Software Packaging for High-energy Physics

Brett Viren

May 19, 2015

Contents

1 Overview

This note attempts to break down the issues and problems related to software "packaging" in high-energy physics. The term "packaging" is interpreted broadly to include aspect of developing, releasing, distributing, installing and ultimately using software in high-energy physics.

It first defines a number of roles that individuals fill (sometimes simultaneously) in their relationship with HEP software. It then describes some requirements for systems to assist each role in making their actions more efficient.

Much of this note consists of opinion based on experience in these issues in the context of Intensity Frontier experiments (specifically MINOS, Daya Bay, DUNE) and smaller software projects.

2 Roles

These distinct roles are considered in defining the problem.

user follows given and possibly site-specific instructions to configure an environment in order to run provided executable programs or develop high-level analysis code (eg, ROOT plots) against previously installed packages.

physicist-programmer develops relatively high-level software elements but ones that are used by others. They develop in the context of an experiment's software without being expected to understand its full breadth and depth. Eg, they may provide modules in an event processing framework, they may add features to some core component, they would use whatever build system the experiment adopts but not be expected to understand it or develop it. They tend to need to develop software in a context which is similar and maybe exactly the same as a *user*.

core-developer develops the experiment's over-aching software systems (eg, frameworks, build systems, core services). Is expected to understand the breadth and depth of the system. Can add new elements or expand the overall structure.

builder builds the experiment software suite from source either for in-place use at a site or to produce binary packages for local or distributed installation

installer manages the installation of provided binary packages at a site.

release manager determines the versions and build configuration of the experiment's software suite.

3 Tools

To support each of these roles, a set of tools are needed. In general, these tools should automate tasks and codify policy.

3.1 End-user, Run-time Environment Management

A *user* needs a way to configure their run-time management. A system is needed which:

- provides an end-user run-time environment which allows a user to execute an experiment's software suite and develop high-level software against it.
- aggregates specific versions of packages in the experiment's software suite into a working whole in a reproducible manner.
- allows multiple, distinct aggregations by one user.
- hides the complexity of the aggregation, requiring the user to know a single, brief "suite identifier".

There are two basic types of end-user, run-time environment management which differ based on their aggregation mechanism.

shell environment variables (EV) software is installed to locations distinct to the package and its version (eg, a "package/version" tree: `/path/to/<package>/<version>/{lib,bin,include}/`) and a number of the user's shell environment variables (eg `PATH`) are modified in order that these areas take precedence over system corresponding locations.

file-system (FS) software is either installed in a single-root (eg `/path/to/{lib,bin,include}/`) or is installed into a package/version tree and a single-root is emulated through production of symbolic links or file copying. The shell environment variables are still modified as above but only a single entry is required instead of one entry per package.

The two approaches have the following trade-offs:

- EV causes user shell environment variable bloat which can become confusing. FS requires a "link farm" which can also be confusing. Both require tools to manage this complexity.
- EV requires knowing both the "suite identifier" and detailed knowledge of how the environment variables are set in order to understand what software was run (or a `/bin/env` dump must be run). FS leaves an indelible record on the file system.
- Building software in an EV aggregation requires fine-grained build configuration (eg `"./configure --with-XXX=YYY"`). FS's single-root hides this complexity.

- EV aggregation can be broken (intentionally or accidentally) by end-users. FS aggregation may exist in read-only file system space.

Besides the simpler view of a suite, some other benefits of FS aggregation which are not easy to implement using EV aggregation include usage analytics to determine who is currently requiring a given package, indirection for providing statically labeled version (eg, "pro" vs. "dev"), upgrade and roll-back.

Some existing EV aggregators:

Configuration Management Tool (CMT) custom, simple configuration language, inter-package dependencies, multiple installation bases, Free Software, widely used by CERN and some other experiments.

Environment Modules (EM) TCL configuration language, inter-package dependencies, multiple installation bases, Free Software, used fairly widely in and outside of HEP.

Unix Product Support (UPS) custom, difficult to use configuration language (objective statement based on observation of how even experts use it), inter-package dependencies, multiple installation bases, used at Fermilab (variant EUPS used by LSST).

Some existing FS aggregators:

NixOS/ GUIX two Linux distributions with common design (different configuration languages) which allow for aggregating an entire OS of packages. They require a universally shared mount root directory in order for `RPATH`, etc to resolve correctly. They provide predictable, reproducible builds allows some level of trust of binary packages built by peers.

Conda similar idea but with a local root. Developed in a Python ecosystem but can handle arbitrary binary packages. Affiliated with Binstar which provide free package hosting.

Nox an initial attempt by this author to provide a Nix-like tool while relaxing some of its more stringent requirements.

It should be noted that the two approaches to aggregation, EV and FS, are not mutually exclusive. A UPS "products" area can also serve as a EM "modules" area while also providing a source of packages that are aggregated through some FS-based aggregation tool.

3.2 Development build tool

A physics-programmer needs to develop a module in the context of the overall experiment software suit in a way that provides:

- reduced overhead inside the edit-compile-test-install loop.
- ability to run both unit and integration tests, ideally with no overhead.
- dependency management to allow building of those experiment software packages which depend on their package.
- automated rebuild all of just code with modified dependencies.
- ability to ease the development of both new and existing packages.
- ability to work on branches in the code repository and share the intermediate development with others.
- ability to maintain multiple, independent lines of development in the same user account.

Some existing development build tools

- CMT
- mrb

3.3 Build automation

All roles (except maybe *user*) need a way to simply build the software. Some of the desired features of the build automation:

- Build from source the shared libraries and other artifacts
- Work in a highly automated fashion, ideally based on running a single command parameterized by a high-level suite version identifier
- Build from a *green field* needing only minimal and well characterized pre-requisites.
- Build incrementally to reuse past products in order to save time
- Be easy to port to new platforms.

Some existing build automation systems

- CMT + LCGCMT + high level scripting
- Worch
- homebrew/linuxbrew
- guix/nix

3.4 Release management

A release management system must provide a mechanism to:

- precisely specify experiment software versions, build configuration options and likewise for all software packages on which the experiment relies.
- support an overall version identifier of this specification
- disseminate this information so that it can be replicated by others

Some existing release management systems

- CMT's set of requirements files
- Worch's configuration file set

4 Some words about Worch

Worch is a general purpose build orchestration system with "batteries included" to support many common build methods. It uses "Waf" to provide a kernel of dependency management and idempotent task scheduling.

Worch's main value is in the simple text-based configuration language it adds as a user interface to producing Waf tasks. This configuration language allows one to concisely express all specifically pertinent information about how each package in a suite is built. There is one configuration section per package and it provides information on the build methods to apply, the location of initial source code and any non-default parameter values. The language supports simple macros which allows for better management of the information (eg, through DRY strategy).

The configuration language is parsed by Worch and then interpreted by a number of Waf *tools* which produce Waf *tasks* (written in Python). Worch comes with a number of tools which provide common tasks to perform build procedures such as:

- download source as tar archives or from VCS such as git/svn/cvs/hg
- prepare source with Autoconf's `configure` or CMake
- run usual `make` and `make install`
- create simple binary `tar`-based packages or (with `worch-ups`) binary UPS "product" packages

While Worch comes with some common "batteries" it is general purpose. Any system which requires interdependent tasks to be performed can be orchestrated and automated with Worch. Tasks which are driven in a parameterized way are particularly suitable for being automated with Worch.

By capturing and tagging the configuration file in a code repository one has a simple mechanism to provide solid release management.