THE HEP SOFTWARE FOUNDATION (HSF)

HSF-TN-2016-03
January 12, 2017

HSF Packaging Working Group Report

L. Sexton-Kennedy[1], B. Hegner[2], B. Viren[3], G. Amadio[4]

[1]*FNAL,*[2]*CERN,*[3]*BNL,*[4]*UNESP*

Abstract

The note describes the outcome of the discussions in the HSF Packaging Working Group. It summarizes the discussion on existing configuration and build tools and the possibility to converge on more common solutions.

1 Introduction

Software development in high energy physics follows the paradigm of open-source software (OSS). Experiments as well as the theory community heavily rely on software being developed outside of the field. The number of such third party software (so-called *externals*) used within a given context can easily reach over 100 interdependent software packages.[†] Creating a consistent and working stack out of 100s of packages, on a variety of platforms is a non-trivial task. Within the field multiple technical solutions exist to configure and build those stacks, in the following these will be referred to as *build tools*.

Furthermore, quite often software has to be ported to new platforms and operating systems and subsequently patches to the individual externals need to be created. This is a manual and time consuming task, requiring a very special kind of expert knowledge.

None of this work is experiment specific and our working group agrees that this effort is being duplicated. The aim of the HSF packaging working group is to see whether a better synergy within the field on porting/patching and build tools is possible.

This document describes the discussions and findings of the working group.

1.1 The Working Group

The working group consisted of members of the HEP community from the energy and intensity frontiers as well as accelerator modeling. The group met in a series of meetings over the months of May and June 2015 [1]. Additional discussions took place via the public packaging working group mailing list.

1.2 The Packaging Problem Space

Collectively across HEP and even within some large institutions there is a need to maintain software installations which span a broad configuration space. It is helpful to consider this as an actual multi-dimensional space and decompose it along the following orthogonal and discrete dimensions:

project enumerates a set of software units each of which provides some cohesive functionality, often in source code form and which can be used to produce a number of library or executable files. Examples include ROOT, Geant4 or BOOST.

version labels the state of a project's software either as a formal release or as a snapshot in time. Of course the exact label convention will cosmetically differ between projects but this coordinate can nonetheless be considered ordered in some manner and as a union of all possible conventions. Examples include 6.04/10 (ROOT), 10.01p02 (Geant4), 2015118 (some snapshot) or `0400b69` (abbreviated git commit hash).

[†] A package in this context is a revision controlled collection of source files including files to provide an implementation of how to build the source into binary products. These can either be object libraries or executables.

variant describes the set of options applied during the process of building a particular (project,version) pair. The variant of a package is in general itself multi-dimensional. Between any given set of (project,version) points there is some amount of degeneracy in the variant dimension. For example, it is common now that all C++ packages that must work together must be built with a consistent compiler and C++ standard. However, some packages may be built with optimization and no debug symbols and others, vice versa. Debug symbols and the choice of a particular C++ standard are the most used variants in HEP.

platform denotes the combination of operating system, hardware architecture and chosen compiler and compiler version. Many HEP experiments define their own naming conventions for the *platform* dimension in this space, and they are all equally valid but inhibit sharing due to trivial language differences. A sub-goal of this working group is to agree on a field wide naming convention, in form of another technical note [2].

dependencies of a particular build of a project can be of different version and variant. The chosen setup of the dependencies are thus another dimension in the configuration space.

For the purpose of this note a single (*project,version,variant,platform,dependencies*) point in this space is called a built *package* (for now, ignoring details of file system layout, bundling, distribution, run-time environment). The software that is provided by the operating system may be thought of as a single point in this space.

In this picture, an end-user *software suite* is represented as a tree of connected points in this space. Connections are made along lines of dependency and such that desired build consistency is secured. It is the act of building one package on top of another that a line is drawn.

This picture allows the expression of some required and desired features of a packaging system:

- require ability to precisely describe, record and instantiate a package-space tree.
- require the ability to produce different package-space trees.
- require the ability for end-users to selectively activate a specific package-space tree.
- require the ability to prune installed package-space trees but keep shared branches needed by any retained trees.
- desire ability to avoid rebuilding common branches when two trees overlap.
- require ability to precisely assemble package-space trees from individual points (built packages) exactly as they were built

- desire ability to loosely assemble package-space trees from individual points close to where they were built (eg, build against Python 2.7.9 but run against Python 2.7.10).
- require ability to derive a novel package-space tree from an existing one (eg, upgrade ROOT to new version, make debug/opt versions of some projects, use native compiler or build one as part of the tree).

