Life after Make - Building Software with SCons

Joe VanAndel

NCAR

Atmospheric Technology Division

Research Technology Facility

Audience survey

- Do you enjoy using autoconf and make? (or do you just tolerate them!)
- How frequently do you run “ make clean” - just to be safe!
- What alternatives have you tried?

What's wrong with Make(1)

- No built-in dependency tools
- Quirky syntax – tabs matter – can't cut/paste text from makefiles!
- Another “ little language” to learn.
- Uses shell to extend Make's capabilities (shell is an awful programming language!)

```
include/linux/version.h: ./Makefile
        @expr length "$(KERNELRELEASE)" \<= $(uts_len)
> /dev/null || \
        (echo KERNELRELEASE \"$(KERNELRELEASE)\"
exceeds $(uts_len) characters >&2; false)
       @echo \#define UTS_RELEASE
\"$(KERNELRELEASE)\" > .ver
       @echo \#define LINUX_VERSION_CODE `expr
$(VERSION) \\* 65536 + $(PATCHLEVEL) \\* 256 +
$(SUBLEVEL)` >> .ver
       @echo '#define KERNEL_VERSION(a,b,c) (((a) <<
16) + ((b) << 8) + (c))' >>.ver
      @mv -f .ver $@
```

What's Wrong with Make(2)

- Dependence on timestamps –
  - Clock skew in networked development systems
  - Fooled by restoring old source files
- Make doesn't know that changing compiler flags builds different object files
  - Debug flags (-g)
  - Optimize flags (-02)
  - Pre-processor definitions (-DDEBUG_THIS)

What's Wrong with make(3)

- Hard to build multi-directory projects with libraries, include file dependencies
- Unreliable – frequent use of 'make clean' to insure everything is built consistently
- Scaling issues – Linux kernel has 19000 lines of Makefiles! (and we still have to run 'make mrproper' and 'make dep')
- Multiple versions of 'make' exist, each with own quirks, features

What's wrong with autoconf/automake (1)

- Mix of shell and m4 – no high level programming language
- Requires multiple time consuming passes to regenerate configure scripts and Makefiles.
  - Aclocal
  - Autoheader
  - Automake
  - Autoconf
  - configure

What's wrong with autoconf/automake (2)

- Generates 11000 line configure shell scripts – hard to debug
- Leading edge packages require non-standard versions of autoconf/automake
- Although (usually) easy for end-users, lots of hassles for developers.

What is SCons? (1 of 2)

- Next-generation build tool (i.e., yet another `Make` replacement…)
- Configuration files are Python scripts
- Embeddable: build engine is separate from interface
- Supports: C, C++, Java (including `jar`, `javah` and `RMIC`), Fortran, Lex, Yacc, M4, PDF, PostScript, Tar, Zip, RCS, SCCS, CVS, BitKeeper, Perforce, precompiled header files, Microsoft resource files (`.res`), Visual Studio files (v6: `.dsp`, `.dsw`, .NET: `.sln`, `.vcproj`)
- MD5 signatures
- Automatic dependency scanning

What is SCons? (2 of 2)

- Integrated `Autoconf`-like functionality
- Extensible for other tools/file types
- Cross-platform
- Improved parallel build model
- Dead simple installation on multiple platforms from multiple formats: `.tar.gz, .zip, .rpm, .deb, .exe`
- Rigorous regression testing development methodology
- Open Source: MIT license

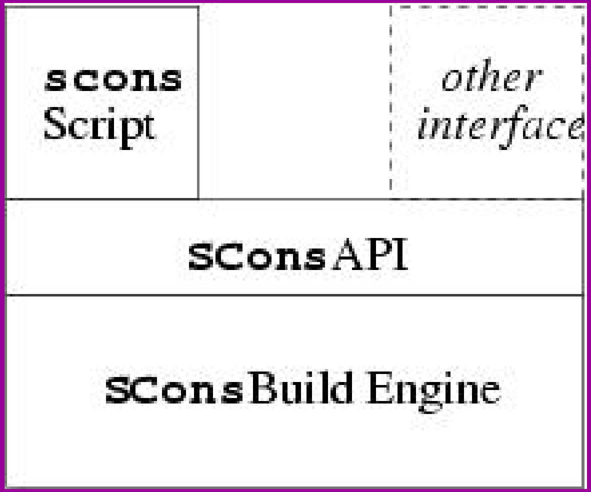
Example: C program

```
env = Environment()
env.Program('foo.c')
```

