MAKING SCIENTIFIC COMPUTATIONS REPRODUCIBLE

To verify a research paper's computational results, readers typically have to recreate them from scratch. ReDoc is a simple software filing system for authors that lets readers easily reproduce computational results using standardized rules and commands.

In the mid-1980s, we realized that our laboratory's researchers often had difficulty reproducing their own computations without considerable agony. We also noticed that junior students, who typically build on the work of more advanced students, frequently spent a great deal of time and effort just to reproduce their colleagues' computational results.

Reproducing computational research poses challenges in many environments. Indeed, the problem occurs wherever people use the traditional methods of scientific publication to describe computational research. For example, in a traditional article, the author simply outlines the relevant computations—the limitations of a paper medium prohibit complete documentation, which would ideally include experimental data, parameter values, and the author's programs. Readers who wish to use and verify the work must reimplement it, which is often a painful process. Even if readers have access to the author's source files (a feasible assumption given recent progress in electronic publishing) they can only recompute the results by invoking the various programs exactly as the author invoked them; such information is something that is usually undocumented and difficult to reconstruct.

To address these problems, we developed ReDoc, a system for reproducing scientific computations in electronic documents. Since implementing it in the early 1990s, ReDoc has become our principal means for organizing and transferring our laboratory's scientific computational research.

ReDocs are best defined operationally: After an author completes a ReDoc, readers can destroy all existing results—principally the illustrations—and rebuild them using the author's underlying programs and raw data. Using ReDoc, authors describe their computations and preserve their details in fully functional examples. Authors can also test their archived research software by occasionally removing and regenerating the document's results using ReDoc's standardized interface commands. ReDoc also lets authors develop automatic scripts to verify any document's completeness and reproducibility before its publication. Scientific journal publishers might also use ReDoc in the referee process to test the reproducibility of illustrations.

ReDoc benefits readers in several ways. Just as a driver wants to find the brake pedal in the

1521-9615/00/$10.00 © 2000 IEEE

MATTHIAS SCHWAB, MARTIN KARRENBACH, AND JON CLAERBOUT
Stanford University

same location in every car, readers benefit by having a few standard commands that let them explore any document's scientific contents. Consistent, standard commands to remove and reproduce a document's result files help readers access and study an unknown document.

ReDoc overview

In essence, ReDoc is a filing system that makes research cooperation effortless: Researchers can publish their results and colleagues can immediately build upon them, adding improvements and innovations. To create a ReDoc, authors combine project-specific rules with general, standard rules and naming conventions to offer readers a simple set of commands to remove and rebuild the author's results. ReDocs become reservoirs of easily maintained, reusable software, and thus authors, along with readers, benefit from using the system.

In our laboratory, using ReDoc has tremendously increased the amount of software that is readily accessible to researchers. Students can now take up former students' projects and easily remove and recompute the original result files. Students who graduate and leave our laboratory can seamlessly continue their research at new locations. Although the effort to generate reproducible research documents is typically directed at helping other people use existing work to further their own, authors also benefit. Because many researchers typically forget details of their own work, they are not unlike strangers when returning to projects after time away. Thus, efforts to communicate your work to strangers can actually help you communicate with yourself over time.

ReDoc's current reader interface is implemented in the platform-independent GNU `make`, which excels in efficiently maintaining even complex software packages. Conceptually, the ReDoc reader interface is independent of both document format (TeX, HTML, and so on) and the underlying computational software, whether it be Matlab or Mathematica, or C and Fortran programs. Although we restrict our focus here to Unix systems and the `make` utility, the concept—using a reader interface to reproduce a document's computational results—should apply to electronic documents in other computer environments as well. The GNU ↑ code that implements the ReDoc rules, this article, and the accompanying example is freely available on our Web site at http://sepwww.stanford.edu/research/redoc.

ReDoc features

In addition to a traditional article and the application's source code, a ReDoc features three main components: makefiles, a small set of universal `make` rules (less than 100 lines), and naming conventions for files. ReDoc offers a reader four standard commands:

- `burn` the figures,
- `build` the figures,
- `view` the figures, and
- `clean` up and remove all intermediate results.

These four commands are centered around three types of files:

- *Fundamental files* are data sets, programs, scripts, parameter files, and makefiles (any file not generated by some computer process).
- *Result files* are typically plot files, which can come in various formats including postscript or gif files.
- *Intermediate files* are the files that a computer process generates when computing the result files from the corresponding fundamental files. Examples include object files, executables, and partially processed data.