2 The HEP build and packaging tool landscape

Software stacks are needed by many organizations most notably HEP experiments. If a project is authored by the organization it is not considered an external project however it still will need build recipes. Any system that builds external projects into a coherent whole must also be capable of building the internal projects as well. In the area of external projects it will be useful to distinguish between "foreign" and "domain" external projects. Domain projects are authored by groups within the field of HEP and therefor the HSF could influence the behavior of the domain projects. The foreign projects have to be dealt with as they exist. Examples of foreign projects include: gcc, fftw3, eigen and lapack. Examples of domain projects include clhep and ROOT. In the following a few of the solutions used within the community are summarized. For more details we refer to the project websites.

2.1 Solution 1 – Worch

Worch [3] is the build tool being proposed for the DUNE collaboration. It has been used to build the software suite common to many liquid argon experiments, called larsoft, as well as the experiment specific code that builds on top of larsoft.

2.2 Solution 2 – LCGCMake

LCGCMake [4] is the build tool used for creating the LCG releases used by the LHC experiments ATLAS and LHCb. It is based on a set of CMake macros and each external project is described using one of these macros in a semi-declarative style.

2.3 Solution 3 – cmsBuild

cmsBuild [5] is the build tool used for making releases of CMSSW for CMS. It orchestrates the building of the whole stack from the compiler on up on an as needed basis for any particular release/architecture combination. It is based on rpm build, python scripts, and text files that specify dependencies, library names and other optional information.

2.4 Solution 4 – Contractor

Contractor [6] is a build tool used in the accelerator modeling community and was developed for the Synergia project. The project must support a wide diversity of platforms from laptops to super-computers. Looking at the requirements of this community and the solutions they chose to meet them should inform the HSF community as we look forward to a time when many more diverse platforms will be need in HEP computing. Contractor has no external dependencies and is implemented in Python.

2.5 Solution 5 – SciSoft (mrb/ups/cetbuildtools)

MRB is an acronym for Multi-Repository-Build and is used by some of the intensity frontier experiments at FNAL. Unlike other tools in this list it requires ups (an acronym for Unix-Product-Support) to create an environment for it to operate in properly. cetbuildtools is a set of scripts that drive CMake to build an interdependent set of packages. The resulting packages are uploaded to a distribution server called SciSoft.

2.6 Solution 6 – aliBuild

aliBuild [7] is an evolution of cmsBuild that fixes many of the weakness of cmsBuild. Relative to the above it is not dependent on rpm and builds tar-files that can be curated into a number of diffent package managers such as yum, and apt.

3 Taxonomy of non-HEP tools

3.1 Solution 7 - Homebrew

Homebrew [8] is a MacOS specific build tool. Packages build instructions are declared by writing Ruby classes with a given set of attributes. The feature set of these declarations is similar to those of cmsBuild and lcgcmake. A project fork focuses on the addition of Linux as a supported platform.

3.2 Solution 8 - Nix

Nix [9] is a cross-platform build tool. Package build instructions are written in the *Nix expression language*.

3.3 Solution 9 - Spack

Spack [10] is a packaging system specifically developed for the HPC use case which has a lot in common with the HEP requirements. “Spack provides a novel, recursive specification syntax to invoke parametric builds of packages and dependencies. It allows any number of builds to coexist on the same system, and it ensures that installed packages can find their

dependencies, regardless of the environment." On creating Spack the authors did a review of existing packaging systems, including a few of those discussed in this document.

3.4 Solution 10 - Portage

Portage [11] is a GPLv2 package management system based on ports collections from BSD-based operating systems. It was originally created for Gentoo Linux, but is also used by ChromeOS and CoreOS, among others. Portage packages, called *ebuilds*, contain build instructions written as bash shell scripts with a special syntax. Portage itself is written in a combination of python and bash. Portage can be used to manage software on Linux and several other Unix variants, including macOS (see [12] for more details).

3.5 Commonalities

All these non-HEP tools rely on properly set RPATHs and do not provide any relocatability options. The targeted deployment scenarios are different ones compared to the current approach in HEP.

4 Tool Features and Requirements

We compared the feature set of the tools described. In evaluating them we defined a set of criteria in which to measure the tools against each other. However, one experiment's strict requirement may just be considered a feature for another experiment. In the following we explain the comparison criteria and the situation for the individual tools.

4.1 Supported Platforms and Environments

The most basic criterion is the support for the various platforms and environments used in high-energy physics. There are three categories of environments:

1. **Linux**
2. **MacOS X**
3. **Windows**

The first category is split into several flavours of distributions. The main distribution are the RedHat derived Scientific Linux [13] and CentOS [14]. Compared to these, Debian [15] and Debian based distributions play a smaller role in computing centres. On desktops, the Debian based Ubuntu [17] seems rather popular.

