SCONS:

PLATTFORMÜBERGREIFENDES "MAKE" AUF PYTHON- BASIS

Reinhard Wobst
UNIX- Stammtisch Dresden 5.10.05

1. Ärger mit 'make'

- Abhängigkeiten (*.h) vergessen oder inzwischen geändert (bei jeder Umstrukturierung des Quelltextes!)
- Neu-Übersetzen mit anderen Flags: Meist vorsichtshalber `make clean; make` --> oft sehr viel Zeit verschwendet
- Zeitdifferenzen bei NFS ärgerlich (!Portierung auf andere Umgebungen, z.B. bei mir)
- Objektfiles bei `make clean` vergessen, werden wieder eingebunden, obwohl Compileflags geändert --> endlose Fehlersuche
- Plattformwechsel: grausiger Ärger, oft extra Makefile --> parallele Pflege, Fehleranfälligkeit wächst stark mit Größe des Projekts

2. Auswege

- **cmake:** plattformübergreifend, aber Charme der 70er Jahre, fürchterliche Syntax, unübersichtlich, stösst bei komplexen Projekten an die Grenze
- **ant:** Wer Java kann; XML ist ein netter Ansatz, aber feste Syntax - Plattformabhängigkeit muss von Hand codiert werden (eigene Zweige)
- **jam:** http://www.perforce.com/jam/jam.html; soll eigenartige Syntax haben, Einzelheiten kenne ich noch nicht
- **cons:** vielversprechender Ansatz, plattformübergreifend, in Perl geschrieben, aber seit 2001 tot.

- **scons:** komplettes Redesign von cons, entworfen von Steven Knight , in Python geschrieben;
  einfach zu erweitern, sehr universell, modular

  Wichtige Personen (Entwickler, Mailingliste):

  - Steven Knight (Chief developer)
  - Anthony Roach („Co-Entwickler“)
  - Gary Oberbrunner (in Mailingliste sehr aktiv)
  - Chris Cogdon (ebenfalls in Mailingliste sehr aktiv)

  Bezugsquelle: www.scons.org, etwa 100 Downloads pro Tag
- ...?

3. BEISPIEL

Im obersten Verzeichnis muss File Sconstruct stehen (anderer Name ist möglich), ist normales Python-Skript mit „Bibliotheksfunktionen“.

Inhalt eines C-Programms:

```
t.c = "hello world"
```

Sconstruct besteht aus nur einer Zeile:

```
Program('hw', 't.c')
```

Ausgabe unter Posix (= nicht-M$:-):

```
$ scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
gcc -c -o t.o t.c
gcc -o hw t.o
```

```
scons: done building targets.
```

Weniger geschwätzig:

```
scons -Q       # auch: export SCONSFLAGS="-Q"
gcc -c -o t.o t.c
gcc -o hw t.o
```

!Räumt selbständig wieder auf:

```
scons -c -Q
Removed t.o
Removed hw
```

Ausgabe unter Windows (MinGW):

```
scons -Q
gcc -c -o t.o t.c
g++ -o hw.exe t.o

scons -c -Q
removed t.o
removed hw.exe
```

➡ erkennt normalerweise selbstständig Platform und vorhandene Software, bei "flat directories" nur simple Zeile nötig.

Allgemein: Liste als Argument, kann beliebig weit getrieben werden:

```
Program('hello',
    ['main.c', 'atti.c', 'axel.c'] +
    ['mylib.so', 'libthread.so'] +
```

```
    ['joerg.o']
    )
```

4. Was kann scons alles?

Systeme:

Posix, Win32, Cygwin, SunOS, Irix, HP-UX, AIX, Darwin (Mac), OS2

"Compiler":

386asm, aixc++, aixcc, aixf77, aixlink, ar, as, bcc32, c++, cc, cvf, dmd, dvipdf, dvips, f77, f90, f95, fortran, g++, g77, gas, gcc, gnulink, gs, hpc++, hpcc, hplink, icc, icl, ifl, ifort, ilink, ilink32, jar, javac, javah, latex, lex, link, linkloc, m4, masm, midl, mingw, mslib, mslink, msvc, msvs, nasm, pdflatex, pdftex, qt, rmic, sgiar, sgic++, sgicc, sgilink, sunar, sunc++, suncc, sunlink, swig, tar, tex, tlib, yacc, zip

Sprachen:

Assembler, C, C++, D, Fortran (F77, F90, F95), "SWIG", lex, yacc, tar/zip, TeX, m4, Java, ... (wer kennt "D"?)

