Going from a simple Makefile to Autotools

I started looking into Autotools, to make it easier to run my code on multiple platforms.

Naturally you can use cmake, or scons, or waf, or ninja, or tup, all of which are interesting in their own respect. But none of them has seen the amount of testing which went into autotools, and none of them have the amount of tweaks needed to support about every system under the sun. And I recently found pyconfigure which allows using autotools with python and offers detection of library features.

Update 2016: Contains some cargo-cult-programming — my current setup is cleaner thanks to using AC_CONFIG_LINKS in configure.ac. For quick setup of a Makefile for a shell script, you can also use conf.

Update 2025: Migrated here from its old location, because updating cmake broke the build of a program I use daily. I don't want volatile build tools.

Contents

I had already used Makefiles for easily storing the build information of anything from python projects (python setup.py build) to my PhD thesis with all the required graphs.

I also had used scons for those same tasks.

But I wanted to test, what autotools have to offer. And I found no simple guide which showed me how to migrate from a Makefile to autotools - and what I could gain through that.

So I decided to write one.

1 My Makefile

The starting point is the Makefile I use for building my PhD. That's pretty generic and just uses the most basic features of make.

If you do not know it yet: A basic makefile has really simple syntax:

```
# comments start with #
thing : required source files # separated by spaces
    build command
    second build command
# ^ this is a TAB.
```

The code above is a rule. If you put a file with this content into some folder using the filename Makefile and then run make thing in that folder (in a shell), the program "make" will check whether the source files have been changed after it last created the thing and if they have been changed, it will execute the build commands.

You can use things from other rules as source file for your thing and make will figure out all the tasks needed to create your thing.

My Makefile below creates plots from data and then builds a PDF from an org-mode file.

```
all: doktorarbeit.pdf sink.pdf

sink.pdf : sink.tex images/comp-t3-s07-tem-boas.png images/comp-t3-s07-tem-bona.png i
    pdflatex sink.tex
    rm -f  *_flymake* flymake* *.log *.out *.toc *.aux *.snm *.nav *.vrb # kill litte

comp-t3-s07-tem-boas.png comp-t3-s07-tem-bona.png : nee-comp.pyx nee-comp.txt
    pyxplot nee-comp.pyx

doktorarbeit.pdf : doktorarbeit.org
    emacs --batch --visit "doktorarbeit.org" --funcall org-export-as-pdf
```

2 Feature Equality

The first step is simple: How can I replicate with autotools what I did with the plain Makefile?

For that I create the files configure.ac and Makefile.am. The basic Makefile.am is simply my Makefile without any changes.

The configure.ac sets the project name, inits automake and tells autoreconf to generate a Makefile.

```
dnl run `autoreconf -i` to generate a configure script.
dnl Then run ./configure to generate a Makefile.
dnl Finally run make to generate the project.

AC_INIT([Doktorarbeit Inverse GHG], [0.1], [arne.babenhauserheide@kit.edu])
dnl we use the build type foreign here instead of gnu because I do not have a NEWS fi
AM_INIT_AUTOMAKE([foreign])
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

Now, if I run ‘autoreconf -i‘ it generates a Makefile for me. Nothing fancy here: The Makefile just does what my old Makefile did.

> First milestone reached: Feature Equality!

But the generated Makefile is much bigger, offers real –help output and can generate a distribution - which does not work yet, because it misses the source files. But it clearly tells me that with ‘make distcheck‘.

3 make dist: distributing the project

Since ‘make dist‘ does not work yet, let’s change that.

. . . easier said than done. It took me the better part of a day to figure out how to make it happy. Problems there:

- I have to explicitely give automake the list of sources so it can copy them to the distributed package.
- distcheck uses a separate build dir. Yes, this is the clean way, but it needs some hacking to get everything to work.
- I use pyxplot for generating some plots. Pyxplot does not have a way (I know of) to search for datafiles in a different folder. I have to copy the files to the build dir and kill them after the build. But only if I use a separate build dir.

- pdflatex can't find included images. I have to adapt the TEXINPUT environment variable to give it the srcdir as additional search path.
- Some of my commands litter the build directory with temporary or intermediate files. I have to clean them up.

So, after much haggling with autotools, I have a working make distcheck:

```
pdf_DATA = sink.pdf doktorarbeit.pdf

sink = sink.tex
pkgdata_DATA = images/comp-t3-s07-tem-boas.png images/comp-t3-s07-tem-bona.png
dist_pkgdata_DATA = images/bona-marble.png images/boas-marble.png

plotdir = .
dist_plot_DATA = nee-comp.pyx nee-comp.txt

doktorarbeit = doktorarbeit.org

EXTRA_DIST = ${sink} ${dist_pkgdata_DATA} ${doktorarbeit}

MOSTLYCLEANFILES = \#* *~ *.bak # kill editor backups
CLEANFILES = ${pdf_DATA}
DISTCLEANFILES = ${pkgdata_DATA}

sink.pdf : ${sink} ${pkgdata_DATA} ${dist_pkgdata_DATA}
    TEXINPUTS=${TEXINPUTS}:$(srcdir)/:$(srcdir)/images// pdflatex $<
    rm -f  *_flymake* flymake* *.log *.out *.toc *.aux *.snm *.nav *.vrb # kill litte

${pkgdata_DATA} : ${dist_plot_DATA}
    $(foreach i,$^,if test "$(i)" != "$(notdir $(i))"; then\
      cp -u "$(i)" "$(notdir $(i))"; fi;)
    ${MKDIR_P} images
    pyxplot $<
    $(foreach i,$^,if test "$(i)" != "$(notdir $(i))"; then\
      rm -f "$(notdir $(i))"; fi;)

doktorarbeit.pdf : ${doktorarbeit}
    if test "$<" != "$(notdir $<)"; then cp -u "$<" "$(notdir $<)"; fi
    emacs --batch --visit "$(notdir $<)" --funcall org-export-as-pdf
    if test "$<" != "$(notdir $<)"; then \
      rm -f "$(notdir $<)"; \
      rm -f $(basename $(notdir $<)).tex $(basename $(notdir $<)).tex~; else \
      rm -f $(basename $<).tex $(basename $<).tex~; fi
```

You might recognize that this is not the simple Makefile anymore. It is now a setup which defines files for distribution and has custom rules for preparing script runs and for cleanup.

But I can now make a fully working distribution, so when I want to publish my PhD thesis, I can simply add the generated release tarball. I work in a Mercurial repo, so I would more likely just include the repo, but there might be reasons for leaving out the history - and be it only that the history might grow quite big.

> Second milestone reached: make distcheck!

An advantage is that in the process of preparing the dist, my automake file got cleanly separated into a section defining files and dependencies and one defining build rules.

But I now also understand where newer build tools like scons got their inspiration for the abstractions they use.

I should note, however, that if you were to build a software project in one of the languages supported by automake (C, C++, Python and quite a few others), I would not have needed to specify the build rules myself.

And being able to freely mix the dependency declaration in automake style with Makefile rules gives a lot of flexibility which I missed in scons.

4 Finding programs

Now I can build and distribute my project, but I cannot yet make sure that the programs I need for building actually exist.

And that's finally something which can really help my build, because it gives clear error messages when something is missing, and it allows users to specify which of these programs to use via the configure script. For example I could now build 5 different versions of Emacs and try the build with each of them.

Also I added cross compilation support, though that is a bit over the top for simple PDF creation :)

Firstoff I edited my configure.ac to check for the tools:

```
dnl run `autoreconf -i` to generate a configure script.
dnl Then run ./configure to generate a Makefile.
dnl Finally run make to generate the project.

AC_INIT([Doktorarbeit Inverse GHG], [0.1], [arne.babenhauserheide@kit.edu])
# Check for programs I need for my build
AC_CANONICAL_TARGET
```

```
AC_ARG_VAR([emacs], [How to call Emacs.])
AC_CHECK_TARGET_TOOL([emacs], [emacs], [no])
AC_ARG_VAR([pyxplot], [How to call the Pyxplot plotting tool.])
AC_CHECK_TARGET_TOOL([pyxplot], [pyxplot], [no])
AC_ARG_VAR([pdflatex], [How to call pdflatex.])
AC_CHECK_TARGET_TOOL([pdflatex], [pdflatex], [no])
AS_IF([test "x$pdflatex" = "xno"], [AC_MSG_ERROR([cannot find pdflatex.])])
AS_IF([test "x$emacs" = "xno"], [AC_MSG_ERROR([cannot find Emacs.])])
AS_IF([test "x$pyxplot" = "xno"], [AC_MSG_ERROR([cannot find pyxplot.])])
# Run automake
AM_INIT_AUTOMAKE([foreign])
AM_MAINTAINER_MODE([enable])
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

