EPJ Web of Conferences **245**, 05035 (2020) https://doi.org/10.1051/epjconf/202024505035
CHEP 2019

SpackDev: Multi-Package Development with Spack

Chris Green[1,*], *James* Amundson[1], *Lynn* Garren[1], *Patrick* Gartung[1], and *Elizabeth* Sexton-Kennedy[2]

[1]Scientific Computing Division, Fermi National Accelerator Laboratory
[2]Office of the Chief Information Officer, Fermi National Accelerator Laboratory

Abstract. High Energy Physics (HEP) software environments often involve ~ hundreds of external packages and libraries, and similar numbers of experiment-specific, science-critical packages—many under continuous development. Managing coherent releases of the external software stack is challenging enough, but managing the highly-collaborative—and distributed—development of a large body of code against such a stack adds even more complexity and room for error.

Spack is a popular Python-based package management tool with a specific focus on the needs of High Performance Computing (HPC) systems and system administrators whose strength is orchestrating the discrete download, build, testing, and installation of pre-packaged or tagged third-party software against similarly stable dependencies. As such it is becoming increasingly popular within HEP as that community makes increasing use of HPC facilities, and as efforts to develop future HPC systems utilize Spack to provide scientific software on those platforms [1].

SpackDev is a system to facilitate the simultaneous development of interconnected sets of packages. Intended to handle packages without restriction to one internal build system, SpackDev is integrated with Spack as a command extension in order to leverage features such as dependency calculations and build system configuration, and is generally applicable outside HEP. We describe SpackDev's features and development over the last two years, initial experience using SpackDev in the context of the LArSoft liquid argon detector toolkit, and work remaining before it can be considered a fully-functional multi-package build system for HEP experiments utilizing Spack.

1 Introduction

Development of scientific software is by nature a collaborative exercise: scientists working together on a paper or project will divide between them the labor of writing code to produce, simulate or analyze data. Code for HEP experiments in particular is generally very library- and plugin-oriented: frameworks connect data sources, algorithms, and outputs. An application consists of many parts of different origins and organizational responsibilities, with the exact makeup of a particular application often being determined at runtime by a configuration file. Data definitions, utility code, algorithms, and framework code are logically distinct and often

*e-mail: greenc@fnal.gov

separated into different packages or repositories by category or subject matter such as tracking or clustering algorithms, simulation infrastructure, or event display facilities.

A large experiment such as the Compact Muon Solenoid (CMS) [2] can produce millions of lines of code, much of it interdependent, with a significant fraction under continuous development [3]. From a software engineering perspective, the HEP environment is complex: simulations, reconstruction algorithms and analyses change as an experiment improves its understanding of its experimental hardware, data acquisition systems, and the underlying physics being studied. Changes to the way *e.g.* functions from other libraries are called—the "Application *Programming Interface*" (API)—and other breaking changes are common and often affect many parts of the software system. Additionally, where a project utilizes a significant amount of C++ in internal or external software, compatibility between the compiled code of different libraries used in the same application—the "Application *Binary Interface*," (ABI)—is especially critical and requires careful management of the development environment to ensure compilations against pre-installed software produce consistent builds of composite applications. Achieving a manageable development cycle can be difficult when the entire system can take several hours to build and test, even factoring out the time required to build external dependencies such as database clients or domain-specific toolkits. In HEP, this problem is often solved with a multi-package development system or workflow.

2 Multi-Package Development in HEP

Using a tagged release of an experiment's software built and installed centrally on a (possibly distributed) filesystem as a base, a developer will obtain the version-controlled source for only those parts of the codebase they wish to develop and those necessary for a consistent build, and develop based on the most recent version of that codebase consistent with the selected central release. Approaches to achieving this have historically been applicable only to one or a small number of related experiments, although they share many of the following characteristics:

- Facilitation of the full development cycle including compiling, linking, testing, and—if applicable—installing or packaging.
- The development system will either use—or be part of—a particular underlying build system (*e.g.* GNU Make or CMake), often with specific macro packages or protocols.
- Utilization of a central area with pre-installed programs, libraries, and headers against which to develop higher-level packages locally.
- The ability to interact with one or more release, package management, or Source Code Management (SCM) systems.
- The developer will be able to choose multiple packages to develop together.
- Management of dependencies between packages being developed and pre-installed external packages, including versions and ABI-specific characteristics where appropriate.