Suffixes for sources:

.asm, .ASM, .c, .C, .cc, .cpp, .cxx, .cxx, .c++, .C++, .d, .f, .F, .for, .FOR, .fpp, .FPP, .s, .S, .spp, .SPP

Zitate

"Once more I have to say that I really like SCons. Being a python-newbie I managed to write some extensions that allow the usage of SCons for our Make-driven projects nearly out of the box. The first of our projects I have converted to SCons consists of 750 '#include'-Files in about 100 include-directories and 500 C/C++ sourcefiles in 76 libraries. The

original MAKE-Project-setup was heavily recursive and is now substituted with a single SCONSTRUCT-file of about 120 lines of python-code. This is Great!"

-Thomas Runge, NEWAGE AVK SEG

"We use SCons at work to build a huge, highly componentized project with upwards of 5000 source files. It's hands down the best experience I've ever had for building large projects."

-Greg Falcon, National Instruments

---

5. Wie arbeitet scons?

- Standardmäßig MD5-Hash über:
  - Dateiinhalt
  - Build Environment (enthält auch Compilerflags!)
  - offenbar auch Abhängigkeiten
- statt Zeitstempel (optional auch möglich).
- Files ".sconsign" (lokal zu jedem Skript) enthalten Datenbank (binär).
- Include-Files werden automatisch vom Scanner erkannt (kann man auch selbst schreiben ... #ifdef-Problematik ...) (make depend entfällt)
- Kommandos wie Program(), Object(), StaticLibrary() etc. bauen den Abhängigkeitsbaum auf
- Angabe eines Targets beim Aufruf löst Aktualisierung des Teilbaums unterhalb dieses Knotens aus.
- **`scons -c`**, **`scons -c target`**, **`scons -c .`** löscht erzeugte Zwischendateien (**`make clean`** entfällt)
- Sconstruct kann Rufe von SConscript-Files in Unterverzeichnissen enthalten - modulares Vorgehen

- Grundlage: Environment(), enthält alle Abhängigkeiten (Builder), Kommandos, Umgebungsvariable usw.
- Für jede Spezialaufgabe kann man ein neues Environment erzeugen (z.B. völlig andere Compile- und Link-Kommandos, swig, Erzeugen von Programmen mittels Skripte ...)
- Environment ist auch nur Python-Variable, wird mit `Export('env')` exportiert und von SConscript-Files mit `Import('env')` importiert

  Beispiel:

  ```
  from glob import glob
  Import('env')
  obj = env.Object(glob('*.c'))
  Return('obj')
  ```

  ➞ Verzeichnis-Information hängt an 'obj' mit dran, kein Ärger mehr mit Änderungen der Verzeichnisstruktur!
- Eigene Builder-Methoden leicht einzubauen, z.B. XML➞Headerfile
- voller Zugriff auf kompletten Python-Sprachumfang

6. Goodies

- Bei geänderten Compileflags gleich richtige Reaktion: Build Environment ändert sich, damit gebildete Knoten automatisch nicht mehr aktuell!
- Compilieren mit gcc abwechselnd unter Linux und Windows:
  Platform-Variable im Environment automatisch geändert, daher Neucompilierung von allein
- `scons --debug=dtree ccm`:

```
+-.
  +-AES
  | +-AES/aes.o
  +-CCM
  | +-CCM/cbc_ctr.o
  | +-CCM/ccm.o
  +-Test
  | +-Test/ccm_test
  | | +-Test/ccm_test.o
  | | +-AES/aes.o
  | | +-CCM/cbc_ctr.o
  | | +-CCM/ccm.o
  | +-Test/ccm_test.o
  +-test
```

- `scons --debug=explain`: "Warum mache ich jetzt das?"
- Parallelisierbar
- Sogar MVS Projekte unterstützt (Projektfiles, prekompilierte Headerfiles ...)
- Tar/Zip Methoden
- SWIG
- BuildDir() wahlweise mit Hard-/Soft-Link oder Kopieren
- Compilercache auf Wunsch
- Autoconf - hier: `Configure()` (kein voller Ersatz, aber nützlich)
- CVS/Bitkeeper/RCS/SCCS/Perforce
- scons -h = Hilfe:
  `env.Help('my very personal help text')`

7. Dokumentation

- Manpage! Am besten als HTML (auf www.scons.org) - allerdings bei Beta nur `scons.1` aktuell
- 3x kürzer als zsh, 1.5x länger als muttrc

