How to use cake with interactive documents

Jon F. Claerbout and Martin Karrenbach [1]

/usr/local/src/our/word/tutorials/cake

ABSTRACT

My experience with maintaining 300 illustrations has been distilled to two pages of cake rules. Each directory has a `cakefile` containing a list (`FIGLIST`) of buildable figure names. All figures can be built with the command `cake figures` and removed with `cake burn`. Precious figure files (ones not considered replaceable) are named `NAME.save`. Karrenbach's `activeplot` LaTeX macro launches the UNIX command `cake NAME.menu` or `cake NAME.idoc`. The default cake rule is to come up with an Xwindow menu `xtpanel` or to show a plot with `tube` or show a movie with `X11movie`. For unusual actions, you can override the default by making a rule for the target `NAME.action`.

INTRODUCTION

I (JFC) am pleased and proud of the fact that I can easily run my many programs on machines of many vendors, HP, IBM, SUN, DEC, Convex, and maybe more. The reason I can do this so easily is that I learned about cake and Dave Nichols's compiling and linking rules. I am also pleased and proud that I can also just as easily burn and rebuild the 300 or so figures that my book programs make. As Dave Nichols supplied us with `SEP.obj.rules` and `SEP.prog.rules` and a little accompanying documentation, (at SEP this is found in `/usr/local/lib/cake`) here I am supplying you with a little documentation for my creation, `SEP.idoc.rules`, that shows how I do the job of maintaining the 100 cakefiles that do the job.

I assume my readers are familiar with the UNIX `make` utility which is widely used for software maintenance. Here I use a dialect of the `make` language called `cake`. I never could have succeeded in getting the job done with `make`.

[1] **email:** jon@sep.stanford.edu

The main strategy is to isolate the reusable code in two files, `SEP.defs` that I "include" at the beginning of each figure-making cakefile and `SEP.idoc.rules` that I include at the end. Each of my cakefiles starts with

```
#define USRLIBS SEPCLASSLIB
#include <SEP.defs>
```

The first line says that linking must include the library used in all my classes. If you need additional libraries, add them to the end of this line. `SEP.defs` also sets up a default location for my figures because it contains the line

```
#define FIGDIR ./../Fig
```

which means a sister directory to any of my figure-building directories is named `Fig` (which is generally a symbolic link to a nonprecious area). Not everyone at SEP organizes their directories as I do. I put in one directory all files needed to create a family of figures except for the final plot files which go in `FIGDIR`. A rare few of my directories install programs in the official places for system-wide executables and I use Dave's methods for that.

My typical default behavior when I come into a directory is to type `cake` and view a succession of figures emerging from the `tube` program. In the skeleton cakefile below, typing `cake` will show me two figures named `data` and `spectrum`, and it will build them first if they are out-of-date.

```
#define USRLIBS SEPCLASSLIB
#include <SEP.defs>
#define FIGLIST data spectrum
default& : tube
$         comment
$ Application dependent stuff described later belongs here.
$
clean& : jclean
#include <SEP.obj.rules>
#include <SEP.prog.rules>
#include <SEP.idoc.rules>
```

I have a bunch of file names that I have universally decided to be dregs that can always be removed by a `cake clean`. Thus, all my cakefiles contain:

```
clean& : jclean
```

This line means that when someone types `cake clean` that `jclean` must be built. Neither `clean` nor `jclean` are files; they are abstract and the & is a signal for cake not to complain when no file named `clean` appears after the completion of `cake clean`. The rule for building `jclean` is found in `SEP.idoc.rules` and it consists of `rm` commands for the many file types that I consider to be dregs. You probably agree with my definition of dregs but maybe you should check. (Earlier versions considered fortran files to be dregs!)