In addition, multiple hardware architectures are in use or will be in use in the foreseeable future - x86, ARM, PowerPC, MIC, and various dedicated super-computers. Some of them impose the requirement of cross compilation onto the build and packaging solution, which is why the keyword **Xcompiler** is added to the platform table. However this also

	Linux	MacOS X	Windows	Xcompiler
aliBuild	+	+	-	+
cmsBuild	+	+	-	+
Contractor	+	+	-	+
Homebrew	o	+	-	-
LCGCMake	+	+	o	o
Nix	+	+	o	o
SciSoft	+	+	-	-
Spack	+	+	-	+
Worch	+	+	-	o
Portage	+	+	-	+

Table 1: Supported platforms of the different build tools. A yellow "o" denotes that the support is untested or not of production quality.

refers to the ability to build linux releases on mac, or ability to build mac releases on linux. The table 1 summarizes this information for the tools considered by the working group so far. Both Linux and MacOS X are supported by all of the tools, while Windows is not supported at all. X-compilation does not seem a priority of the projects yet.

4.2 Build and Installation Variants

One important feature of build tools in HEP is the support for the installation of multiple stacks or package versions in parallel.

1. **Multi-Rel**: The support for building and installing multiple stacks in parallel. This is important if multiple versions of experiment software need to coexist on a given installation.

2. **Multi-BuildVar**: The support for multiple variants of the same project, release pair within the same stack. There are many possible reasons why this may be needed, what they have in common is the need to specify compiler switches that have to be applied consistently across the build, like a debugging option or a compiler dialect.

3. **MultiShell-RTE**: The support for setting up the runtime environment for a given stack triplet (project,version,variant), supporting multiple shell flavours.

4. **Relocation**: Whether the build tool supports the relocation of packages or creates fully relocatable packages.

The assessment of the tools according to these criteria are listed in Table 2.

4.3 Ease of Install and Use

The criteria in this category are:

	Multi-Rel	Multi-BuildVar	MultiShell-RTE	Relocation
aliBuild	+	+	+	+
cmsBuild	+	+	+	+
Contractor	+	+	NA	-
Homebrew	-	-	NA	-
LCGCMake	+	+	+	+
Nix	+	-	+	-
SciSoft	+	+	+	+
Spack	+	+	NA	-
Worch	+	+	+	+
Portage	+	+	NA	-

Table 2: Support for multiple releases and build variants. *NA* denotes that the criteria are not-applicable for homebrew and Spack. As it only provides one variant it can rely on default system paths.

1. **Depends-On**: the dependencies of the build tool itself.
2. **Ease-Add-Pkg**: the ease of adding another package to a stack.
3. **Ease-Bootstrap**: the ease of bootstrapping the build system itself. A prerequisite is the possibility to use it w/o root privileges.
4. **Documentation**: existence and quality of documentation

The findings are summarized in Table 3. The dependencies of most of the tools seem well under control. The addition of new packages is of similar complexity in all cases. Only the ease of bootstrapping differs. For most of them it is a simple checkout. However, for others like Nix an entire (fake-)root environment is necessary. The documentation of the domain specific projects is in general very poor, the documentation of the non-HEP tools surprisingly good and complete.

4.4 Other Criteria

During the meetings proponents brought up other criteria that are not so easy to group together. Some might consider these requirements, others might disagree. For now we've just listed them as "other". The criteria in this category are:

1. **Performance**: The relevant metrics are a. Length of time to build the entire stack, this includes support for parallel builds. b. Length of time to incrementally build a developer defined subset of the stack. c. Ability to reuse binary packages to speedup builds on different machines.
2. **Sys-Reuse**: the ability of the build system to reuse parts of the system software that is being built if desired.

	Depends-On	Ease-Add-Pkg	Ease-Bootstrap	Documentation
aliBuild	Python	spec-file	+	o
cmsBuild	Python,rpm,apt	spec-file	o	-
Contractor	Python	python file	git checkout	-
Homebrew	Ruby	Formula	+	+
LCGCMake	Python, Cmake	Cmake-macro	+	o
Nix	Perl	expression	-	+
SciSoft	Cmake	Cmake-macro	o	-
Spack	Python	auto template	git checkout	+
Worch	Waf,Python	Text-files	+	o
Portage	Python	shell script	+	+

Table 3: Ease of use of the various build tools.

	Performance	Sys-Reuse	Community	Unique-IDs	VCS-Support
aliBuild	+	o	-	+	+
cmsBuild	+	o	-	+	+
Contractor	+	+	-	-	+
Homebrew	+	+	+	-	+
LCGCMake	+	+	+	+	+
Nix	+	-	+	+	?
SciSoft	o	o	+	+	+
Spack	o	o	+	+	+
Worch	+	+	-	o	+
Portage	+	+	+	+	+

Table 4: Further considered features of the build tools. Details in the text.