We will focus on a specific multi-experiment ecosystem: users of the *art* [4, 5] event-based analysis framework. *art* is a multi-package suite with ∼15 external dependencies in addition to libraries from the operating system. It is used by multiple HEP experiments and projects[1] [6] and supported with an effort equivalent to less than 3 full-time people including development, build, test, release, distribution and support activities. The same team also produces source and binary release distributions packaged using UPS [7, 8] for the software produced by several of the experiments and projects utilizing *art* along with the more than 120 external products on which they rely. This has resulted in a large ecosystem of experiments and their developers

[1] ArgoNeuT, DUNE, ICARUS, LArIAT, MicroBooNE, Mu2e, Muon g-2, NOνA, SBND, artdaq, and LArSoft

with many common elements to their software environments. In addition to the framework, we provide an optional single-package CMake [9]-based build system, cetbuildtools [10], and a multi-package development system we developed, MRB, both of which depend upon specific behaviors of UPS [7, 8] to integrate locally-developed packages with a centrally-installed release of the full software suite.

3 Rationale for a Spack-Based Multi-Package Development System

Efforts to enable the *art*-using community to migrate away from using the 30 year old UPS as our package management system to Spack [11, 12] in order to meet future needs—especially HEP's increasing reorientation towards massively-parallel HPC systems—have been described elsewhere [13]. As part of that migration, our current development environment must evolve due to its heavy and explicit reliance on UPS via commands, conventions and environment variables. It would be reasonable to imagine that this evolution should avoid reliance on Spack to prevent a similar situation in the future requiring another overhaul. However, some level of integration between the development environment and a specific package management system can enable the development environment to ensure consistent builds with respect both to experiment code not being developed currently, and to compatible external dependencies. One would be able to check out from version control any intermediate packages required for consistency, and third-party dependencies could be installed on demand from source or binary packages. Additionally, this integration could enable the simultaneous development of packages with different internal build systems, useful for evaluation or validation of a new version of an external toolkit and adaption to changes in its API.

4 Spack Basics

Spack is an open source Python package originating at the Lawrence Livermore National Laboratory (LLNL) more than 8 years ago and it remains under active, funded development today. A significant community has formed around it, with nearly 700 people having contributed to the project thus far, over half of whom were active within the last year. Supporting all versions of Python from 2.6–3.8, the Spack application features commands and subcommands, and is comprehensively configurable via YAML [14] configuration files. Packages Spack knows how to build are represented by *recipes*, implemented as Python classes inheriting from a base `class Package` or a subclass thereof representing a known build system such as `class CMakePackage`, `class MakefilePackage` or `class AutotoolsPackage`. These recipes may be part of Spack itself or an external area. The main Spack installation provides recipes for more than 4000 packages at this time, with more being added on a daily basis.

Recipes tell Spack how a package should be configured, built, tested, installed, and made available to be used. Directives at the `class` level describe versions of the package and how to obtain them, dependencies on or conflicts with other packages, patches, or variants—different ways to build the package such as with or without certain features—and more. Additionally, methods such as `cmake_args()`, `build()`, `install()`, or `setup_run_environment()` may be overridden with package-specific actions.

When a command such as `spack install <package>@<version>+<feature>` is invoked, Spack *concretizes the "spec"*—resolves all the constraints described by the recipes and the installation command into a specific set of packages to use, obtain, or build as necessary. The problem of software package dependency resolution is *NP-complete* in terms of formal complexity theory, meaning that while a particular solution may be *verified* quickly, it may not be *findable* in a predictable (or even finite) time. Hence, package managers usually apply

heuristics to mitigate the problem [15]. Notwithstanding, the concretization process may still take a significant time for some package sets. However, Spack has the ability to output the results of a concretization process as a YAML file, allowing the calculations to be bypassed for subsequent operations on the same set of packages. In the context of a specified software stack with known dependencies taken from system software, this allows us to consider distributing the YAML file as part of the stack distribution.

In addition—as a result of a contribution we made [16, 17]—Spack is able to identify pre-built binary packages satisfying the user's specifications, download them from remote or local cache if found there, and configure them correctly for use in the installed location. Packages and dependencies not found in a binary cache will be obtained in source form, built and installed locally, and can be uploaded to a cache for installation elsewhere.

5 SpackDev Basics

We have developed a multi-package development system, SpackDev: a Python application which uses Spack to drive compilation and installation of dependencies, and to provide the information required to obtain development sources, integrate them with dependencies, and compile and test as needed.