At the bottom of my cakefiles I included the compilation and linking rules of Dave Nichols, and I include `SEP.idoc.rules`, which is about two pages of cake rules that are common to all my figure building and whose content is our main subject here. First, however, I need to tell you our naming conventions:

```
NAME.rst        source code needing saw,sat, ratfor and fortran passes
NAME.rs         source code needing saw, ratfor and fortran passes
NAME.rt         source code needing sat, ratfor and fortran passes
NAME.r          source code needing ratfor and fortran passes
NAME.f          fortran
NAME.rscm       source code needing saw, ratfor and CM Fortran 90 passes
NAME.rtcm       source code needing sat, ratfor and CM Fortran 90 qpasses
NAME.rcm        source code needing ratfor and CM Fortran 90 passes
NAME.fcm        source code in CM Fortran 90
NAME.o          object file, ready to be linked
NAME.x          executable (sometimes without .x)
NAME.ps         postscript
NAME.ps.save    precious postscript, not to be burned.
NAME.v          vplot (intermediate plot language)
NAME.v.save     precious vplot, not to be burned.
NAME.v3         vplot movie
NAME.v3.save    vplot movie, precious
NAME.H          data history file.
NAME.p          process parameter file
NAME.A          history file of movie  bytes.
NAME.warning    a  file with explanations
```

`SEP.obj.rules` and `SEP.prog.rules` are smart enough that simple linking jobs are done for you with no 1 assistance. Thanks to Dave Nichols, these rules know about the different compiler flags and library names on different machines. For example, if the rule for building the figure named `spectrum` is

```
#define DATA /q2/wz/wz.26.H
FIGDIR/spectrum.v : DATA  myspectrum.x
        <DATA myspectrum.x | Graph > junk.H out=FIGDIR/spectrum.v
```

and your directory contains a fortran source file named `myspectrum.f`, then the included rules will compile `myspectrum.f` and link it to the libraries. In the above rule you can use `myspectrum` in the place of `myspectrum.x`. I prefer `myspectrum.x` because my `jclean` rule knows about `.x` files whereas `jclean` does not know how to identify `myspectrum` as something that needs to be removed at cleaning time.

Often several files must be separately compiled and linked together. Below is how I create the executable `myprog.x` which links a fortran main with some `.o` files.

```
#define MYPROG mymainprog.o sub.o fun.o
myprog.x : MYPROG
        FLINK(MYPROG, myprog.x)
```

Each of my figure-building cakefiles contains the definition of a list of figures that it can build and manipulate. Above, that definition is

```
#define FIGLIST data spectrum
```

The application-specific rules in the cakefile should make `FIGDIR/data.v` and `FIGDIR/spectrum.v` (or possibly corresponding postscript files, but for now nearly all my figures are made in the `vplot` intermediate plot language, and `SEP.idoc.rules` builds the postscript for me). Because I always include `SEP.idoc.rules` in any of my figure-building directories I can issue the following commands:

```
cake burn               # Remove vplot and postscript figures
cake burnps             # Remove only postscript figures
cake figures            # build postscript for all figs
cake vplot              # build vplot for all figs
cake tube               # view all the vplot files
cake idoc               # run thru push all push button actions
cake NAME.tube          # view with tube the file FIGDIR/NAME.v
cake NAME.idoc          # do the electronic book action of figure NAME
cake NAME.menu          # bring up a menu of possible actions for figure NAME
cake install            # install programs in official place
```

Duplicate targets

To make each target, there must be a rule. Your rules add to those in `SEP.idoc.rules`. You cannot override a rule in `SEP.idoc.rules` by making another rule by the same name. (What does happen if you try is beyond the scope of this paper.) If you don't like one of the rules in `SEP.idoc.rules` you can make your own substitute, but your target must have a different name. If you dislike some feature of `SEP.idoc.rules` the best thing to do is convince us to change it. (You might be offering an improvement that would benefit us all, or you might simply need a better explanation than this document.) Less desireable alternatives are (1) jetison `SEP.idoc.rules` (and lose the functionality above) or (2) copy `SEP.idoc.rules` and revise it to your own liking (please explain your difficulty before trying this.).

ESSENTIAL CONTENTS OF SEP.idoc.rules

You don't need to know how `SEP.idoc.rules` works, but it might help you to know. Burning figures regularly leads to the proof that you can rebuild them. This is done by the following line found in `SEP.idoc.rules`

```
burn&:     [[ sub -i X X.burn  FIGLIST ]]
```

Reading `man cake` you see that this line, along with our earlier definition of `FIGLIST` will expand to

```
burn&:     data.burn  spectrum.burn
```

which means that `cake burn` done in this directory is equivalent to `cake data.burn spectrum.burn` which are the "targets" that can be built by the following idoc rule

```
%.burn&:
        -/bin/rm  -f FIGDIR/%.ps
        -/bin/rm  -f FIGDIR/%.v
```

To understand this rule, you need to know that in cakefiles, "%" is a wildcard like "*" in command shells.

