Debugging Makefiles with remake

Rocky Bernstein

1 Remake

Autotools[Fou09, Fou10a] is still very popular as a framework for configuring and building open-source software. Since it is a collection of smaller tools, such as *autoconf, automake, libtool,* and *m4,* debugging code that it generates can be difficult.

When I wrote my first POSIX shell debugger for *bash,* one of my initial goals was to be able to debug autotools *configure* scripts, and I was rather pleased when it worked. It required, however, writing a custom *bash* module to read the 20,000 lines of shell script into an array much faster than *bash* was able to. (This module has since been incorporated into *bash* as built-in function *readarray.*) It was only after completing this task that I realized a POSIX shell debugger was just one part of the bigger problem of debugging autotools script. Here, I describe the next step in that endeavor, adding debugging to GNU Make[Fou10b, Ber11]. We will see how to use *remake* and a POSIX shell debugger (the one for *bash*) together.

Makefiles have been around for quite a while, and over time, largely through the success of *automake*, they have gotten more complex. *Make* can be somewhat opaque, but after writing the debugger component of *remake,* I can usually solve *make* problems very quickly and easily.

In many programming languages, such as POSIX shell, Perl, Python, Ruby, and Lisp, type expressions or statements have interactive shells to see what happens when they run. Although GNU Make is every bit as dynamic as these other languages, currently there is no such interactive shell. But the debugger briefly described here can serve as a handy substitute.

The programming language Ruby has a really interesting *make* equivalent called *rake*. (If you are writing something from scratch, please consider using both Ruby and *rake*.) But systems administrators often find themselves using tools and code written by others, and much open-source software uses *make*, via *automake*. *Make* is so pervasive that the reference implementations of Ruby use *make* to build themselves.

In keeping with my philosophy of trying to use the smallest hammer that will do the job, this paper shows some of the smallest changes of my forked version of GNU Make. When used in conjunction with one of my POSIX shell debuggers, you can dynamically debug commands issued by GNU Make into the POSIX shell.

1.1 remake –tasks

A useful feature of Ruby's *rake* program is that there is an option to print a list of "tasks" that one can perform. Tasks include things such as building the software, installing it, and running the tests.

In *make* terminology, tasks are a subset of "files" or "targets." However in *make*, we have to distinguish those files which are just supposed to be there in the source code from those that somehow get created; and many of the files that get created represent intermediate steps along the way to producing something larger. I find it good practice to borrow ideas from related tools, and I have added the `--tasks` option from Ruby's *rake.* This handles files in this way: if a target has a command to build it, then it is probably "interesting"; conversely, if there are no commands to build a target, that is, it is only listed as a dependency, then it probably is not interesting—it is there only to support other targets, and when it changes, it triggers other targets to be remade. Also, if a rule is a default rule of *make,* then it is probably not interesting. This would include things like the pattern rules for compiling a C program or extracting something from an archive or source-control system. The same notion of "interesting" is used in debugger stepping.

Here is `remake --tasks` for a typical Makefile system, using the `Makefile` that comes with the GNU Make distribution.

```
$ remake --tasks
.c.o
.c.obj
.dep_segment
CTAGS
ChangeLog
...
NMakefile
README
...
dist
dist-all
dist-bzip2
...
upload-alpha
upload-ftp
```

When I first looked at the output and saw `README` in this list of targets that have commands associated with them, I thought there must be a mistake, because `README` is usually a distribution file. So I broke out the debugger to check what was going on. The answer will become clear below, when I describe how to investigate targets with the debugger.

Another piece of interesting information we learn from this output is that there is a way to make the `ChangeLog` file, presumably from version control, and a way to make just the bzip2 tarball, or upload the distribution to the alpha and FTP sites.

Additionally, targets can have a description added for them so that they appear when the `--tasks` option is given. A description must consist of only one line and begins with `#:`. Here is a `Makefile` tagged this way:

```
#: Build everything
all:
      perl Build --makefile_env_macros 1

#: Create distribution tarball
dist:
```

```
	perl Build --makefile_env_macros 1 dist

#: Build and install package
install:
	perl Build --makefile_env_macros 1 install

#: Create or update MANIFEST file
manifest:
	perl Build --makefile_env_macros 1 manifest

#: Create or update manual pages
manpages:
	perl Build --makefile_env_macros 1 manpages
```

When run with the `--tasks` option we get:

```
all           # Build everything
dist          # Create distribution tarball
install       # Build and install package
manifest # Create or update MANIFEST file
manpages # Create or update manual pages
```

1.2 remake –trace example

Here is a real-world example of tracing and then debugging.