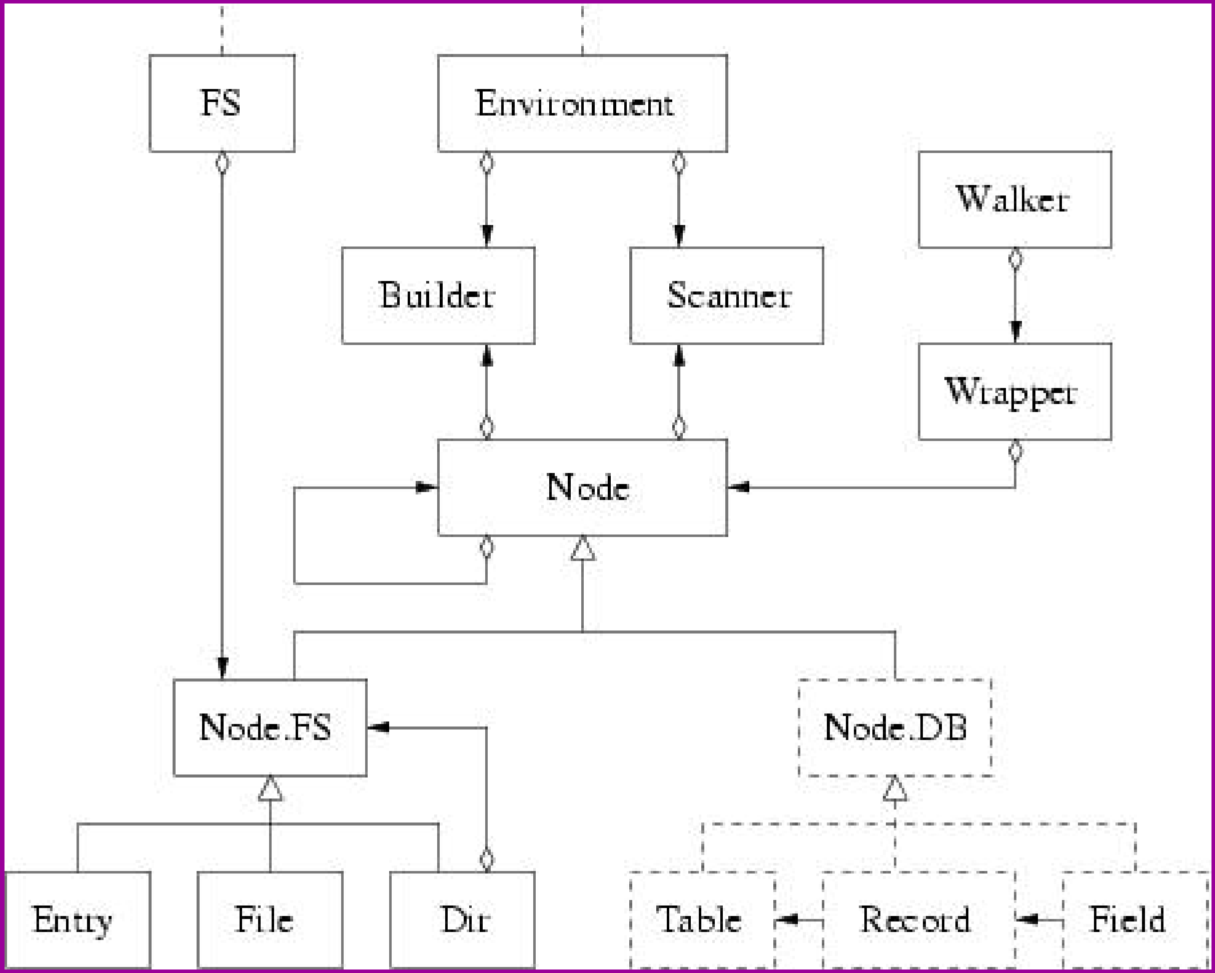
Example: 2 C programs

```
env = Environment()
env.Program('foo.c')

# 'bar' needs its own CPP define

env2 = env.Copy(CCFLAGS = '-DBAR')
env2.Program(target = 'bar',
             source = ['f1.c', 'f2.c'])
```

Scons Architecture



Build Engine



Build Engine Components

- Sconscript files use an *Environment* to communicate build information to scons
- *Environment* contains:
  - Library names
  - Library paths
  - CPP defines
  - *Scanners* – file dependencies
  - *Builders* - compile,link
  - *Nodes* - represent files/directories

Example: Multiple languages

```
env = Environment(LIBS = Split('m util'))
src = Split("""main.c parser.y
               file.cpp calc.f""")
env.Program('bar', src)
```

Example: Library build

```
env = Environment(CCFLAGS = '-O')
env.StaticLibrary('mine',
                  Split(‘f1.c f2.c'))

# Note:

# on Unix/Linux builds
#     'libmine.a'

#  on Windows builds
#    mine.lib
```

Example: customizing SCons for a 40 directory project

- 32 packages with
  - Header files (-I /opt/ACE-5.3.1)
  - Libraries (-L /opt/ACE-5.3.1/ace -lACE)
  - CPP definitions (-D ACE_HAS_QT)

```
#raddx/SConstruct

from atd_scons import
 Pkg_Environment
env = Pkg_Environment()

def RootSetup(env):
    env.Append (CCFLAGS
['-Wall','-Wno-char-subscripts'],
CPPPATH=['.','#'])
    return env

def DebugSetup(env):
    RootSetup(env)
    env.Append(CCFLAGS='-g')
    return env
```

```
def NoUnused(env):
    env.Append (
       CCFLAGS=['-Wno-unused'])
    return env

env.GlobalSetup (lambda env:
 NoUnused(DebugSetup(env)))

Export('env')
Sconscript(dirs=Split("""
  rtfcommon acex dbx logx inix
 rtf_disp
  radd eldora rdow spol ascope"""))
```