We have a special category of figure types that cannot be easily rebuilt so they should not be burned. This is because they are scanned, prepared manually, with exotic hardware, excessive computer time, or because of any other difficulty. (Martin Karrenbach's latex environment will mark such figures NR denoting "non reproducible" (Karrenbach and Nichols, 1990).) To save such files from burning, we give them names of the form NAME`.v.save` and NAME`.ps.save`. From time-to-time we redesign the vplot to postscript program (paper printing is usually black and white, but electronic documents use color) and then we want to burn all the postscript files that can be rebuilt from vplot. Here are the rules I am using now:

```
%.burnps& : if not exist FIGDIR/%.v.save
        -/bin/rm  -f FIGDIR/%.ps

%.burnps& : if     exist FIGDIR/%.v.save
        -/bin/rm  -f FIGDIR/%.ps.save
```

Thus `cake burnps` will not burn your precious plot files if you have no way to rebuild them. (While I am writing this, I am wondering if I should change these rules to something with the `cando` construction.)

Now let us do something constructive. To build all the figures in any of my books or documents, I have a shell that passes over all the cakefiles in the document tree and executes `cake figures`. That evokes the rules below from `SEP.idoc.rules`

```
figures&:  [[ sub -i X X.figure        FIGLIST ]]

%.figure&:  FIGDIR/%.ps.save        if     exist FIGDIR/%.v.save

%.figure&:  FIGDIR/%.ps             if not exist FIGDIR/%.v.save
```

These rules say that a postscript file must be created for every name in `FIGLIST`, either `FIGDIR/%.ps` or `FIGDIR/%.ps.save`. How the figures are created is determined by the following rules:

```
#ifndef PSARGS
#    ifdef HARDCOPY
#       define PSARGS color=n fat=1 fatmult=1.5
#    else
#       define PSARGS color=y
#    endif
#endif

#define PSTEXPEN(A,B) pstexpen FIGDIR/A FIGDIR/B  PSARGS
FIGDIR/%.ps.save : FIGDIR/%.v.save
        PSTEXPEN(%.v.save,%.ps.save)

FIGDIR/%.ps : FIGDIR/%.v
        PSTEXPEN(%.v,%.ps)
```

A few weeks before report generation, we insert in `SEP.defs` a line `#define HARDCOPY` (which you can do anytime in any of your cakefiles), burn all the postscript files, build the paper document, send it to the printer, then remove the line `#define HARDCOPY`, burn all the postscript files and rebuild them again for our electronic documents.

Recall, I said that `cake tube` will take the user through a `tube` view of all the plot files. The tube program will interpret single frame pictures (`NAME.v`) or multiple frame movies (`NAME.v3`). Paper documents can only cope with `NAME.v` files.

```
#ifndef TUBE
#define TUBE tube width=.8 height=.8
#endif

#define TUBEMOVIE ( cando FIGDIR/%.v3 or exist FIGDIR/%.v3.save )
%.tube&: %.tube1        if not TUBEMOVIE

%.tube&: %.tube3        if     TUBEMOVIE
```

These rules choose the movie file if a movie exists or can be made. If a movie cannot be made, the target becomes `%.tube1` is interpreted by

```
%.tube1&:    FIGDIR/%.v       if     cando FIGDIR/%.v
        TUBE FIGDIR/%.v

%.tube1&:    FIGDIR/%.v.save  if exist FIGDIR/%.v.save and  \
                             not cando FIGDIR/%.v
        TUBE FIGDIR/%.v.save
```

These rules say to show the file `NAME.v` if it can be made, and if it cannot, then show the file `NAME.v.save`, if it exists. (Technically, an error condition arises for a few handmade postscript files for which `tube` cannot be used. I should eliminate this error message by including a do-nothing rule, or by calling a postscript previewer where possible.) The rules for `tube3` are like the rules for `tube1`.

