The WAF build system

Sebastian Jeltsch

Electronic Vision(s)
Kirchhoff Institute for Physics
Ruprecht-Karls-Universität Heidelberg

31. August 2010
WorkBuildflow
WorkBuildflow

For us:

- low-level code
- many many layers
WorkBuildflow

For us:

- low-level code
- many many layers

make = major pain
WorkBuildflow

For us:

- low-level code
- many many layers
- make = major pain

What we expect from our build system:

- flexibility
  - integration of existing workflows
  - access to well established libraries
  - extensibility
- power
- usability
GNU Build System

+ few dependencies on user side (shell scripts)
+ generates standard make files
+ widely used
  - platform dependent (bash scripts)
  - autoconf-configure is slow
    Often: $t_{configure} \gg t_{make}$
  - another scripting language
CMake

+ generates standard make files
+ platform independent
+ cross compilation
+ parallel build
  - CMake scripting language
  - file content change detection via fs time stamp

Projects using CMake: Boost, Blender, LLVM, KDE, MySQL, . . .
Why WAF?

- project configuration, building, installation, uninstallation
Why WAF?

- project configuration, building, installation, uninstallation
- packaging & package checks for redistribution
Why WAF?

- project configuration, building, installation, uninstallation
- packaging & package checks for redistribution
- ease of python (WAF comes with batteries) - no need for another language
Why WAF?

- project configuration, building, installation, uninstallation
- packaging & package checks for redistribution
- ease of python (WAF comes with batteries) - no need for another language
- Waf is a 90kb script to execute (no installation required)
Why WAF?

- project configuration, building, installation, uninstallation
- packaging & package checks for redistribution
- ease of python (WAF comes with batteries) - no need for another language
- Waf is a 90kb script to execute (no installation required)
- integrates unit testing into the build flow
Why WAF?

- project configuration, building, installation, uninstallation
- packaging & package checks for redistribution
- ease of python (WAF comes with batteries) - no need for another language
- Waf is a 90kb script to execute (no installation required)
- integrates unit testing into the build flow
- supports build variants
Why WAF?

- project configuration, building, installation, uninstallation
- packaging & package checks for redistribution
- ease of python (WAF comes with batteries) - no need for another language
- Waf is a 90kb script to execute (no installation required)
- integrates unit testing into the build flow
- supports build variants
- Good documentation & active development
Why WAF?

- project configuration, building, installation, uninstallation
- packaging & package checks for redistribution
- ease of python (WAF comes with batteries) - no need for another language
- Waf is a 90kb script to execute (no installation required)
- integrates unit testing into the build flow
- supports build variants
- Good documentation & active development
- Fast and small memory footprint
  - as fast as make and 15x faster than SCons
  - 10x less function calls than SCons
Samba 4

- Build time 5min $\Rightarrow$ 35s
Samba 4

- Build time 5min $\Rightarrow$ 35s
- Build size reduction
  - check object file duplication
  - extensive shared-object and rpath use
Samba 4

- Build time 5min ⇒ 35s
- Build size reduction
  - check object file duplication
  - extensive shared-object and rpath use
- full dependency checks
Build time 5min ⇒ 35s
Build size reduction
- check object file duplication
- extensive shared-object and rpath use
full dependency checks
cleaner build rules
#!/usr/bin/env python
#
APPNAME="CodeJam4_WAF_pres"

top='.'

def configure(context):
    context.check_tool("tex")

def build(context):
    context.new_task_gen(
        features = "tex",
        source   = "main.tex",
    )
basic structure

#!/usr/bin/env python
APPNAME='basic_structure'
VERSION='0.1'
top='.'

def configure(context):
    pass

def build(context):
    pass
Installation

no installation needed

Interpreter: installed version will not run on Python 3 yet

OS: platform independence

Admin: installation is cumbersome, and requires admin privileges

Versions: avoid version conflicts (too old, too new, bugs)

Size: the WAF file is small enough to be redistributed (about 90kB)
def configure(context):
    from Configure import ConfigurationError
    try:
        context.find_program(["touch", 'ls'],
                               mandatory=True)
        context.find_program('echo', var='ECHO',
                              mandatory=True)
    except ConfigurationError:
        context.check_message_2("programs not found")

    print context.env['ECHO']

    # execute custom tool
    context.check_tool('my_tool', tooldir='\')
Option Parser

```python
def set_options(context):
    context.add_option('--foo', action='store',
                       default=False, help='Silly test')

# c++ compiler path
opt.tool_options('compiler_cxx')

# python interpreter path
opt.tool_options('python')

def configure(context):
    import Options
    print('the value of foo is %r' % Options.options.foo)
```

- easy to add options
- values are stored in the context variable
Task System

```python
def build(context):

    commands: build, clean, install and uninstall call build()
⇒ isolate targets from actual code
```

Execution control: targets are evaluated lazily

Parallel: task scheduling

FS abstraction: e.g. distributed build

Language abstraction: flexibility and extensibility

Shell abstraction: platform independence
Task Abstraction Layer

abstraction layer between code execution (task) and declaration (task generators):

- **Task:**
  - abstract transformation unit
  - sequential constraints
  - require scheduler for parallel execution

- **Task generator:**
  - factory tasks creation
  - Handle global constraints (across tasks)
    - configuration set access
    - data sharing
    - OS abstraction
#!/usr/bin/env python
APPNAME='example2a'       # Task Generator
VERSION='0.1337'

build_rule='gcc ${SRC} -o ${TGT}'

import TaskGen
TaskGen.declare_chain(
    rule    = build_rule,
    ext_in  = '.c',
    ext_out = '',
    reentrant = False)

def configure(context): pass

def build(context):
    context(source='t0.c', target='t0', rule=build_rule)
    context.new_task_gen(source='t1.c',
                         target='t1', rule=build_rule)
    context(source='t2.c')
c/c++ support routines

#!/usr/bin/env python
APPNAME='example2b' # Task Generator
VERSION='0.1337'

def set_options(context):
    context.tool_options('compiler_cc')

def configure(context):
    context.check_tool('compiler_cc')

def build(context):
    context(target='t', source='t.c', features='cc cprogram')
example4: demo

#!/usr/bin/env python
APPNAME='example4'  # shell usage & task translation
VERSION='0.1337'

def configure(context):  pass

def build(bld):
    bld(rule='cp ${SRC} ${TGT}', source='wscript',
        target='f1.txt', shell=False)
    bld(rule='cp ${SRC} ${TGT}', source='wscript',
        target='f2.txt', shell=True)

    # commands containing '>', '<' or '&' can not be executed
    # => FALLBACK: shell usage
    bld(rule='cat ${SRC} > ${TGT}', source='wscript',
        target='f3.txt', shell=False)
def build(context):
    context.root # root (/) node
    context.path  # current (.) node

    etc = context.root.find_dir(’/etc’)  
    fstab = context.root.find_resource(’/etc/fstab’)  
    context.root.ant_glob(’etc/**/g*’, dir=True,  
                          src=False, bld=False)