several secondary files when recomputing the result file `frogpond.ps`. The rule that generates the postscript figure stores bitmap versions of it in `junk.pgm` and `junk.ppm`. Furthermore the compilation rule for `frog.x` creates a set of object files and the executable itself. All those files are removed by `make clean`, since they all adhere to certain naming conventions: the `jclean` target removes all files with suffix `.o` (object files), the stem `junk` (in this case temporary files), and files with suffix `.x` which at SEP indicates an executable. The rule `klean.fort` removes the FORTRAN file with suffix `.f` if a

RATFORversion (suffix .r) of the file exists. In the near future, we hope to replace the formulation of the clean rule based on naming conventions with a formulation based on GNU make's notion of secondary files.

Secondary targets

The importance of the clean rule is often underestimated. The reproducibility of an electronic documents requires the source files. The result files illustrate the document's contents. But the secondary files are used exclusively in the process of generating the result files from the source files. Since secondary files can be prohibitively large, and generally clutter the collection of document files, they are typically removed soon after they perform their function of helping to create result files.

The reader uses the clean rule to identify the documents important files: the files which contain the author's intellectual achievement, his research. Additionally, the clean rule enables the author to isolate the minimum set of files which he exchanges with his reader, saving bandwidth in the exchange of the electronic document.

In general `make` treats secondary files in a manner unsatisfactory to us. In some cases[9], `make` considers a result file out-of-date when a secondary file that it depends on does not exist. In these cases, it insist on the recomputation of the result file even when all existing source files are older than the result file. Consequently a document needs to retain secondary files to avoid redundant recomputations of the result files.

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 intermediate 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 in every application makefile causing GNU `make` to assume any missing secondary files to be up-to-date.

GUIDELINES FOR SHARED RULES

The previous section illustrated the implementation of the fundamental `red` rules. This section discusses three principles that guided our implementation of the `burn, build, view` and `clean` rule.

- **No code duplication**: all common rule and variable definitions are located in shared include files.
- **The author rules**: any definition by the author takes precedence over the included default definitions.
- **Rules are grouped by object**: an author can selectively include default definitions according to his needs.

[9]For GNU `make` this refers to non-pattern rules.

No code duplication

Instead of rewriting a `burn, build, clean` and `view` rule for each makefile, a set of common files is included in each individual application makefile.

```
include ${SEPINC}/SEP.top
```

includes a file `SEP.top`. It finds the `SEP.top` file by expanding the variable `SEPINC` which at SEP is defined as an environement variable pointing to the directory which contains all shared include files. `SEP.top` itself is actually a file which includes a set of other shared resource files. The code implementing the `burn, build, clean` and `view` rules resides in such a shared resource file.

Having a single set of shared source files ensures consistency of the reader command interface accross different documents. Furthermore changes to the common source files propagate immediately to all application makefiles. Consequently an author of an application makefiles supplies the bare minimum of definitions and rules: definitions and rules that are specific to his application.

The author rules

When designing the SEP make rules it was our goal to offer the author of an reproducible electronic document a maximum of helpful default rules and variable definitions without limiting him in his freedom. Consequently we arranged the included files so that any variable or rule definition by the author of an application makefile takes precedence over included competing default definitions. Unfortunately, we have not been able to devise a simple and reliable mechanism to warn an author when he accidently overwrites a default rule or variable.

0.0.1 Precedence of variable definitions

GNU `make`'s precedence rules for variable definitions are rather arcane. GNU `make` expands some variables at the time a makefile is read and some at the time a certain rule is executed. When first reading the makefile, GNU `make` expands all variables

- on the right-hand side of ':='
- on the target-dependency line of a rule (top line of a rule)
- in the conditional expressions of `define` statements

Other variables, such as

- on the right-hand side of '=' (exclusively used in SEP's include files)
- in the shell commands of a rule (indented)
- in the body of `define` statements

are expanded when GNU `make` executes a rule.

For example following makefile

```
foo = hello

${foo} :
        @echo ``target is $@'';
        @echo ``foo is ${foo}'';

foo = goodbye
```

leads to following result:

```
> make hello
target is hello
foo is goodbye
> make goodbye
gmake *** No rule to make target `goodbye'. Stop.
```

The bottom definition `foo = goodbye` affected the commands of the rule above but not the target-dependency line. GNU `make` expanded the variable `foo` in the target-dependency line when it first read the makefile from top to bottom. However it expanded the `foo` variable in the command body after it had read the entire makefile, had redefined `foo` according to the second definition, and was executing the rule.

Consequently, an author of an application makefile has to include all default variable definitions at the top of his makefile, above his own, potentially competing definitions. Thus his own variable definitions will be the bottom definitions used by the makefile when executing a rule.

Additionally, all default rules have to be included at the bottom of the makefile, so that the author's variable definitions are read by the makefile before expanding variables in the target-dependency line of these rules. For example in the `frog` makefile above, the makefile redefines the included variable `RESDIR = ../Fig` to `RESDIR = ./Fig`. This redefinition occurs below the included default variable definition in `Doc.defs.top`, but above the included rules in `Doc.rules.fig` such as `${RESDIR}/%.ps : ${RESDIR}/%.v`.