Example: Building with subdirectories

```
SConscript('enet_ingest/SConscript')
SConscript('merge_beam/SConscript')
SConscript('sim_piraq/SConscript')
SConscript('util/SConscript')
SConscript('product_gen/SConscript')
SConscript('display/SConscript')
SConscript('perp/SConscript')
```

Example: define library and include dependencies

```
import os
OPT_PREFIX='/opt/local_rh90'
def PKG_INILIB(env):
    env.Append(LIBPATH=[os.path.join
(OPT_PREFIX, 'lib')],)
    env.Append(LIBS=['ini',])

    env.Append(CPPPATH=[os.path.join
(OPT_PREFIX,'include'),  ] )
Export('PKG_INILIB')
```

Example: specify package dependencies

```
Import('env')
  my_env = env.Create('spol.enet_ingest'
my_env.Require (Split("""PKG_SPOL
 PKG_XMLRPC PKG_ACEX PKG_RTFCOMMON
PKG_INIX PKG_RDOW PKG_DBX PKG_LOGX
 PKG_INILIB"""))

Default(my_env.Program
 (target='enet_ingest' , source =
Split(""" main.cpp EnetIngestApp.cpp
viraq_handler.cpp  xdwell_handler.cpp
ingest_handler.cpp """) ))
```

Verifying a header file exists

```
conf = Configure(my_env)
if not conf.CheckCXXHeader
  ("num_util.h"):
   print "Missing the num_util extension
  to Boost Python"
   Exit(1)
```

Verifying Qt libraries and header files

```
def CheckQt(context, qtdir):
    context.Message( 'Checking for qt ...' )
    lastLIBS = context.env['LIBS']
    lastLIBPATH = context.env['LIBPATH']
    lastCPPPATH= context.env['CPPPATH']
    context.env.Append(LIBS = 'qt', LIBPATH = qtdir +
'/lib', CPPPATH = qtdir + '/include' )
    ret = context.TryLink("""
#include <qapp.h>
int main(int argc, char **argv) {
  QApplication qapp(argc, argv);
  return 0;
}
""")
    if not ret:
        context.env.Replace(LIBS = lastLIBS,
LIBPATH=lastLIBPATH, CPPPATH=lastCPPPATH)
    context.Result( ret )
    return ret

env = Environment()
conf = Configure( env, custom_tests = { 'CheckQt' : CheckQt
} )
if not conf.CheckQt('/usr/lib/qt'):
    print 'We really need qt!'
    Exit(1)
env = conf.Finish()
```

Example: Java application

```
env = Environment()
classfiles = env.Java('classes', 'src')
env.JavaH('outdir', classfiles)
env.Jar('myapp', 'classes')
```

Example: Fetching files from CVS

```
env = Environment()
cvs = env.CVS('/usr/local/cvsroot',
              'module')
env.SourceCode('.', cvs)
env.Program('foo',
            Split('file1.c file2.c'))
```

Example: Variant build

```
env = Environment(LIBS = 'c')
ccflags = '-O'
SConscript('src/SConscript', build_dir='opt'
           exports="env ccflags")
ccflags = '-g'
SConscript('src/SConscript', build_dir='debug',
           exports="env ccflags")
```

```
Import("env ccflags")
src = Split('main.c file1.c file2.c')
env.Program('foo', src, CCFLAGS = ccflags)
```

SCons: Design principles

- Correctness
  - Default behavior is a *correct build*
- Performance
  - Options allow you to speed up things by sacrificing correctness in unusual end-cases
- Convenience
  - Dead-simple installation
  - Tools and things work out of the box
  - Easy to configure desired behavior

Strengths of SCons

- Code is regression tested before release
- Good User Manual (60+ pages)
- Dependency checking produces reliable builds
- Responsive development team
- Suports Qt
  - Moc
  - uic

Weaknesses of SCons

- Startup takes ~8 seconds on a 2.6 Ghz workstation
  - Interpreted language
  - dynamic dependency checking
- Syntax errors in SConscript files can trigger Python stack backtraces

SCons: Team

- Steven Knight (project leader)
- Anthony Roach (backup project leader, task engine)
- Charles Crain (Node subsystem)
- Chad Austin
- Steve Leblanc
- Greg Spencer (Visual Studio support)
- Christoph Wiedemann (SConf subsystem)
- Gary Oberbrunner

SCons: Reference projects

- NEWAGE AVK SEG
- National Instruments
- Bombyx
- AI Loom
- Sphere
- Aerosonde
- Computational Crystallography Toolbox (cctbx)
- Evans & Sutherland
- Cheesetracker
- SCons

See the SCons web site for testimonials and details

Acknowledgements

- Steven Knight
  - team leader for Scons
  - supplied some of these slides & graphics
- Gary Granger – built production build environment for project with 40 directories for NCAR/ATD/RTF.

More Information?

- http://www.scons.org
- http://sourceforge.net/projects/scons/
- scons-announce@lists.sourceforge.net
- scons-users@lists.sourceforge.net
- scons-devel@lists.sourceforge.net
- Distributed C/C++ compiler: http://distcc.samba.org