- Leider noch nicht klar genug.
- Scons User Guide (nur für erste Versuche)
- Mailingliste (sehr aktiv, bis zu 20/30 Postings pro Tag), Suche derzeit defekt ➞ googeln oder eigene Suche im lokalen Archiv (sehr groß, 6000 Einträge)
- Wiki auf www.scons.org, noch unfertig (selber helfen - derzeit allerdings read-only wegen Fehler)
- Source (Wahnsinn -- aber gut kommentiert): source_factory, target_factory, emitter …

8. Stolpersteine

- Install: Wenn nicht root, dann PYTHONPATH setzen, MANPATH setzen
- Umgebung wird NICHT importiert - wegen Plattformunabhängigkeit, aber: PATH benötigt, Suse-colorgcc braucht z.B. HOME (crazy error msg)
  Lösung z.B.:
  ```
  env = Environment(ENV = {'HOME' : environ['HOME'], 'PATH' : environ['PATH']})
  ```
- Aktion an Abhängigkeit binden immer etwas schwierig (Dokumentation!). Möglichkeiten:
  - `Command(target, source, command)`
  - Builder(): generisch, aber aufwändiger:
    - Python-Funktion `action(target, srclist, environ)` schreiben
    - Builder aus `action()` erzeugen
    - mit Builder environment erzeugen
  - Grundprinzip von Python: **KISS** (Keep it simple, stupid). Einfache Pythonfunktion ist oft besser wartbar für Fremde (z.B. man selbst nach 3 Monaten).
- schwierig: „Zielfiles“ werden erst während des Builds erzeugt (krasses Beispiel: Doxygen); Auswege:

- SideEffect(...)
- Depends(...)

- Stolperstein für Anfänger: Build-Reihenfolge *nicht* festgelegt!
- Fehlermeldungen oft kompliziert zu interpretieren (Python-Stack, sehr nett, aber Code schwierig)
- Python ist typ-streng, z.B. `int != string`, keine Addition Liste + String usw.

9. Tricks

- Einfache String- und Listenverarbeitung unter Python extrem hilfreich:
  Listen addierbar, z.B. kann Source- oder Objektfileliste in einem "if"-Zweig simpelst erweitert werden:

```
srclst = ['main.c', 'Fit.c', 'Wasser.c']

if sauber:
    srclst += ['entkalk.o', 'libspuel.o']
if extra_schoen:
    srclst += ['duft.o']

Program('spuelmaschine', srclst)
```

  Oder: Compileoptionen, eine pro Zeile:

```
cpu_cflag = """ -mmcu=opteron286
        -Os
        -Wall
        -std=gnu99
        -DF_CPU=1000000UL
        """
cpu_ldflag = """-mmcu=opteron286
        -Wl,-Nice-printf
        -Wl,-UV-scanf
        -lprintf_tinte
```

```
            -Wl,-uextmem_neverinit
            -lscanf_fleet
            -Wl,-Tdata,0x307100
            -L%s/MuttiApp/build
            -Wl,-rpath,%s/MuttiApp/build
            -lm
            -s
            """ % (fiho, fiho) \
              #fiho = find-home, very long string

      ccflags = cpu_cflag.replace('\n', ' ')
      ldflags = cpu_ldflag.replace('\n', ' ')
```

- Modifikationszeit in C-File eintragen:

  ```
  env['CCCOM'] = 'setd $SOURCE\n%s' % env['CCCOM']
  ```

  (Nachteil: übersetzt zweimal wegen MD5, hier aber nicht kritisch)

- Builder sind günstig, wenn viele Files im Spiel sind (`*.in` per `transfrm` in `*.out` verwandeln), aber Overhead im Programm

10. Weitere Features zum selber forschen

- Actions:

  Action - Execute, AddPre/Post-Action, Alias, BuildDir, Builder, CacheDir, Clean, Command, Configure (!), Default, Copy/Chmod/Delete/Mkdir/Move/Touch (portable!) - auch in shutils (kann einfacher sein); Depends, Import/Export/Return, Help, Platform, Scanner (for dependencies), SConscript, SConsignFile, SideEffect, SourceCode(entries, builder), SourceSignatures(type) (MD5/timestamp)

- Variables:
  - **SConscript Variables** u.a.: ARGLIST, ARGUMENTS!

- **Construction arguments**: AS=assembler, CC, CCFLAGS, CPPFLAGS, CPPPATH, JARFLAGS, LIBPATH, LIBS, LINKFLAGS, SHELL, SWIG, TARFLAGS, ...
- **BUILDERS:**

```
env = Environment()
env['BUILDERS]['NewBuilder'] = foo
```

- **generator**: function returns a list of actions to build targets from sources
- **emitter:** Target/Source-Listen verändern, bevor die Targets gebildet werden (damit kann man u.a. Seiteneffekte an scons mitteilen)