An exception are *conditional variable definitions* such as:

```
ifndef COLOR
  COLOR = n
  FAT = 1
  FATMULT = 1.5
  INVRAS = n
endif
```

Since conditional define statements are evaluated as the makefile is read, conditional definitions have to be included at the bottom of an author's application makefile. Thus the author has a chance to redefine the variable `COLOR` before GNU `make` encounters the conditional `ifndef` test. Conditional variable definitions are convenient when offering two or more default settings for a group of variables such as in the case of a color or grey-scale plot in the example above. In the `frog` example all default conditional definitions are included in `Doc.defs.bottom` below the author's definitions.

Since an author of an application is unaware of all variables defined in the included default files, the GNU `make` manual suggests that an author should use only lower case letters for local variable names in his application makefiles. Since the officially included files use strictly upper case variable names, an author using lower case letters cannot accidently overwrite any official definition. The author should reserve upper case variables to reference or deliberately override included standard variables (e.g. `RESDIR` in the `frog` example).

0.0.2 Precedence of rule definitions

Having ensured that an author can always overwrite any default variable, we also have to guarantee that his rules take precedence over included default rules. When a target corresponds to several explicit rules, only one rule can list a body of corresponding shell commands: all other rules specify additional dependencies. If more then one rule contains a body of shell commands GNU `make` fails when trying to up-date that target. If a target corresponds to an explicit rules and an implicit rule, the explicit rule takes precedence. If several implicit rules correspond to a target, the rule listed first in the makefile takes precedence.

Except the basic reader targets `burn, build, view,` and `clean`, all included default rules are implicit rules or targets without a command body. All default rules are included at the bottom of the application makefile. An author's explicit rules take precedence over all included implicit rules. An author's implicit rules take precedence since they are listed in the makefile above the default rules included at the bottom. However, if an author attempts to overwrite one of the very few included explicit rules, `burn, build, view` and `clean`, the author's and the default rules will collide and up-dating the target will fail.

For example, the rule for a postscript figure created by a mathematica script (its explicit or implicit version) overwrites the default rule for postscript figures:

```
# author's EXPLICIT rule overwrites his own included IMPLICIT rule (next)
# as well as the IMPLICIT default rule (last rule)
${RESDIR}/math.ps : math.ma
        < math.ma  mathematica > ${RESDIR}/math.ps

# author's IMPLICIT rule overwrites the included IMPLICIT rule stated BELOW
${RESDIR}/%.ps : %.ma
        < $*.ma  mathematica > ${RESDIR}/$*.ps

#included default rule (is overwritten by both rules above)
${RESDIR}/%.ps: ${RESDIR}/%.v
        pstexpen ${RESDIR}/%.v ${RESDIR}/%.ps
```

In summary, all default rules and variables of the `red` rules except `burn, build, view` and `clean` can be overwritten by the author of a SEP makefile as long as he enters all his makefile entries between the inclusion statements at the top and bottom.

```
include ${SEPINC}/SEP.top

# Here is where the application specific definitions belong.

include ${SEPINC}/SEP.bottom
```

The files `SEP.top` file contains a set of include directives:

```
default :

${SEPINC}/SEP.top : ;

.SUFFIXES:                              # delete all implicit rules
.SUFFIXES: .out .a .F .e .y .ye .yr .l .s .S .info .dvi .tex .texinfo \
#               .cweb .web .sh .elc .el

# suppress the Entering/Exiting directory messages
MAKEFLAGS += --no-print-directory

include ${SEPINC}/Prg.defs.top
include ${SEPINC}/Doc.defs.top
include ${SEPINC}/Doc.rules.idoc
```

By the way the line `${SEPINC}/SEP.top :  ;` prevents GNU `make`from attempting to up-date the file `SEP.top` itself and thereby speeds up GNU `make`'s execution.

The `SEP.bottom` file includes all the shared files which need to be placed below the author's definitions:

```
${SEPINC}/SEP.bottom : ;

include ${SEPINC}/Prg.rules.obj
include ${SEPINC}/Prg.rules.exe
include ${SEPINC}/Doc.rules.test
include ${SEPINC}/Doc.rules.fig
include ${SEPINC}/Doc.rules.action
include ${SEPINC}/Doc.rules.clean
include ${SEPINC}/Dir.defs.bottom
include ${SEPINC}/Prg.defs.bottom
include ${SEPINC}/Doc.defs.bottom
```

Rules are grouped by object

While the names of `SEP.top` and `SEP.bottom` indicate where the files have to be included in an application makefile (see last section on precedence of definitions), the names of all other include files indicate their contents: The first part of the name indicates if the file is concerned with compilation of executables (prefix `Prg` for *program*) or with documents (prefix `Doc`). The middle part of each name indicates if the file contains variable definitions (`defs`) or rules (`rules`). Finally, the suffix of a file name states what the rules are concerned about respectively, where the variable definitions are to be included in a makefile. For a comprehensive list of all GNU `make` include files at SEP see Appendix A.

