Software Build Orchestration with Worch

Brett Viren

Physics Department

BROOKHAVEN NATIONAL LABORATORY

FIFE Workshop, June 2014

Outline

Overview of worch

Using Worch for Release Management

Summary

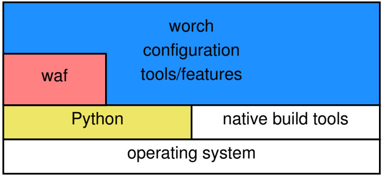
worch = waf + orchestration

A system for "orchestrating" installation of software suites. 3 parts:

waf: A cross-platform, Python program that **executes commands satisfying expressed dependencies**. Think "`make`" but with a real programming language.

tools: Worch **extends waf** with Python code interpreting the **config** to exercises common native package build systems. Users may provide their own tool extensions.

config: Worch adds a **purely declarative** configuration language to describe the tasks to be performed, assert file system layout and build policies, and package versions.



waf in one slide

- Batteries included, Pure Python (supports: 2.4 – 3.4), single-file, cross-platform, user-extensible executable.
- Built-in support for popular, compilers, toolkits and build methods. (GCC/CLANG, TeX, Qt, Boost, SWIG, and more)
- High-performance, tested on large code bases, build profiling, unit tests, build groups, fine-grained content-based dependencies.
- Full expressive power of Python (death to `Makefile`s!!!)
  - But still fairly simple; eg, building this LaTeXpresentation is essentially:

```
$ waf configure

$ waf


$ evince build/worch.pdf
```

```
def configure(cfg):
  cfg.load("tex")

def build(bld):
    bld(features = "tex", source = "worch.tex",
        type = "pdflatex", outs = "pdf")
```

High-level waf "features" map to detailed Python code to generate waf "**tasks**", executed in parallel respecting any dependencies.

worch Layers

- **waf** is low-level and fully general
- *worch* is a **vehicle** for specific policies/conventions
  - "batteries included" tools/features tend to impose policies
  - file-system conventions mostly exposed to configuration layer

Examples of existing policies and conventions:

FNAL/UPS build of LArSoft producing "Fermilab standard" UPS products area. Makes use of Fermilab build scripts (Lynn Garren).

UPS-free build of `art` using new, low-level CMake build for art-like packages. Can still result in "Fermilab standard" UPS products (Ben Morgan, in development)

EM Environment Modules managed binaries, g4lbne (bv)

Nox A Nix-like code aggregation system providing file-system instead of environment variable. based package aggregation (bv, experimental).

Multiple policies/conventions may overlap in the same products area

Procedural Configuration Considered Harmful

Or, don't hand a baby a katana

waf is very well designed, layered, extensible, etc, **but**:

- Python is exceedingly powerful for a configuration language.
- All the more power to get one into deep trouble.
- You've seen crazy `Makefile`s,
  - → Now, imagine their authors high on Python.

⟹ **worch** puts a layer of simple, purely-declarative configuration language on top of Python's power.

- hide the power, but still allow for it when needed
- abstract out common patterns and parameterize them
  - provide reasonable, overridable default parameters

This results in a simple, high-level description of all details related to building the software suite.

Worch Configuration Language

basic syntax text-based schema, named sections of key/value pairs (a.k.a. "INI", a.k.a. Python `ConfigParser`)

extensions hierarchical data representation, variable expansion with inter-section reference and file inclusion

conventions "**group**s" of "**package**s" defining parameters for **waf** "**feature**s" that define **waf** "**steps**".

Configuration allows:

- defining software suite packages and their versions
- determine per-package installation procedures to apply
  → and their detailed parameters
- asserting layout policies for intermediate and final files

Still exposes a lot of power. There is no magic and configuration authors still have to think.

Get a Flavor for a Worch Config File

Fake snippet showing some features of the configuration language:

```
[start]
groups = buildtools, gnuprograms, mystuff
features = tarball, autoconf, makemake
install_dir = {PREFIX}/{package}/{version}

[group gnuprograms]
packages = hello
source_url = http://ftp.gnu.org/gnu/{package}/{source_package}

[package hello]
version = 2.8
features = tarball, patch, autoconf, makemake

[package myapp]
version = 1.0
features = tarball, cmake, makemake
```

Default `{variables}` can be set generally and overridden locally.
Config groups translate to atomic groups of waf tasks.