SpackDev sets up a development area which is associated with a Spack installation. The user will specify a number of packages (the "development set") they wish to develop and a description of the package dependency network into which they should fit. This dependency network will be concretized by Spack, and SpackDev will identify any intermediate packages (packages not part of the development set but required because they are both dependencies of and dependents on packages in the development set) and add them to the set. The development set will be checked out from source as described in the Spack recipe for each package, and SpackDev will extract environment, tool, build, and test instructions from Spack to enable the user's development activities. Finally, SpackDev will use the concretized specification to instruct Spack to build and install all necessary dependencies. When this operation is complete, the development area will be ready to use. The user may build and/or test the full development set as a unit, or they may decide to focus on a particular package at a time, working in the appropriate environment for that package and building the whole set less frequently.

6 Some SpackDev Details

Originally conceived as a distinct Python application invoking Spack as an external utility, we have reimplemented SpackDev to use the Spack API directly by taking advantage of a recent third-party contribution [18, 19] to Spack allowing "command extensions" analogous to those allowed by the Git utility. This enables the concretization step to be carried out only once, and information on dependencies, source locations and build instructions to be usable directly via Spack's functions and internal data structures rather than via intermediate files after extraction via Spack's YAML export facility and other ad hoc methods.

SpackDev makes use of CMake's *ExternalProject* [20] module to drive the build for each package in the development set, as configured using information extracted by Spack from the recipe for each package. Each package in the development set is therefore built separately, in an order determined by Spack's concretized specification, thereby avoiding an inherent constraint on the build system used by each package internally. Parallel build operation is enabled on a per-package basis—packages with no dependency relationship may be built simultaneously, and on a per-file basis within each package.

The user specifies the full dependency network—essentially the fully-specified release distribution of which the development set is a part—by means of a file containing valid

specifications for `spack install` commands. Versions, compilers, and package options for top-level packages and dependencies are specified using Spack's normal syntax. In effect, this file *is* the release: the only thing the user needs in addition to Spack itself and any additional recipes to replicate a given software environment on their own system.

7 The SpackDev Workflow

A reasonable workflow for developing multiple packages together as part of a larger release with SpackDev is likely to include the following phases:

1. **Initialize the development area and build dependencies.**

   ```
   $ spack dev init [--dag <install-spec-file>] <package>...
   ```

   This operation will initialize a new development area, checking out source for the development set as necessary. The Spack installation—including recipes and installed dependencies—may be exclusive or shared between multiple development areas.

2. **Build the development set.**

   ```
   $ spack dev build
   ```

   Build (and optionally test) the full development set together.

3. **Rapid development cycle for one package.**

   ```
   $ spack dev build-env --cd <package>
   $ make; ctest # Or whatever is appropriate for <package>.
   ```

   Initialize a sub-shell with an environment suitable for developing `<package>`. Exit the sub-shell to return to the previous environment. For Bash users, the `--prompt` option will add the name of the package to the command prompt.

8 Test Case: the LArSoft Toolkit

LArSoft [21, 22] is a C++ toolkit for simulation and reconstruction of HEP events in liquid argon Time Projection Chambers (TPCs), comprising approximately 20 packages and > 120 external packages upon which they depend. Developing a new release is a complex exercise, and the requirement that contributors be able to construct and use a coherent development environment with minimal expenditure of time, effort and computing resources is a non-trivial one. For someone developing an addition to LArSoft in the context of their own experiment and its software, there is an even greater need for dependencies and build coherency questions to be taken out of the equation.

In order to be able to provide a "technology preview" to our target audience of *art*- and LArSoft-using experiments, we developed a self-contained system—the *Minimal Viable Product* (MVP)—wherein all the dependencies required to develop packages are built locally, with one Spack installation for basic tools such as the compiler and its dependencies, and another for everything else. A "bootstrap" script checks system prerequisites, downloads all relevant components such as Spack, external recipe repositories and SpackDev itself, and builds the basic tools. The bootstrap script also configures Spack to find certain packages on the system rather than building them from scratch, thereby truncating the dependency network somewhat. Even so, the LArSoft network has 140 nodes and > 400 edges, and the full transient network comprises nearly 220,000 nodes.

A significant amount of up-front work was required in order to be able to test LArSoft with Spack and SpackDev, as the suite was developed within the current UPS-based packaging

and development model, including the UPS-dependent cetbuildtools build system. Recipes were created for LArSoft packages and those of their external dependencies that did not already have recipes available in the main Spack distribution. Additionally, the source for each LArSoft package required modification to remove its build system's dependence on UPS via cetbuildtools. This required the development of a new, related build system: cetmodules.