The files containing definitions concerning reproducibility are:

- `Doc.rules.fig` contains the `burn`, `build` and `view` rules
- `Doc.rules.clean` contains the `clean` rule
- `Doc.defs.top` contains the `simple` variable definitions
- `Doc.defs.bottom` contains all `conditional` variable definitions

The contents of each file is carefully commented. The contents of the files concerning reproducibility are included in Appendix B.

By separating code by functionality, we hope to maintain smaller, encapsulated source files. Authors can plug and play with the different rule sets of their interest without being burdened with unnecessary rule or variable definitions.

CONCLUSION

We believe the set of reader commands (`burn, build, clean, view`) presented in this article, combined with a carefully prepared package of an author's application files, facilitates a reader with a reproducible electronic document of a scientific computational research project.

Traditional research documents in computational sciences only contained *advertisement* for the researcher's work, not the actual research. Because of the complexity of computational research, the research described in paper documents usually lack the details necessary for its result's exact reproduction. But it is the reproducibility which in Newton's words enables a researcher *to stand on the shoulders* of his colleagues.

SEP offers 150 lines of GNU `make` code that implement a reader interface. The simple commands, `make burn, build, view,` and `clean`, enable a reader who has access to a copy of the author's application files, to remove and reproduce an author's results without knowing any application specific details (such as parameter settings or names of executables). The process of recomputing the author's results allows a reader to understand and if he wishes to modify the interaction of the various components.

We chose to implement the reader interface in GNU `make` which excels in the efficient maintenance of even complex software packages. An author who already stores his commands in `make` only needs to adhere to certain naming conventions and include the rules which SEP distributes on his world wide web site.

Conceptually the reader interface presented in this paper is independent of the document format (TEX, `html`, etc) and independent of the underlying *computational machinery*, such as Matlab, Mathematica, or C and FORTRAN programs. Even this paper restricts itself to UNIX systems, the concept of a reader's interface to reproduce a documents contents should apply to electronic documents on other operating systems as well.

The reader interface has a taste of an electronic filing system: the research software maintained by each SEP researcher and accessible to any colleague has increased tremendously. Furthermore, SEP students commonly take up projects of former students, starting by easily removing and recomputing the original result files. Students, who graduated and left SEP, report little trouble to seamlessly continue their own research at their new location.

We successfully equipped three of Jon Claerbout's books and SEP's most recent sponsor report with GNU `makefiles`. These documents contain a total of about 483 result files (276 easily reproducible, 21 conditionally reproducible, and 186 nonreproducible figures). Before publication all 276 easily reproducible result files mentioned above have been tested by frequent burning and rebuilding on different platforms. Additionally SEP published 12 PhD theses that use a earlier version of the reader interface in a `make` dialect called `cake`. All these electronic documents are available on CDROMs (Claerbout, 1996a).

What is next? We, of course, want to publish our results on the world wide web. The web distributes conveniently the package of human reading material and computational machinery. We are carefully following the development of Java(1996). However, each figure in a world wide web document would surely be accompanied by a *push-button* for the `burn, build, clean`, and `view` command.

We believe that such rules are useful to most researchers who conduct scientific computations. We distribute our own rules, this article, and the article's `frog` example on the world wide web (Claerbout, 1996b).

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`. Joel Schroeder conceived the three result lists and understood precedence of GNU `make` definitions. He refined the first rough translation of SEP's cake rules significantly. Martin Karrenbach implemented a TEX macro which extracts the reproducibility information `(ER,CR,NR)` from the makefile and indicates it in each figure caption of a LATEXdocument. Steve Cole and Dave Nichols implemented a mechanism variable in their `xtpanel.builder` which made it a breeze to use GNU `make` in SEP's standard interactive graphical figure interface. We thank Mihai Popovici for his courage in testing an early version of SEP's GNU `make` 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 their original cake rules.

REFERENCES

Buckheit, J., and Donoho, D., 1996, Wavelab and reproducible research: http://playfair.Stanford.EDU:80/ wavelab/.

Claerbout, J. F., 1996a, CDROMs of the Stanford Exploration Project: http://sepwww.stanford.edu/office/sepcd.html/.

Claerbout, J. F., 1996b, Home page of the Stanford Exploration Project: http://sepwww.stanford.edu/.

Cole, S., and Nichols, D., 1993, Xtpanel update: Interactivity from within existing batch programs: SEP–**77**, 409–416.

Lamport, L., 1986, Latex: A document preparation system: Addison-Wesley Publishing Company.

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

Oram, A., and Talbott, S., 1991, Managing projects with make: O'Reilly & Associates, Inc.

Somogyi, Z., 1984, Cake: a fifth generation version of make: http://munkora.cs.mu.oZ.au/ zs/.

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

SUN, 1996, Java: Programming for the internet: http://java.sun.com/.

APPENDIX A: FILE DESCRIPTION

APPENDIX B: GNU MAKE FILES