Waf “Features”

A **waf**-technical term: *features*.

- Named chunks of parameterized Python code that generate **waf** tasks to do something.
- Features tend to be written to work in concert.
  - Tasks are linked by the files the produce/consume.
  - Worch features follow naming convention: download, unpack, prepare, patch, build, install
- **waf** comes with low-level features such as those that produce C/C++/FOTRAN compiler tasks.
- *worch* adds high level features:

tarball	download and unpack a source tarfile
vcs	same but from git/hg/svn/cvs
patch	download and apply some patch to the source
autoconf	configure source with GNU autoconf
cmake	same but with CMake
makemake	run `make/make install`

Other *worch* additions

logging step-specific log file holding command line, working directory, full environment, internal worch/waf state, and `stdout/stderr`

fail script step failure generates a script to reproduce the failure with CWD, ENV, and cmd line set.

controls each step produces a conventionally named file on success: dependency linkage and forced-redo of a step.

fail early/often *worch* tasks are written to succeed or else fail vociferously. Goal is to have no silent false-successes.

Tools Provide *Features*

A **waf**-technical term: *tools*.

- Python modules following **waf** +*worch* conventions to define *features* with access to full data structure from parsed configuration files.
- Loaded via *worch* configuration file directives.
- Provides user-extension of **waf**/*worch*.
- Tool code may be stored with worch configuration files.
  - → version control the two together, or
  - → general purpose tools rolled back into *worch* for wider benefit

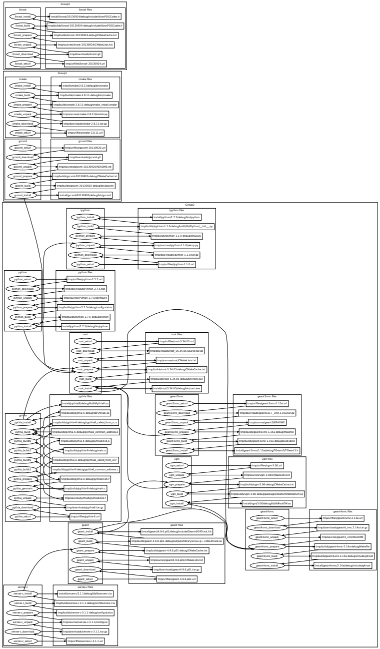
LBNE has special worch tools for building the LArSoft/*art* suite:

- via Fermilab scripts.
- via UPS-free CMake for *art*-type packages, (in development).

Example Suite

1. cmake
2. gccxml
3. geant
4. geant3vmc
5. geant4vmc
6. ilcroot
7. pythia
8. python
9. root
10. vgm
11. xerces-c

Relatively small project. Figure shows build steps and their dependencies.