ReDoc's naming conventions let a community formulate universal rules that recognize a file's type and thus permit the system to handle it appropriately. For example, a cleaning rule uses a community naming convention to identify and remove all intermediate files, such as those with suffix `.O` (object files) or files with the name stem `junk` (temporary files). The `clean` rule leaves a tidy directory without any distracting clutter. We also use naming conventions for result files. We base our rules for displaying, removing, or recomputing a result file on the result file's name. For example, our result files are always figure files and can have various formats, such as PostScript or GIF. Our naming conventions require that authors indicate the result file's format by a suffix, such as .ps or .gif. Consequently, we can supply a universal format-independent rule for displaying result files: The rule identifies the result file's suffix, concludes the file's format, and invokes the appropriate viewing program, such as Ghostview for PostScript or Xview for GIF files.

ReDoc rules are easy to implement. As the example in the box, "Creating a ReDoc" explains, authors who already use makefiles need only adhere to the ReDoc naming conventions and include the ReDoc rules to make a traditional doc-

Creating a ReDoc

Because an author's rules deal with a document's application, we can best illustrate ReDoc creation using an actual example. The electronic version of this article is accompanied by a subdirectory called Frog. Frog contains a complete, albeit small, ReDoc that discusses a finite-difference approximation of the 2D surface waves caused by a frog hopping around a rectangular pond. The files paper.latex and paper.ps contain two formats of the short scientific article describing the finite-difference approximation. Some Ratfor files implement the 2D wave propagation code (Ratfor is a Fortran preprocessor that provides control flow constructs similar to C; Ratfor is freely available at our Web site). The `Fig` directory contains the result files: a figure (PostScript and GIF versions) of the pond after some wild frog hops, and the output (two float numbers) of a dot-product test of the linear finite-difference operator and its adjoint. To organize the document's files, the Frog example's author wrote the makefile shown in Figure A.

The variable `RESULTSER` contains the list of the document's easily reproducible results, `frog`, and `dot`.

The next rule contains the commands to build the PostScript and GIF versions of the frog result. Such a rule is application-specific and cannot be supplied by existing default rules. The target names comprise the directory `RESDIR`, which contains the result files and file suffixes (.ps and .gif), which indicate the file formats. The rule depends on an executable `frog.x`, which it executes during result computations.

A shared include file, Prg.rules.std, supplies default rules for compilation and linking of executables such as `frog.x`. Compilation and link rules are compiler-dependent. In this example, we include some generic Fortran rules. At our laboratory, compilation rules depend on an environment variable that indicates compiler type. The author must define the executable's dependency on its subroutine object files (as in the case of `frog.x`) because it depends on the application-specific file names.

In the case of the `dot` result file, the author-supplied rules reflect the reader-interface commands: `dot.build` creates the result file, `dot.view` displays it, and `dot.burn` removes it.

Finally, the target `clean` invokes the included default target `jclean`, which removes intermediate files based on our laboratory's naming conventions.

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

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
```

Figure A. The makefile that organizes the Frog files.

ument reproducible. We now describe ReDoc makefiles and rules in more detail.

Makefile

A makefile is a standard Unix utility for software maintenance and contains the commands to build a software package. More powerful than a simple command script, a makefile indicates that result files (targets) are up-to-date when they are younger than their corresponding source files (dependencies). (For more information, see the "clean, up-to-date support" sidebar.) Because maintaining a reproducible research project resembles maintaining software, the `make` utility solves our problem elegantly. Fortunately, fine tutorial books exist on the `make` language,[1,2] and students can easily begin using `make` without a formal introduction.

Because our laboratory deals with computational problems of various sizes that use a diverse collection of software and hardware tools, not every reader will be able to easily reproduce all result files. Consequently, authors typically define their application makefiles as one of three result list variables—`RESULTSER, RESULTSCR,` and `RESULTSNR`—with the ending letters indicating the degree of reproducibility:

- `ER`: Easily reproducible result files that can be regenerated within 10 minutes on a stan-

```
burn: burnER

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

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

Figure 1. `make burn` **finds and removes all easily reproducible result files.**

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

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

Figure 2. Implemented much like the `burn` **rule, the** `build` **rule updates easily reproducible result files.**

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

%.view:  FORCE
           if  ${CANDO_GIF}       ; then \
               ${MAKE} $*.viewgif ;      \
           elif ${CANDO_PS}       ; then \
               ${MAKE} $*.viewps  ;      \
         else                            \
           echo "can't make $*.viewps $*viewgif";\
         fi
```

(a)

```
%.viewgif : ${RESDIR}/%.gif FORCE
      ${XVIEW} ${UXVIEWFLAGS} ${RESDIR}/$*.gif

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

(b)

Figure 3. The `view` **rule (a) updates and displays results. When the** `%.view` **rule finds a workable version, it invokes** `%.viewgif` **or** `%.viewps`, **which updates and displays the result file (b).**

dard workstation.