During the initialization stage of a development area for the full LArSoft suite, the concretization of the dependency network takes a significant time—about 180 seconds on an Opteron-6136 server with no significant other load. Concretization is a serial procedure, so time reduction via parallelization is not an option for this task. Other calculations and extraction of information from dependency and recipe information take a few more seconds, and then the development set is checked out and dependencies built—the latter taking by far the bulk of the time.

The build of the development set is straightforward, and development of an individual package likewise. However, it is easy to forget to enter a new development environment (or exit the old one) when moving frequently between global builds and builds of multiple individual packages.

9 Lessons Learned: Remaining Issues

9.1 The Need for a Scalable Distribution Model

For an experiment of many tens of developers, a self-contained, locally-built copy for every developer of every software release that developer needs is simply not scalable, especially when building that software release consumes several fully-loaded hours of a multi-core development server and $\geq$ 25 GiB of online storage. Instead, it must be possible to develop against a compatible centrally-installed release of known provenance, and rely on the environment being set up correctly and consistently for every development session. A reasonable release and distribution model would produce versioned releases of a validated software stack with precise version, feature and compilation specifications for all internal and external software. As envisioned here, each such release would manifest as:

1. a YAML file for each supported combination of feature and compilation specifications describing the fully-concretized dependency network as calculated by Spack;
2. the corresponding Spack recipes for each package; and optionally:
3. pre-built binary archives for each package in the release for supported platforms.

This would allow local managers to install specific releases into central Spack installations on their own systems either from a central, vetted build cache or by building some or all of the release from source as determined by the fully-concretized dependency network description. In the situation where microarchitecture-specific compilation of some packages is required (*e.g.* for CPU vectorization or offloading to GPGPUs or other accelerators), a hybrid central installation combining pre-built and locally-built packages is also feasible. SpackDev would connect the relevant central release to a developer's local builds via Spack's "chain" [23] facility.

9.2 Development Environment Consistency

This early version of SpackDev is not robust against user error in terms of ensuring the correct environment in which to perform development operations either globally or with

individual packages. However, we expect that the provision of package-specific commands like `spack dev build|test|install <package>` analogous to the global `spack build` command will eliminate the issue of correct environment for the vast majority of users of the system while still allowing the use of package-specific build commands and environments if that flexibility is required for unusual situations. Spack's "environments" [24] facility will also be evaluated for use by SpackDev for this purpose.

9.3 Initialization Time

The time required for initialization of a development area for a large software release with many packages and complex interdependencies is currently a major obstacle to adoption of Spack and SpackDev by an HEP experiment. Notwithstanding the issue of building an entire release locally as addressed in section 9.1, significant time is required to concretize a particular release, integrate the specified development set, and identify any additional intermediate packages to build locally. However a completely new implementation of Spack's concretizer is due to be completed during 2020 which should vastly decrease the time required to concretize a complex dependency network. We have also requested the ability to access the concretizer from SpackDev in order to speed up the time required to calculate required intermediate packages.

9.4 Package Re-Use

In addition to the use of centrally-installed releases by multiple developers, it is also reasonable to reduce the footprint of multiple such releases by having only one installed copy of a particular version and variant of a software package across multiple colocated releases. Several current characteristics of Spack mitigate against this, particularly the lack of a concept of a package which is not dependent upon a particular compiler and language standard. Examples are the data-only packages common in HEP, those built only from C source, or consisting only of interpreted-language code such as Python, Perl or other scripting languages. Additionally, the hashed checksum of a recipe is a major factor in deciding whether a pre-built package is suitable for re-use, limiting re-use of packages built for previous releases either installed locally or stored in a build cache. Based on continuing interactions with the developers, we expect the new concretizer to alleviate these issues and more, although some development work within SpackDev will be required to take best advantage of the improvements.

9.5 Generalization

In order to become a reasonable general solution, we must generalize SpackDev to enable the development of any combination of packages regardless of build system or any other characteristic of their recipes. We believe this will be best achieved with agnostic build and install commands as described in section 9.2, and by closer integration with Spack.

10 Conclusion

The issues outlined above do demonstrate that more work is required before an integrated model using Spack and SpackDev satisfies all our requirements for development, packaging, and distribution of experimental software. However, we believe that SpackDev has the potential to become an essential part of the workflow not only of developers on *art*-using experiments but elsewhere in HEP and the Spack-using scientific computing community at large.