Group3
ilcroot
ilcroot_install
ilcroot_build
ilcroot_prepare
ilcroot_unpack
ilcroot_download
ilcroot_seturl
ilcroot files
install/ilcroot/20130924/debug/include/IlcIonPDGCodes.h
tmp/builds/ilcroot-20130924-debug/include/IlcIonPDGCodes.h
tmp/builds/ilcroot-20130924-debug/CMakeCache.txt
tmp/sources/ilcroot-20130924/CMakeLists.txt
tmp/downloads/ilcroot.git
tmp/urlfiles/ilcroot-20130924.url
Group1
cmake
cmake_install
cmake_build
cmake_prepare
cmake_unpack
cmake_download
cmake_seturl
cmake files
install/cmake/2.8.11/debug/bin/cmake
tmp/builds/cmake-2.8.11-debug/bin/cmake
tmp/builds/cmake-2.8.11-debug/cmake_install.cmake
tmp/sources/cmake-2.8.11/bootstrap
tmp/downloads/cmake-2.8.11.tar.gz
tmp/urlfiles/cmake-2.8.11.url
gccxml
gccxml_seturl
gccxml_download
gccxml_unpack
gccxml_prepare
gccxml_build
gccxml_install
gccxml files
tmp/urlfiles/gccxml-20130920.url
tmp/downloads/gccxml.git
tmp/sources/gccxml-20130920/README.rst
tmp/builds/gccxml-20130920-debug/CMakeCache.txt
tmp/builds/gccxml-20130920-debug/bin/gccxml
install/gccxml/20130920/debug/bin/gccxml
Group2
ipython
ipython_install
ipython_build
ipython_prepare
ipython_unpack
ipython_download
ipython_seturl
ipython files
install/python/2.7.5/debug/bin/ipython
tmp/builds/ipython-1.1.0-debug/build/lib/IPython/__init__.py
tmp/builds/ipython-1.1.0-debug/setup.py
tmp/sources/ipython-1.1.0/setup.py
tmp/downloads/ipython-1.1.0.tar.gz
tmp/urlfiles/ipython-1.1.0.url
python
python_seturl
python_download
python_unpack
python_prepare
python_build
python_install
python files
tmp/urlfiles/python-2.7.5.url
tmp/downloads/Python-2.7.5.tgz
tmp/sources/Python-2.7.5/configure
tmp/builds/python-2.7.5-debug/config.status
tmp/builds/python-2.7.5-debug/python
install/python/2.7.5/debug/bin/python
root
root_seturl
root_download
root_unpack
root_prepare
root_build
root_install
root files
tmp/urlfiles/root-5.34.05.url
tmp/downloads/root_v5.34.05.source.tar.gz
tmp/sources/root/CMakeLists.txt
tmp/builds/root-5.34.05-debug/CMakeCache.txt
tmp/builds/root-5.34.05-debug/bin/root.exe
install/root/5.34.05/debug/bin/root.exe
pythia
pythia_install
pythia_build
pythia_buildf2
pythia_buildc2
pythia_buildf1
pythia_buildc1
pythia_prepare
pythia_genmain
pythia_unpack
pythia_download
pythia_seturl
pythia files
install/pythia/6/debug/lib/libPythia6.so
tmp/builds/pythia-6-debug/libPythia6.so
tmp/builds/pythia-6-debug/tpythia6_called_from_cc.o
tmp/builds/pythia-6-debug/pythia6_common_address.o
tmp/builds/pythia-6-debug/pythia6416.o
tmp/builds/pythia-6-debug/main.o
tmp/builds/pythia-6-debug/tpythia6_called_from_cc.F
tmp/builds/pythia-6-debug/pythia6_common_address.c
tmp/builds/pythia-6-debug/pythia6416.f
tmp/builds/pythia-6-debug/main.c
tmp/sources/pythia6/pythia6416.f
tmp/downloads/pythia6.tar.gz
tmp/urlfiles/pythia-6.url
geant3vmc
geant3vmc_seturl
geant3vmc_download
geant3vmc_unpack
geant3vmc_prepare
geant3vmc_build
geant3vmc_install
geant3vmc files
tmp/urlfiles/geant3vmc-1.15a.url
tmp/downloads/geant321+_vmc.1.15a.tar.gz
tmp/sources/geant3/README
tmp/builds/geant3vmc-1.15a-debug/Makefile
tmp/builds/geant3vmc-1.15a-debug/build.done
install/geant3vmc/1.15a/debug/TGeant3/TGeant3.h
vgm
vgm_seturl
vgm_unpack
vgm_prepare
vgm_build
vgm_install
vgm files
tmp/urlfiles/vgm-3.06.url
tmp/sources/vgm-3.06/CMakeLists.txt
tmp/builds/vgm-3.06-debug/CMakeCache.txt
tmp/builds/vgm-3.06-debug/packages/RootGM/libRootGM.so
install/vgm/3.06/debug/lib/libRootGM.so
geant4vmc
geant4vmc_seturl
geant4vmc_download
geant4vmc_unpack
geant4vmc_prepare
geant4vmc_build
geant4vmc_install
geant4vmc files
tmp/urlfiles/geant4vmc-2.14a.url
tmp/downloads/geant4_vmc.2.14a.tar.gz
tmp/sources/geant4_vmc/README
tmp/builds/geant4vmc-2.14a-debug/Makefile
tmp/builds/geant4vmc-2.14a-debug/include/g4root
install/geant4vmc/2.14a/debug/include/g4root
geant
geant_install
geant_build
geant_prepare
geant_unpack
geant_download
geant_seturl
geant files
install/geant/4.9.6.p01/debug/include/Geant4/G4Track.hh
tmp/builds/geant-4.9.6.p01-debug/outputs/library/Linux-g++/libG4track.so
tmp/builds/geant-4.9.6.p01-debug/CMakeCache.txt
tmp/sources/geant4.9.6.p01/CMakeLists.txt
tmp/downloads/geant4.9.6.p01.tar.gz
tmp/urlfiles/geant-4.9.6.p01.url
xerces-c
xerces-c_install
xerces-c_build
xerces-c_prepare
xerces-c_unpack
xerces-c_download
xerces-c_seturl
xerces-c files
install/xerces-c/3.1.1/debug/lib/libxerces-c.la
tmp/builds/xerces-c-3.1.1-debug/src/libxerces-c.la
tmp/builds/xerces-c-3.1.1-debug/config.status
tmp/sources/xerces-c-3.1.1/configure
tmp/downloads/xerces-c-3.1.1.tar.gz
tmp/urlfiles/xerces-c-3.1.1.url