INTERACTION

Martin Karrenbach's figure macro `activeplot` converts a button push to an invocation of `cake NAME.menu` or `cake NAME.idoc`. The choice is made in the TEXfile by either setting `\IdocMenu` (which will issue the command `cake NAME.menu`) or setting `\noIdocMenu` (which will issue the command `cake NAME.idoc`). By default a menu will come up with several possible buttons on it. The appearance of the menu is determined by the existence of certain files and the existence and feasibility of certain cake rules. You will see those buttons:

```
INTERACT
SHOW
WARNING/README
BURN
REBUILD
CLEAN
```

Pressing the push button `INTERACT` starts some kind of action most likely a movie or an interactive program. The `SHOW` button displays the figure as it is on the book page, but you can resize it, zoom to full screen or make it smaller. This is useful if you want to compare several figures side by side. The `WARNING/README` pushbutton displays some explanation usally related to burning and rebuilding. The pushbutton `BURN` removes a figure and the `REBUILD` pushbutton regenerates it. The `CLEAN` pushbutton removes all temporary files.

The menu command is captured by lines similar to the following

```
%.menu&:
        xtpanel -cpp -var mechanism cake -var name % -DINTERACT \
                -file builder.xt panel >&/dev/null &
```

The menu will have `REBUILD` and `BURN` buttons if the figure is rebuildable. A figure is judged rebuildable if the cakefile can generate the file `FIGDIR/%.ps`. The command `cake NAME.burn` is issued by pressing the `BURN` button and pressing the `REBUILD` button issues the command `cake NAME.ps` If the figure is not rebuildable those buttons are inactive on the screen menu.

The menu will have an inactive `INTERACTION` button in case the author does not provide some interaction. An default interaction is possible if there exists one or more of the following:

- vplot movie files `FIGDIR/%.v3` or `FIGDIR/%.v3.save`.
- X11 movie files `FIGDIR/%.A` or `FIGDIR/%.A.save`.
- xtpanel files `%.panel`
- editable parameter files `%.p` and menu file `%.panel` and a backup parameter file %.p.orig
- ed1D or zplane startup files

The command `cake NAME.idoc` is issued by pressing the `INTERACTION` button.

Some figures are "hazardous" or impossible to rebuild, although the mechanism for rebuilding is working perfectly. One reason could simply be that it takes too long to rebuild the figure or that you need specific hardware to run the programs. The author can put messages or explanations into the file `NAME.warning`. This will trigger the existence of a button which lets the reader display the warning message.

```
#define FIGWARN ( exist %.warning )
```

If you do not want a menu, you should define your own `%.action` The following two rules from `SEP.idoc.rules` allow you to capture what happens when `%.idoc` gets involved.

```
        %.idoc & :  %.action       if     cando %.action

        %.idoc & :  %.defaction    if not cando %.action
```

For example, if you want the figure caption pushbutton to launch an unusual action (whistle Dixie?) then your cakefile should have a target `NAME.action` that will whistle Dixie. Normally, you will prefer the default actions defined below. (I'll assume you have no `ed1D` or `Zplane` figures.) The default is to show a movie with `X11movie` if there is a movie or if one can be made. Otherwise, invoke `tube` on a vplot file.

```
        %.defaction& : FIGDIR/%.A       if     cando FIGDIR/%.A
                     < FIGDIR/%.A       MOVIE &

        %.defaction& : FIGDIR/%.A.save if not cando FIGDIR/%.A      \
```

```
                                      and exist FIGDIR/%.A.save
                  < FIGDIR/%.A.save MOVIE

%.defaction&: %.tube if  not exist FIGDIR/%.A.save        \
                         and not cando FIGDIR/%.A
```

This means if your cake files build a file called `FIGDIR/%.A`, then your readers get movies when they press your caption pushbuttons. You need write no explicit rules to show movies.

Finally, if the directory contains a file called `paper.tex` you should be able to type `cake read` to bring up `xtex` with included figures or `cake print` to print it. This is because of the rules below.

```
dvi&      :   PAPER.dvi

PAPER.dvi :   PAPER.tex
        texpr TEXPROPTS -v -2 -d PAPER.tex

view& read&:  figures         PAPER.dvi
        xtex                  PAPER.dvi
```

That's all! You have seen all the common part of my 100 cakefiles. If I add any more, it can only be philosophy, generalities, and maybe a few recalled pitfalls.