- `NR`: Nonreproducible result files that cannot be recalculated (examples include hand-drawn illustrations or scanned figures).
- `CR`: Conditionally reproducible result files that require proprietary data, licensed software, or more than 10 minutes for recomputation. The author nevertheless supplies a complete set of source files; this ensures that readers can reproduce the results if they possess the necessary resources. Authors list the required resources in a warning file (such as `myresult.warning`), which accompanies the CR result file (`myresult.ps`). In industrial-scale geophysical research, many interesting results are conditionally reproducible.

Following this usage, the standard `make` targets `burn` and `build`, which we describe below, are complemented by targets `burnER`, `burnCR`, `burnNR`, and `buildER`, `buildCR`, `buildNR`. For example, `burnCR` burns all conditionally reproducible result files. The target `burn` defaults to `burnER` to restrict the standard removal of result files to easily reproducible ones. The target `build` defaults to `buildER` to recompute the result files that `make burn` removes.

Make rules

ReDoc's third component is a small, consistently available set of standard `make` rules. These rules let readers interact with the document without knowing the underlying application-specific commands or files. In the Frog example (see the box, "Creating a ReDoc"), a reader invokes targets, such as `build`, that are not listed in the author's application makefile. Standard laboratory rules supply these targets, which the researcher includes in the document's makefile (`Doc.defs.top`, `Doc.rules.red`, and `Doc.rules.idoc`).

Standard rules ensure a consistent reader interface and free authors from the need to reimplement the ReDoc rules in every makefile, letting them concentrate exclusively on the makefile's application-specific aspects. Although we formulated the rules so that individual authors can override them, in our experience this is rarely necessary or desirable. Furthermore, a central rules set accumulates the community's wisdom on how to organize a ReDoc.

Our laboratory offers about 100 lines of GNU `make` code that constitutes our ReDoc rules. The ReDoc rules facilitate four commands: `make burn`, which removes the result files (usually figures); `make build`, which recomputes them; `make view`, which displays the figures; and `make clean`, which removes all files other than source or result files. Although authors and readers do not need to understand the implementation details of the ReDoc rules to use them (most researchers at our laboratory have never inspected them), we describe the rules in detail here so that you can understand their operation and adapt them to your own computational environment.

Burn. As Figure 1 shows, `make burn` invokes a chain of rules, which ultimately finds and removes all easily reproducible result files. Every application makefile contains the `burn` target as part of the ReDoc rules.

The `burn` target invokes its dependency `burnER`. The `burnER` rule selects the easily reproducible result files for removal. The `burnER` rule uses GNU `make`'s built-in function `addsuffix` to generate its dependency list. Each entry of `burnER`'s dependency list is a concatenation of the name of an easily reproducible file and the suffix `.burn`. In the Frog example, `burnER` depends on `frog.burn` and `dot.burn`. The dependency `frog.burn` invokes the pattern rule `%.burn`, which removes the result files corresponding to the result `frog.` At our laboratory, a single result name such as `frog` usually denotes several result files of identical contents but differing format, such as PostScript or GIF. The `%.burn` rule scans a list of possible suffixes (`RES_SUFFIXES = .ps .gif`) and removes all related result files: `frog.ps` and `frog.gif`.

Because text result files, such as `dot.txt`, are rare at our laboratory, our ReDoc rules do not contain laboratory-wide rules for handling them. Given this, the Frog document's author supplies an explicit `dot.burn` rule in the makefile. This explicit `dot.burn` rule overrides the default `%.burn` pattern rule, which generates PostScript result files.

The standard `burn` rule exclusively removes the easily reproducible result files. Readers can remove conditionally reproducible result files by invoking `make burnCR` and can exclusively remove the result files related to the `frog` result by invoking `make frog.burn`.

Build. Figure 2 shows the `build` rule, which updates the document's easily reproducible result files. The `build` rule's implementation resembles the `burn` rule's implementation.

At our laboratory, almost every result file is a figure in PostScript format. Consequently, the ReDoc `%.build` rule updates the PostScript version of any easily reproducible result, such as `${RESDIR}/frog.ps`. ReDoc typically generates additional versions of the result (such as frog.gif) as a side effect of the rule that computes the PostScript version (frog.ps).

As in the case of `dot.burn`, authors must supply an explicit `dot.build` rule for the nonstandard text result `dot.`

View. As Figure 3a shows, the `view` rule updates and displays the results.