Software Suite Releases

Three ingredients:

1. *worch* config file captures:
   - entire list of packages
   - all their version strings
   - complete build details
2. Keep *worch* config file + any custom waf tools in a repository.
3. Branch/tag this repository during release process.
   - allows various release management schemes
   - track source version and build version separately

→ One tag fully specifies the entire suite's source and build.

→ Can be reproduced for anyone and for all time.

- (given source repositories, etc)

→ Lends itself to a high degree of automation.

Example Software Suite Build

waf (+*worch*) is already highly automated:

```
$ git clone https://github.com/brettviren/worch.git
$ git clone http://myserver.com/myworchcfg.git
$ myworchcfg
$ git checkout my-release-tag
$ cd ../worch
$ waf --prefix=/path/to/install \
      --orch-config=../myworchcfg/main.cfg \
      configure build
```

That last waf command may take hours, depending on software suite size and build environment power, but it runs with no human intervention. At the end, the installation is ready to use.

LBNE Release Automation

LBNE wraps the high-level commands from the previous slide to provide simplification, automation and encapsulation:

```
$ wget https://cdcvs.fnal.gov/.../lbneinst
$ ./lbneinst "larsoft-1.00.02_build-1"
```

That's a tag on the LBNE worch config repository:

`larsoft-1.00.02_build-1`

`larsoft`	suite name
`1.00.02`	suite version
`build-1`	build number

- Complete specification, one command, fully automated install.
- Build number versions build configuration.
  - bumped for bugs in build configuration
  - adding new platforms to existing releases

Role of *worch* in LBNE Continuous Integration

LBNE has active contributors to the CI group and are looking forward to implementing CI client tests!
Planned build tests:

- *worch*-driven, green-field, full stack software builds triggered by release tags on:
  - → LBNE packages
  - → larsoft
  - → art
- incremental rebuilds of LBNE packages triggered by commits to these packages.
  - following some dwell period of no commits (eg, 1 hour)

We will explore populating local site installation areas, as a side effect, with successful CI build results.

Summary

- **waf** provides a platform independent, highly capable build tool.
- *worch* builds on top of waf, more batteries and a simplified configuration layer and provides a solid basis for release management.
- *worch* is used by LBNE to automate building software from source (so far, suites driven by LArSoft and g4lbne).
- I'm looking forward to seeing how *worch* can help a wider community!

Links (clickable):

- worch GitHub and waf GoogleCode
- LBNE worch-based releases
  → so far just larsoft, development on hold for UPS-free CMake
- g4lbne install
  → worch-based, release-management in development