I made a change to the GNU Make source code and I wanted to make a new distribution. I ran the usual command that does this:

```
$ make dist
NEWS not updated; not releasing
make: *** [distdir] Error 1
```

It is not clear how NEWS needs to be updated, but more information might be available by consulting the rules for the `distdir` target.

Now let us run *remake*:

```
$ remake dist
NEWS not updated; not releasing
Makefile:887: *** [distdir] Error 1

#0  distdir at /tmp/remake/Makefile:887
#1  dist at /tmp/remake/Makefile:1004
Command-line invocation:
  "remake dist"
```

This shows additional information: the line number inside the Makefile for target `distdir` (887), the target that got us to this one, and the command-line invocation.

With standard *make*, we could use the name of the target `distdir` and search inside `Makefile` for that, but frequently it might not be in the top-level Makefile. Instead, it might be in some recursive invocation or in a file included from the top-level Makefle. The traceback information and file names reduce the detective work needed.

Now let us run *remake* with tracing turned on:

```
$ remake --trace dist
GNU Make 3.82+dbg-0.7.dev
...
Reading makefiles...
Updating goal targets....
  /tmp/remake/Makefile:1004 File 'dist' does not exist.
    /tmp/remake/Makefile:887 File 'distdir' does not exist.
    /tmp/remake/Makefile:887 Must remake target 'distdir'.
to be continued...
```

The indentation in the lines containing file name and line numbers gives target level nesting: target `distdir` was asked to be remade because it is a dependency of target `dist`. With this option, we show the dependency nesting as we build and traverse the tree. Without it, we show these only at the point of error, if there is one.

```
Invoking recipe from Makefile:888 to update target 'distdir'.
##>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

```
case `sed 15q ./NEWS` in \
*"3.82+dbg-0.7.dev"*) : ;; \
*) \
  echo "NEWS not updated; not releasing" 1>&2; \
  exit 1;; \
esac
##<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
NEWS not updated; not releasing
Makefile:887: *** [distdir] Error 1

#0  distdir at /tmp/remake/Makefile:887
#1  dist at /tmp/remake/Makefile:1004
Command-line invocation:
  "remake -x dist"
```

So finally we get the commands that were run to get that message, indicating which check was performed.

Another difference between *remake* and standard GNU Make is that *remake* has lines of the form:

```
##>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
##<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
```

These lines serve to separate shell commands about to be run from the output that they produce.

In brief, the changes to GNU Make to improve tracing and error reporting are:

- The Makefile file name and the line inside this file are reported when referring to a target
- On error:
  - a stack of relevant targets is shown, again with their locations
  - the command invocation used to run *make* is shown
  - an option allows for entering the debugger on error
- Shell input that is about to be run is separated from the output in running that shell code
- a `--tasks` option prints a list of interesting targets and any associated description line for each

1.3 remake –debugger example

The tracing described in the previous section will be enough for some purposes. But we can make the computer do more work to show us what is going on by using the built-in debugger.

Why does `README` appear when we run `rake --tasks` ? We can ask the debugger to describe the target `README`:

```
$ remake --debugger
GNU Make 3.82+dbg-0.7.dev
Reading makefiles...
Updating makefiles....
-> (/tmp/remake/Makefile:477)
Makefile: Makefile.in config.status
remake<0> target README
README: README.template Makefile
#  Implicit rule search has not been done.
#  Implicit/static pattern stem: 'README'
#  Modification time never checked.
#  File has not been updated.
#  Commands not yet started.
# automatic
# @ := README
# automatic
...
# < := README.template
# automatic
...
#  commands to execute (from 'Makefile', line 1329):
rm -f $@
sed -e 's@%VERSION%@$(VERSION)@g' \
    -e 's@%PACKAGE%@$(PACKAGE)@g' \
    $< > $@
chmod a-w $@
remake<1>
```

The file `README` is created from `README.template`. In the commands section, there are a number of expanded variables such as `$@` and `$<`. Earlier though, the values of the automatic variables `@` and `<` are shown; here they are `README` and `README.template` respectively. If, however, we want *remake* to do the expansion when showing the commands, there is an option to the `target` command for that:

```
remake<1> target README expand
README:
#  commands to execute (from 'Makefile', line 1329):
rm -f README
sed -e 's@%VERSION%@3.82+dbg-0.7.dev@g' \
    -e 's@%PACKAGE%@remake@g' \
    README.template > README
chmod a-w README
remake<1>
```

Although is is not immediately apparent, some expansion was done in showing the target and dependencies. Line 1328 in file `Makefile` looks like this:

```
$(TEMPLATES) : % : %.template Makefile
```

The debugger command `expand` can be used to get the expanded value of the variable `TEMPLATES`:

```
remake<2> expand TEMPLATES
Makefile:1319 (origin: makefile) TEMPLATES := README README.DOS ...
```

Now we return to tracking down what was happening when we tried to run `make dist`. Again we go into the debugger:

```
$ remake --debugger dist
GNU Make 3.82+dbg-0.7.dev
...
Reading makefiles...
Updating makefiles....
-> (/tmp/remake/Makefile:477)
Makefile: Makefile.in config.status
```

It appears that the first thing that is done is to check whether the Makefile itself is up to date. As before, we could list information from the target that we crashed on, `distdir`. However instead let us run until the target:

```
remake<0> continue distdir run
Breakpoint 1 on target distdir: file Makefile, line 887.
Updating goal targets....
  /tmp/remake/Makefile:1004 File 'dist' does not exist.
    /tmp/remake/Makefile:887 File 'distdir' does not exist.
.. (/tmp/remake/Makefile:887)
distdir
```

There are three interesting points in time when updating a target:

1. before checking dependencies of the target
2. after checking but before running commands to update the target
3. after runnning commands when the target update is finished

Adding `run` to the end of `continue distdir` causes us to stop after dependency checking.

The debugger first stopped before dependency checking, as shown by an icon, the two-character arrow `->`, so it lists the dependencies for the target. For the `Makefile` target, they were `Makefile.in` and `config.status`. After continuing, it next stops dependency checking, so dependencies of the target are not automatically shown, unless explicitly requested with `target` just as for the commands.

A common problem in designing this kind of tool is trying to figure out how to cut down the amount of information shown. We usually do not want a list of all dependencies for `distdir` here since that would include a list of all of the files in the distribution. With the `--tasks` option above, files without associated commands are dropped from the listing.

Another indication that the debugger stopped after dependency checking is that the two-character icon is `..` rather than `->`. I try to use analogous *gdb* commands when possible. Here, the *gdb*-like command `info program` makes the stopping place more explicit:

```
remake<3> info program
Starting directory `/tmp/remake'
Program invocation:
 remake  -X distdir
Recursion level: 0
Line 887 of "/tmp/remake/Makefile"
Program is stopped after rule-prerequisite checking.
```

At this point we can list the commands that are to be run next using the `target` command, which shows information regarding a target. We will use variables that have been set up by GNU Make when giving a target name. As we saw when listing variables for `README`, `@` is an automatically set variable containing the name of the current target. Since we have run to target `distdir`, `@` is set to that.

```
remake<1> target @ commands

distdir:
#  commands to execute (from `Makefile', line 888):
@case `sed 15q $(srcdir)/NEWS` in \
*"$(VERSION)"*) : ;; \
*) \
echo "NEWS not updated; not releasing" 1>\&2; \
exit 1;; \
esac
@list='$(MANS)'; if test -n "$$list"; then \
list=`for p in $$list; do \
if test -f $$p; then d=; else d="$(srcdir)/"; fi; \
if test -f "$$d$$p"; then echo "$$d$$p"; else :; fi; done`; \
.. about 90 other lines.
```

Makefile commands can be confusing because there are two sources for variables: GNU Make variables and POSIX-shell variables. Here we see things like `$(VERSION)` which is a GNU make variable and `$$p` which is the POSIX-shell variable `$p`. An extra `$` needs to be added in the Makefile. We can ask the debugger to expand all of the Makefile variables, but instead, let us write this code out to a file using the `write` command:

```
remake<2> write
File "/tmp/distdir.sh" written.
```

We can use the *bash* debugger *bashdb* to debug the rest.

```
remake<3> quit
remake: That's all, folks...
$ bashdb /tmp/distdir.sh
bash debugger, bashdb, release 4.2-0.7
...
(/tmp/distdir.sh:4):
4: case `sed 15q ./NEWS` in \
bashdb<3> step
(/tmp/distdir.sh:4):
4: case `sed 15q ./NEWS` in \
sed 15q ./NEWS
```

If we do not know what `sed 15q ./NEWS` does, rather than look this up in a manual, we can let the debugger show us. The parentheses in the *bashdb* prompt mean that we are inside a subshell, the backtick part of `` `sed 15q ./NEWS` ``.

A useful command I added not too long ago to the debuggers is `eval` without any arguments. Here, it takes the line that is about to be run and runs it.