11 Acknowledgments

We gratefully acknowledge the Spack community in general, and its principal authors in particular, for their help.

This manuscript has been authored by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the U.S. Department of Energy, Office of Science, Office of High Energy Physics.

References

[1] M.A. Heroux, J. Carter, R. Thakur, J.S. Vetter, L.C. McInnes, J. Ahrens, T. Munson, J.R. Neely, Tech. rep., United States Department of Energy (2020), https://doi.org/10.2172/1597433

[2] C.P. Office, *CMS*, https://web.archive.org/web/20200623084340/https://home.cern/science/experiments/cms

[3] K. Lassila-Perini et al., *The CMS Offline SW Guide*, https://twiki.cern.ch/twiki/bin/view/CMSPublic/SWGuide

[4] C. Green, J. Kowalkowski, M. Paterno, M. Fischler, L. Garren, Q. Lu, *The Art Framework*, in *Proceedings, 19th International Conference on Computing in High Energy and Nuclear Physics (CHEP 2012): New York, USA, May 21–25, 2012* (2012), Vol. 396, p. 022020

[5] The SciSoft Team <scisoft-team@fnal.gov>, *The art event processing framework*, http://art.fnal.gov/

[6] The SciSoft Team <scisoft-team@fnal.gov>, *Who uses art?* (2018), https://web.archive.org/web/20181030144907/http://art.fnal.gov/who-uses-art/

[7] M. Votava, W. Bliss, S. Cutts-Bone, C. Debaun, F. Donno-Raffaelli, R. Herber, K. Leininger, B. Lindgren, J. Nicholls, G. Oleynik et al., *UPS UNIX Product Support*, in *Seventh Conference Real Time '91 on Computer Applications in Nuclear, Particle and Plasma Physics Conference Record* (1991), pp. 156–159, https://doi.org/10.1109/RTCON.1991.673182

[8] M. Mengel et al., *The ups redmine project*, https://cdcvs.fnal.gov/redmine/projects/ups

[9] Kitware, Inc., *CMake* (2012), http://cmake.org/

[10] The SciSoft Team <scisoft-team@fnal.gov>, *The cetbuildtools project*, https://cdcvs.fnal.gov/redmine/projects/cetbuildtools

[11] T. Gamblin, M. LeGendre, M.R. Collette, G.L. Lee, A. Moody, B.R. de Supinski, S. Futral, *The Spack Package Manager: Bringing Order to HPC Software Chaos*, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (ACM, New York, NY, USA, 2015), SC '15, pp. 40:1–40:12, ISBN 978-1-4503-3723-6, https://doi.org/10.1145/2807591.2807623

[12] T. Gamblin et al., *Spack on github*, https://github.com/spack/spack

[13] C. Green, J. Amundson, L. Garren, P. Gartung, M. Paterno, *Spack-Based Packaging and Development for HEP*, in *European Physical Journal Web of Conferences* (2019), Vol. 214 of *European Physical Journal Web of Conferences*, p. 05013, https://doi.org/10.1051/epjconf/201921405013

[14] T.Y. project, *YAML*, https://yaml.org/

[15] R. Di Cosmo, Technical report, Environment for the Development and Distribution of Open Source (EDOS) (2005), https://hal.inria.fr/hal-00697463

[16] P. Gartung, *[Spack] Create, install and relocate tarballs of installed packages*, https://github.com/spack/spack/pull/4854
[17] T. Gamblin et al., *[Spack] Build Caches*, https://spack.readthedocs.io/en/0.14.2/binary_caches.html
[18] M. Culpo, *[Spack] Spack can be extended with external commands*, https://github.com/spack/spack/pull/8612
[19] T. Gamblin et al., *[Spack] Custom Extensions*, https://spack.readthedocs.io/en/0.14.2/extensions.html
[20] B. King et al., *ExternalProject*, https://cmake.org/cmake/help/v3.18/module/ExternalProject.html
[21] E. Snider, G. Petrillo, J. Phys. Conf. Ser. **898**, 042057 (2017)
[22] The SciSoft Team <scisoft-team@fnal.gov>, *The LArSoft project*, https://larsoft.org/
[23] T. Gamblin et al., *[Spack] Chaining Spack Installations*, https://spack.readthedocs.io/en/0.14.2/chain.html
[24] T. Gamblin et al., *[Spack] Environments*, https://spack.readthedocs.io/en/0.14.2/environments.html