`RESULTSALL` lists all result files; we define it as the concatenation of `RESULTSALL`, `RESULTSCR`, and `RESULTSER`.

At our laboratory the `%.view` rule checks for the various formats of a result and chooses the first version that the makefile can generate. The variable `CANDO` contains the return value of a recursive `gmake -n` call. This return value indicates whether the system can build that particular version of the result. As Figure 3b shows, once it finds a workable version, the `%.view` rule invokes another rule (`%.viewgif` or `%.viewps`) that updates and displays the result file.

In the Frog example, the `frog.view` rule finds a rule for computing a .gif version of `frog.` It then invokes the GIF rule `frog.viewgif`. In return, `frog.viewgif` executes Xview, a .gif viewer, to display the result file `frog.gif`. If

Clean, up-to-date support

Keeping documents clean and up to date has obvious benefits in terms of order and reader confidence. Furthermore, we found that only clean documents function on a CD-ROM; intermediate files stored in a CD-ROM's read-only memory cannot be overwritten when readers later regenerate the files. Unfortunately, in early versions of ReDoc, we found that popular `make` dialects did not support rules to keep documents simultaneously clean and up-to-date. Such dialects considered a result file outdated when certain intermediate files were missing (for example, GNU `make` considered a result file outdated when a nonpattern rule formulated a missing file's dependency). Treating intermediate files this way was convenient for software maintenance, but unsuitable for reproducible electronic documents.

Today, ReDoc rules maintain a clean, up-to-date document in the GNU `make` dialect. At our request, Richard Stallman enhanced GNU `make` to adequately handle the ReDoc rules' intermediate targets (GNU `make` refers to intermediate files as secondary files, as it uses "intermediate" in a slightly different, more restrictive sense). Stallman added a special built-in target, `.SECONDARY`, that lets authors choose GNU `make`'s behavior with respect to its missing intermediate files. If a makefile includes a `.SECONDARY` target without dependencies (the default at our laboratory), every missing intermediate file is presumed up to date. We implemented the `.SECONDARY` target in GNU `make` versions higher than 3.74. The current GNU `make` version we use is available on our Web site, http://sepwww.stanford.edu/research/redoc.