```
bashdb<(4)> eval
eval: sed 15q ./NEWS
Version 3.82+dbg-0.6
GNU make NEWS
 History of user-visible changes.
28 July 2010
...
```

One more step and we go to where we do not want to be:

```
bashdb<(5)> step
(/tmp/distdir.sh:7):
 7:   echo "NEWS not updated; not releasing" 1>&2; \
bashdb<6> list
 2:    #/tmp/remake/Makefile:887
 3:    #cd /tmp/remake
 4:    case `sed 15q ./NEWS` in \
 5:    *"3.82+dbg-0.7.dev"*) : ;; \
 6:    *) \
 7: =>   echo "NEWS not updated; not releasing" 1>&2; \
 8:      exit 1;; \
 9:    esac
10:    @list='make.1'; if test -n "$list"; then \
11:      list=`for p in $list; do \
bashdb<7>
```

What is wrong is that we were looking for `3.82+dbg+0.7dev` inside the first 15 lines of the file NEWS and we did not find that.

The above example barely scratches the surface of what is available in both my GNU Make debugger and my POSIX shell debuggers. There is extensive help inside the debuggers and in the online manuals `http://bashdb.sourceforge.net/remake/remake.html/index.html` and `http://bashdb.sourceforge.net/bashdb.html`.

1.4 History and Acknowledgments

The idea for a GNU Make debugger came about after I had completed a debugger for *bash*[Ber09] and realized that there was much more to debugging distribution building in *autoconf* and *automake* scripts than just the *configure* script. So I first floated the idea in freshmeat forum[McC03]. A year later, in response to a challenge[Smi04], I wrote the first code without much trouble.

GNU Make already had a wealth of debugging information stored, so all that was needed was to keep track of a dependency stack and add calls to a REPL (read, eval, print loop) at appropriate times. Delving into the code to figure out the right times and places was the bulk of the hard work.

One suggestion is to display a tree or subtree of targets, possibly as a graph. Unfortunately, GNU Make does not save a tree of targets. Instead, it grows the branch it needs as it traverses targets and removes it afterwards. In order to provide debugging, I had to extend the code to save information from the current target back to the goal target.

So, some target actions can affect whether subsequent targets are up-to-date or not. To make things more complex, targets can be patterns that dynamically match the files created at run-time, and short of "building" the code, one can only give an approximation of existing dependencies.

I would like to thank Calyxa D. Tokay for her constant encouragement, and Stuart Frankel for turning my jumble of ideas into a slightly more coherent and well-organized paper. Tthe anonymous reviewers' comments were very helpful.

1.5 Availability

The home page for this project is `http://bashdb.sourceforge.net/remake/`. Download links for source code can be found there.

Yaroslav Halchenko has been providing Debian packages. The `git` source repository is at:
`https://github.com/rocky/remake`.

References

[Ber09] Rocky Bernstein. *Debugging with the Bash Debugger*, 4.2-0.8 edition, April 2009. Available from `http://bashdb.sourceforge.net/bashdbOutline.html`.

[Ber11] Rocky Bernstein. *Remake — GNU Make with comprehensible tracing and a debugger*, 3.82+dbg-0.7 edition, October 2011. Available from `http://bashdb.sourceforge.net/remake`.

[Fou09] Free Software Foundation. *GNU Automake*, 1.11.1 edition, July 2009. Available from `http://sources.redhat.com/automake/`.

[Fou10a] Free Software Foundation. *GNU Autoconf*, 2.6.8 edition, September 2010. Available from `http://www.gnu.org/software/autoconf/`.

[Fou10b] Free Software Foundation. *GNU Make*, 3.82 edition, July 2010. Available from `http://www.gnu.org/software/make/`.

[McC03] Andrew McCall. Stop the autoconf insanity! why we need a new build system., June 2003. Available from `http://freshmeat.net/articles/stop-the-autoconf-insanity-why-we-need-a-new-build-system`.

[Smi04] Paul D. Smith. *Re: Adding debugging to GNU make (Mailing lists are a disaster lately!)*, March 2004. Available from `https://lists.gnu.org/archive/html/make-alpha/2004-03/msg00001.html`.