And then I used the created variables in the Makefile.am: See the @-characters around the program names.

```
pdf_DATA = sink.pdf doktorarbeit.pdf

sink = sink.tex
pkgdata_DATA = images/comp-t3-s07-tem-boas.png images/comp-t3-s07-tem-bona.png
dist_pkgdata_DATA = images/bona-marble.png images/boas-marble.png

plotdir = .
dist_plot_DATA = nee-comp.pyx nee-comp.txt

doktorarbeit = doktorarbeit.org

EXTRA_DIST = ${sink} ${dist_pkgdata_DATA} ${doktorarbeit}

MOSTLYCLEANFILES = \#* *~ *.bak # kill editor backups
CLEANFILES = ${pdf_DATA}
DISTCLEANFILES = ${pkgdata_DATA}

sink.pdf : ${sink} ${pkgdata_DATA} ${dist_pkgdata_DATA}
    TEXINPUTS=${TEXINPUTS}:$(srcdir)/:$(srcdir)/images// @pdflatex@ $<
    rm -f  *_flymake* flymake* *.log *.out *.toc *.aux *.snm *.nav *.vrb # kill litte

${pkgdata_DATA} : ${dist_plot_DATA}
    $(foreach i,$^,if test "$(i)" != "$(notdir $(i))"; then cp -u "$(i)" "$(notdir $(
    ${MKDIR_P} images
    @pyxplot@ $<
    $(foreach i,$^,if test "$(i)" != "$(notdir $(i))"; then rm -f "$(notdir $(i))"; f
```

```
doktorarbeit.pdf : ${doktorarbeit}
    if test "$<" != "$(notdir $<)"; then cp -u "$<" "$(notdir $<)"; fi
    @emacs@ --batch --visit "$(notdir $<)" --funcall org-export-as-pdf
    if test "$<" != "$(notdir $<)"; \
        then rm -f "$(notdir $<)"; \
        rm -f $(basename $(notdir $<)).tex $(basename $(notdir $<)).tex~; else \
        rm -f $(basename $<).tex $(basename $<).tex~; fi
```

Third milestone reached: Checking for required tools!

5 Summary

With this I'm at the limit of the advantages of autotools for my simple project.

They allow me to create and check a distribution tarball with relative ease (if I know how to do it), and I can use them to check for tools - and to specify alternative tools via the commandline.

For a C or C++ project, autotools would have given me a lot of other things for free, but even the basic features shown here can be useful.

You have to judge for yourself if they outweight the cost of moving away from the dead simple Makefile syntax.

5.1 Comparing SCons

A little bonus I want to share.

I also wrote an scons script as alternative to my Makefile which I think might be interesting to you. It is almost equivalent to my Makefile since it can build my files, but scons does not match the features of the full autotools build and distribution system. Missing: Clean up temporary files and create a validated distribution tarball.

You might notice that the more declarative style with explicit dependency information looks quite a bit more similar to automake than to plain Makefiles.

Missing in SCons: No distcheck!

The following is my SConstruct file:

```
#!/usr/bin/env python
## I need a couple of special builders for my projects
# the $SOURCE replacement only uses the first source file.
```

```
# $SOURCES gives all.
# specifying all source files makes it possible to rerun the build if
# a single source file changed.
orgexportpdf = 'emacs --batch --visit "$SOURCE" --funcall org-export-as-pdf'
pyxplot = 'pyxplot $SOURCE'
# pdflatex is quite dirty. I directly clean up after it with rm.
pdflatex = ('pdflatex $SOURCE -o $TARGET; rm -f  *_flymake* flymake* *.log '
            '*.out *.toc *.aux *.snm *.nav *.vrb')

# build the PhD thesis from emacs org-mode.
Command("doktorarbeit.pdf", "doktorarbeit.org",
        orgexportpdf)

# create plots
Command(["images/comp-t3-s07-tem-boas.png",
         "images/comp-t3-s07-tem-bona.png"],
        ["nee-comp.pyx",
         "nee-comp.txt"],
        pyxplot)

# build my sink.pdf
Command("sink.pdf",
        ["sink.tex",
         "images/comp-t3-s07-tem-boas.png",
         "images/comp-t3-s07-tem-bona.png",
         "images/bona-marble.png",
         "images/boas-marble.png"],
        pdflatex)

# My editors leave tempfiles around. I want them gone after a build
# clean. This is not yet supported!
tempfiles = Glob('*~') + Glob('#*#') + Glob('*.bak')
# using this here would run the cleaning on every run.
#Command("clean", [], Delete(tempfiles))
```

If you want to integrate building with scons into a Makefile, the following lines allow you to run scons with ‘make sconsrun‘. You might have to also mark sconsrun as .PHONY.

```
sconsrun : scons
    python scons/bootstrap.py -Q

scons :
    hg clone https://bitbucket.org/ArneBab/scons
```

Here you can see part of the beauty of autotools, because you can just add this to your Makefile.am instead of the Makefile and it will work inside the full autotools project (though without the dist-integration). So autotools is a real superset of simple Makefiles.

6 Links

- Autotools: practitioner's guide
  Useful examples — and *prefixes*.
- Autotools Mythbuster
  With years of Gentoo experience: How **not** to enrage your distributions.
- Autoconf Manual
  The official manual. Detailed, huge and hard to digest.
- Automake Manual
  As above. Keep it handy as reference.