Build Orchestration with worch

Brett Viren

Physics Department

BROOKHAVEN NATIONAL LABORATORY

November 13, 2013

Contents

worch in a nutshell

worch is a **meta**build system that orchestrates the running of native build mechanisms. it consists of these layers:

configuration	a simple, declarative configuration language specifying all information required to build a package.
features	Python methods translating configuration into tasks taking well defined inputs and producing well defined outputs.
waf	the waf engine to sort out the dependencies and execute the tasks in a parallel manner.

worch configuration overview

The configuration...

- partitions the software suite into groups of packages.
- sets the directory layout for intermediate and installed files.
- lists which *feature methods* are used to build a package.
- provides the values of variables required by the features
- sets per-package build and run time environment variables.
- variables be defined in terms of other package and system defined variables.
- may define external worch modules for loading.

The “start” and “keytype” sections

The parsing starts at the `start` section and builds up a hierarchy of information by interpreting any keys found in the `keytype` section as lists of sections of the given type.

```
# this is a comment
[start]
groups = buildtools, compiler, externals, art, larsoft
includes = default.cfg, externals.cfg, art.cfg, lbne.cfg
download_dir = downloads
...
[group buildtools]
packages = cmake, ...
...
[package cmake]
version = ...
...
[keytype]
groups = group
packages = package
```

Any items defined at high levels are copied down to the leafs of the hierarchy. Eg, `download_dir` is available to consumers of the `cmake` package information.

group sections

- List a number of packages in the group
- A place to define variables to apply to all packages in the group.
- A completely built in a serial fashion
  - → packages within a group are built in parallel

```
[group gnuprograms]
packages = hello, bc
source_archive = {package}-{version}.tar.gz
source_url = http://ftp.gnu.org/gnu/{package}/{source_archive}
unpacked_target = configure

[package hello]
version = 2.8
```

Shows example of using variable reference and defining variables that may apply to all packages in the group, exploiting symmetry that exists among GNU packages.

package sections

- List the `features` that implement the installation
- List any variables expected by the features
    - Provide any not given higher up (in `group` or `start`).
    - Override any defaults given higher up with needed specialization.

```
[package hello]
version = 2.8
features = tarball, autoconf, makemake
build_target = src/hello
install_target = bin/hello
```

dependencies

Three types of dependencies can be expressed:

file	declaring fail to be output by one step and input by another
explicit	any step can be declared dependent on another by name ("package_step" naming convention)
environment	any package may declare its dependency on another packages exported environment variables. These will be defined in the calling environment for all steps

```
[package foo]
features = tarball, autoconf
# may be used as output to tarball, input to autoconf
unpacked_target = configure
depends = prepare:bar_install
environment = group:compiler, package:cmake
```

more dependencies

worch enforces a standard set of file-based dependencies which may be used to glue features together. Every successful package step produces a `package_step` file.

```
my_unpack = tgen.control_node('unpack')
root_install = tgen.control_node('install', 'root')
```

These can be used, for example, to make sure a package is only compiled after it's been unpacked and ROOT has been installed.
For steps defined all in one feature, the "control node" need not be used.

external methods

worch comes with batteries-included for common native build mechanisms. Support for novel ones can be added as waf `tools`.

```
[package foo]
features = tarball, fooinst
tools = foo.tool, bar.tool
```

All python modules listed by `tools` will be loaded using waf's tool `load()` mechanism. This may be used to define novel features. Here `foo.tool` may provide feature `fooinst`.

feature methods

Duties

- define defaults configuration values
- produce waf tasks via a worch-provided interface

Extension on waf features.

```
import orch.features
orch.features.register_defaults(
  "fooinst", foo_dir = "{PREFIX}/foo",)
from waflib.TaskGen import feature
@feature('modulesfile')
def feature_modulesfile(tgen):
  tgen.step("foostep1", rule=..., source=...)
  tgen.step("foostep2", rule=..., target=...)
```

- May define default values for configuration items or use existing ones.
- The `tgen` is an augmented waf `TaskGen`, mostly the `step()` method is used.

using worch for an installation

```
$ waf --prefix=/path/to/install \
      --orch-config=suite.cfg \
        configure build install
```

worch comes with a copy of waf.

worch bundles

A mechanism to pack waf, worch Python modules, external feature methods, and the suite's configuration files into a self-extracting Python program.
Cartoon calling:

```
$ wget http://some.server.gov/lbne-software-rX.Y.Z
$ ./lbne-software-rX.Y.Z configure build install
```

Batteries included (so far)

tarball	download tar/zip, unpack
vcs	same but for git/hg/svn/cvs
patch	download and apply a patch file
autoconf	run Autoconf's configure script to prepare the source
cmake	same but run `cmake`
makemake	`make/make install` doublet
modulesfile	produce a `http://modules.sf.net` modules file for environment setup
upspkg	same but for UPS
pypackage	install a Python package via its `setup.py`
special	some special-purpose package installation (tbb, pythia6)

Note, some of these "batteries" have been turned into external tools/features.

Near future plans for worch

- Merge the external tool/features support into `master`
- Produce external tool/features to implement Lynn's high-level from-source instructions
- Add a feature to allow bulk of configuration and external tools to be specified by URL and downloaded (to assist with release management)
- Continue to work with Sebastien Binet with ATLAS adoption of worch and investigate his `hwaf` tool to simplify user-level activities.

worch development at:

https://github.com/brettviren/worch