3. **Community**: whether the tool is used by a wider community or not. Green means it is a standard opensource project (e.g. homebrew, conda, nix), yellow means it is HEP / science only community, red means it is a single experiment or person effort.

4. **Unique-IDs**: A method of uniquely identifying a build product such that if it already exists, it does not have to be built again. This feature insures that the software suite will always be built consistently throughout the whole stack. Further discussion with the group reveals that this point is complicated. We will expand on this in the next section.

5. **VCS-Support** Integrated support for check-out of given tags or branches from the projects repositories directly.

The findings are summarized in Table 4.

5 Sharing of porting/patching – "Build recipes"

At the core of all packaging systems sits the execution of a given set of build instructions for a particular package, so-called *build recipes*. Framed by collaboration-specific pre- and post-processing steps. The behaviour of these build instructions may be platform dependent or they may be influenced by parameters like build type or paths to their dependencies.

A lot, probably even most of the work in the packaging infrastructures goes into porting software and preparing these *recipes* for the various platform-dependency-compiler-version combinations. When comparing the (implicit) interfaces of the packaging systems towards their individual recipes, they are strikingly similar. Thus starting with sharing these recipes looks like a viable solution. Various options have been discussed on the HSF packaging github tracker [18]. A first implementation of this idea was done in the context of the ALICE experiment [19].

6 Summary

Together with most of the software librarians of the HEP community we assessed the existing build and packaging tools. For this we identified 17 criteria, which are however of different importance to different users. The goal was to see whether there is a base for working on a future common build tool.

While in general of high quality, the non-HEP tools we looked at seem very weak on the support for multi-stack, multi-configuration setups. On the other hand, the tool Spack, originating from the HPC community, covers all these points excellently. The problematic points of Spack are the lack of a possiblity to install pre-compiled binaries, and the non-relocatability of build artifacts. The former is a blocker for the HEP use-case, as the number of installations is much larger than in the very centralized HPC environment – even though the usage of CVMFS for software distribution changed the deployment scenarios significantly for HEP. Preliminary investigations indicate that enhancing Spack and/or combining it with other tool(s) would be straightforward. Therefore Spack currently seems to be the most suitable candidate for a common packaging tool. Among the HEP-specific tools examined so far, *LCGCMake* and *aliBuild* currently seem to be the primary candidates for generalization.

The next step would be to investigate whether the weak points of Spack can be addressed. Successful first steps were done in an HSF-specific fork of Spack. If successful, it could serve as the tool for the *HSF Reference Builds* of the projects participating in the HSF. A more ambitious goal would be enabling it as a common tool for building experiment stacks.

In parallel to the assessment of existing build tools, we discussed the concept of shared *build recipes*. As one of the next steps in the working group we will try to lay this out in a more concrete way.

Acknowledgements

We would like to thank all people participating in the constructive and lively discussions. In particular James Amundson, Marco Clemencic, Giulio Eulisse, Ben Morgan, Pere Mato, and Shahzad Muzaffar.

References

[1] Indico agendas of 25.2.2015, 2.6.2015, 9.6.2015, 16.6.2015, 23.6.2015, 4.11.2015, 18.11.2015, 10.2.2016, 2.11.2016

[2] Naming convention TN listed on http://hepsoftwarefoundation.org/technical_notes.html

[3] Worch: https://github.com/brettviren/worch

[4] LCGCMake: http://ph-dep-sft.web.cern.ch/document/using-lcgcmake

[5] cmsBuild: https://github.com/cmsbuild/cmsdist

[6] Contractor: https://cdcvs.fnal.gov/redmine/projects/contractor/wiki

[7] aliBuild: http://alisw.github.io/alibuild/

[8] Homebrew: http://brew.sh/

[9] Nix: https://nixos.org/nix/

[10] Spack: http://www.computer.org/csdl/proceedings/sc/2015/3723/00/2807623.pdf

[11] Portage: https://wiki.gentoo.org/wiki/Project:Portage

[12] Gentoo Prefix project https://wiki.gentoo.org/wiki/Project:Prefix

[13] Scientific Linux: https://www.scientificlinux.org/

[14] CentOS: https://www.centos.org/

[15] Debian: https://www.debian.org/

[16] Gentoo: https://www.gentoo.org/

[17] Ubuntu: https://www.ubuntu.com/

[18] Packaging Protocol Discussion: https://github.com/HEP-SF/packaging/issues/1

[19] The alidist package: https://github.com/alisw/alidist.