PHILOSOPHY AND PITFALLS

Two philosophical approaches to programming are "procedural" and "rule based". Fortran and shell language are examples of procedural languages and `cake` and `make` are rule-based languages. Many of the rules in a cakefile each invoke a small shell-like program. Some rules might contain a long string of shell-like commands with many intermediate results. It is a question of philosophy if you want to break up such a long string into separate rules. I generally prefer one long rule which is like a shell. The long string of commands is easier to understand and debug than a bunch of small rules. However, many circumstances will cause me to break up a long rule into parts. The best reason to break a long rule into separate short ones is if some of the short ones are reusable, either in this directory, or in all directories. We never like to see repeated code in any language because it makes maintenance susceptible to inconsistency. Further, it is the isolation of nifty rules that enables us, after improving them, to immediately bring the improvements into effect everywhere. Another reason to break a long rule into parts is if some of the parts are time consuming and we intend frequent restarts from an intermediate place.

What are the rules when the rules are changing? When we start coding in make or cake we are writing rules that will evolve with time. (It is like when the tax law changes.) This issue is particularly confusing to people switching from make to cake.

(make requires you to keep all the intermediate files, so to force a rebuild you just remove any file that is defective because of the changing rules). Suppose you just used the editor to change `sign1=0` to `sign1=1` in the rule below.

```
FIGDIR/spectrum.v : DATA
        <DATA Ft3D sign1=1 | Graph > junk.H out=FIGDIR/spectrum.v
```

If you type `cake figures` the figure may not build because no files have changed their dates. Technically, you should have put `sign1` in a parameter file and then the date of the parameter file would have changed when you changed the value. Often you can use the system program `touch` to change the date of the data or the program, but that is a poor idea if these are system wide objects that don't belong to you. In this case (which I have chosen to be particularly pesky) you could type `cake spectrum.burn`, or you could temporarily change your rule to

```
FIGDIR/spectrum.v : DATA cakefile
        <DATA Ft3D sign1=1 | Graph > junk.H out=FIGDIR/spectrum.v
```

because the date of the `cakefile` itself did change when you changed the value of `sign1`. Later you must remember to remove this figure dependence on the cakefile, or this figure will rebuild every time you change anything in the cakefile.

Finally there are some hazards to coding in the cake language. We might think that rules with no dependencies would never need to run, but the inventor of cake decided (probably by experience) that such rules should always run.

Most hazards arise because there can be many rules and they can interact in unexpected ways. Sometimes cake will attempt to use a rule recursively and will run for a half minute and then give up. I learned this when I tried working with file names like NAME`.save.v`. It is particularly confusing if two rules can make the same target. Which should be used? I think cake should tell you that two rules can make the same target and you should get rid of one of them. Instead it seems that we cannot tell which will be used, or maybe it tries to use both. This hazard is especially likely with the wildcard `%`. Recently I had trouble when I had two similar rules, one for building plot files and one for merging them. These rules were

```
        skmod%.v : mod%.v dat%.v
                vp_SideBySideAniso   mod%.v  dat%.v  > skmod%.v

        %.A %.V %.v : %.H
                <%.H Byte > %.A
                <%.A Ta2vplot title=% labelsz=12 > %.V out=%.v
```

Notice that the second rule creates the same target as the first if "%" takes the value `skmod%`. I'm not sure exactly what caused the trouble but I solved it when I changed one line to read

```
%.A %.V %.v : %.H  if cando %.H
```

UNFINISHED

`SEP.idoc.rules` constitutes only a part of the essential wisdom of interactive documents. Other things you need to worry about are establishing the overall `seplib` environment and X environment. For this I refer you to (Claerbout, 1991) and (?). Also, on any sizable document, there will be libraries and executables that figure-building cakefiles assume to be present but are not actually built by those figure-building cakefiles. In my textbook (Claerbout, 1992) the building process requires a preliminary tour of all the figure-building cakefiles with a `cake install` which builds a handful of executable main programs needed in the book that are not generally needed by the whole of SEP. Recently, I have been experimenting with another arrangement where such programs are kept in a `Src` directory, and cakefiles that need them use a symbolic link to the `Src` directory.

REFERENCES

Claerbout, J. F., 1991, SEPlib and SEP software - computer: SEP–**71**, 283–286.

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

Karrenbach, M., and Nichols, D., 1990, Progress towards the interactive book: SEP–**67**, 149–160.