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

KLEANUSUAL := core a.out paper.log *.o *.x *.H *.ps *.gif
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} $${name}.r ; then      \
                ${TOUCH} $${name}.f ;           \
                ${RM}  $${name}.f ;             \
            fi ; \
         done
```

Figure 4. The default `jclean` mechanism removes intermediate files from the Frog example.

your computer system does not support XView, you must supplement the `%.viewgif` rule with your own display command. To display the result file `frog.ps`, `frog.viewps` executes Ghostview.

Clean. A cleaning rule is important because a cleaned directory is more accessible and inviting to readers than a cluttered one. Furthermore, a cleaning rule saves resources, such as disk memory, by removing superfluous files.

In theory, a community's cleaning rule would remove the intermediate files and thereby isolate the source and result files; a universal cleaning rule would recognize the intermediate files according to the community's naming conventions. Unfortunately, such a rule cannot possibly anticipate all the names authors might choose for intermediate files. Consequently, our laboratory does not supply a fixed, universal `clean` rule, but rather a Jon's clean (`jclean`) rule. `jclean` removes the files that adhere to our laboratory's naming convention for intermediate files. Authors are responsible for implementing their own `clean` rules, and most accept the default cleaning rule by defining `clean` as `clean: jclean`.

Some of our authors append the default `jclean` with a command to remove additional files that do not adhere to the naming conventions. Very few authors ignore the `jclean` target (and its communal wisdom) and design their own rule.

Because the author of the Frog example adheres strictly to the ReDoc file naming conventions, the default `jclean` mechanism suffices to remove the intermediate files, as Figure 4 shows.

The `jclean` target uses two methods to identify intermediate files. The first method, `klean.usual`, simply removes files whose names fit one of the rule's name patterns, such as the executable `frog.x`, or the intermediate bitmap files junk.pgm and junk.ppm. The second method, `klean.fort`, removes Fortran files, such as `frog.f`, if Ratfor versions of the program, such as `frog.r`, exist.

We have successfully used the ReDoc rules in our laboratory's most recent sponsor report (14 articles by 15 authors) and three of Jon Claerbout's textbooks on seismic imaging. These documents contain a total of 483 result files: 276 easily reproducible, 21 conditionally reproducible, and 186 nonreproducible figures. Before publication, automatic scripts removed and rebuilt all 276 easily reproducible result files (see Figure 5). We use the same scripts and documents to benchmark computer platforms. Our laboratory has also published 12 PhD theses using an earlier version of the user interface based on `cake`.[3] The theses are available on CD-ROM.

Researchers in our laboratory greatly enjoy and benefit from using ReDoc, and we continue to employ reproducibility to check the quality of all our laboratory's publications. Although we believe that reproducible documents represent promising opportunities for Web publications, we are not currently pursuing concrete solutions to this end.

Acknowledgments

We appreciate Richard Stallman's 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. Dave Nichols discovered `cake` and taught our laboratory how to use it. Steve Cole and Dave Nichols wrote xtpanel (http://sepwww.stanford.edu/ oldsep/dave/xtpanel) and helped wrap our tools into an interactive, electronic book.

References

1. M. Stallman and R. McGrath, *GNU Make*, Free Software Foundation, Boston, 1991.
2. A. Oram and S. Talbott, *Managing Projects with Make*, O'Reilly & Associates, Sebastopol, Calif., 1991.
3. Z. Somogyi, "Cake: A Fifth Generation Version of Make," *Australian Unix System User Group Newsletter,* Vol. 7, No. 6, April 1987, pp. 22–31; http://www.cs.mu.oz.au/~zs/ papers/papers.html (current Oct. 2000).

Matthias Schwab works in the Frankfurt office of the Boston Consulting Group. He received his Vordiploma in mathematics and geophysics from the Technical University Clausthal-Zellerfeld, following which he studied on a Fulbright Scholarship with Gerry Gardner at the University of Houston and Rice University. He received a PhD in geophysics from the Stanford Exploration Project. He is a member of the Deutsche Studienstiftung, Society of Exploration Geophysicists, American Geophysical Union, and European Association of Geoscientists and Engineers. Contact him at matt@sep.stanford.edu.

Martin Karrenbach is vice president of seismic modeling at Paulsson Geophysical Services, where he works in large-scale seismic modeling and full waveform modeling for preacquisition, interpretation, and feasibility studies for high-resolution seismic imaging. His other professional interests are in software development and signal processing. He received his MS in geophysics from the University of Houston and his PhD in geophysics from Stanford University, where he was a member of the Stanford Exploration Project. He is a member of the Society of Exploration Geophysicists, the European Assn. of Geoscientists & Engineers, and the American Geophysical Union. Contact him at martin@sep.stanford.edu.

Jon Claerbout is a professor of geophysics at Stanford University, where he founded the Stanford Exploration Project. He has consulted for Chevron and was elected to the National Academy of Engineering. He is an honorary member of the Society of Exploration Geophysicists, which awarded him both its highest award, the Maurice Ewing Medal, and the SEG Fessenden Award for outstanding and original work in seismic wave analysis. He is also an honorary member of the European Assn. of Geoscientists & Engineers and recipient of its Erasmus Award. He received his BS in physics and his MS and PhD in geophysics from MIT. Contact him at claerbout@stanford.edu.

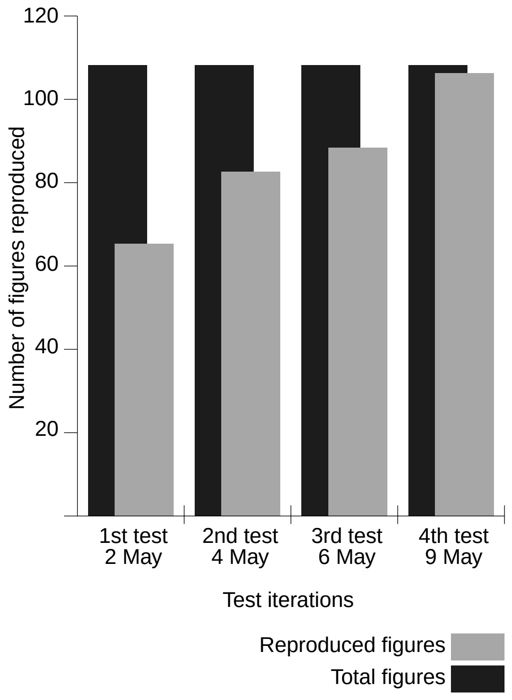

Figure 5. Successfully reproduced figures in a sample document containing 14 articles with 112 easily reproducible figures. To test a document's reproducibility, we use a cycle of burning and rebuilding its results. A simple script can implement such a reproducibility test by invoking the ReDoc rules, which remove and regenerate the document's results independent of the document's content. After each test, the authors had time to make corrections. After the first test, only 60% of the document's easily reproducible figures were in fact reproducible. After the fourth test, almost all figures were reproducible and we published the document.