

# On Spack, Spackdev and their incorporation into the HEP software ecosystem at FNAL

Revision 0.2

---

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Overview . . . . .	2
1.2	Document Outline . . . . .	2
1.3	Terminology . . . . .	3
<b>2</b>	<b>Ecosystem Overview</b>	<b>4</b>
<b>3</b>	<b><i>Spack</i></b>	<b>6</b>
3.1	Product Description . . . . .	6
3.2	Technicalities, Questions and issues . . . . .	6
3.3	Snapshot: Limitations of Spack and their Consequences . . . . .	6
3.3.1	Overly-rigid treatment of compiler as a dependency . . . . .	7
3.3.2	Inability to propagate variants in a general manner . . . . .	7
3.3.3	Issues with inheritance of specs . . . . .	7
<b>4</b>	<b><i>Spackdev</i></b>	<b>7</b>
4.1	Product Description . . . . .	7
4.2	Technicalities, Questions and issues. . . . .	7
<b>5</b>	<b><i>cetbuildtools</i></b>	<b>7</b>
5.1	Product Description . . . . .	7
5.2	Technicalities, Questions and issues. . . . .	8
<b>6</b>	<b><i>Lmod</i></b>	<b>8</b>
6.1	Product Description . . . . .	8
6.2	Technicalities, Questions and issues. . . . .	8
<b>7</b>	<b>Integration</b>	<b>9</b>
<b>8</b>	<b>Plan</b>	<b>9</b>
8.1	Statement of Final Goal . . . . .	9
8.2	Success Criteria . . . . .	9
8.3	Current Status . . . . .	10
8.4	Milestones . . . . .	10
	<b>Bibliography</b>	<b>12</b>

## 1 Introduction

### 1.1 Overview

In 2014, the SSD group (now TAC) put together a requirements document[1] detailing the many requirements for HEP software package building and management, both internal and external, for experiments and collaborations in the *art* [2] ecosystem. This document described primarily what stakeholders were accustomed to expecting in the areas of software development, wider software ecosystem and environment management, and other related subjects. Examples are relocatability, the ability to develop multiple packages at once, and the ability to manage consistent and coherent environments with different releases of experimental software and their external dependencies. There were also some, “wish-list” items: requirements not currently or completely satisfied by the pantheon of tools in use at the time which, while obviously not essential, were desirable. Up to the present day, the requirements described in the document were satisfied by tools like *Redmine*, *Git*, *Subversion*, *relocatable UPS*, *CMake*, *GNU Make*, *Ninja*, *cetbuildtools* [3], *SoftRelTools* [4], *SCONS* [5], *MRB* [6], *UPS* [7] and various scripts and utilities based around the *ssibuildshims* toolkit.

Since then, the landscape has changed: HEP is being pushed toward HPC platforms and applications wherever possible, and some platforms have evolved in ways that are inimical to current methods of satisfying some of the requirements. As an example, *MacOS* has instituted, “SIP,” a security and protection system which renders the `LD_LIBRARY_PATH/DYLD_LIBRARY_PATH` method of ensuring relocatability inoperable by preventing the export of certain environment variables to subprocesses.

At both Fermilab and in the wider HEP community, *Spack* [8] has been mooted as a possible solution. *Spack* is a software packaging tool originating in the HPC community which manages the specification of build and install procedures for software, including handling dependencies.

With some enhancements, and in conjunction with *Spackdev*, a *Spack*-derived toolkit developed at Fermilab to handle multi-package simultaneous development and a suitably enhanced *cetbuildtools*, it is believed that *Spack* can satisfy all the requirements listed in the above-mentioned document in the expanded modern HEP ecosystem that includes SIP-enabled *MacOS* machines and flagship HPC systems. It is also possible that the *Lmod* system could contribute positively to the ecosystem.

This document is intended to lay out issues and items for consideration in the development of an HEP ecosystem utilizing the systems mentioned above, and a roadmap therefor.

### 1.2 Document Outline

Chapter 2 will describe the envisioned ecosystem, along with a description of the *Spack*, *Spackdev*, *cetbuildtools*, and *Lmod* products in their current states.

Chapters 3, 4, 5 and 6 will deal respectively with *Spack*, *Spackdev*, *cetbuildtools* and *Lmod* and the issues specific to each, while chapter 7 will deal with issues of their integration into a whole, with any other “glue” that might be appropriate.

Finally, chapter 8 will lay out a plan for the implementation of an HEP ecosystem using these tools, including milestones, and discuss issues associated with adoption by experiments and collaborations, and migration from their respective existing systems. The issue of collaboration and coordination with the HEP XYZ will also be addressed therein.

### 1.3 Terminology

*ABI*: An *application binary interface* (ABI) is the low-level interface between two program modules, and determines such details as how functions are called and in which binary format information should be passed from one program component to the next, relating to how a binary product was built. For C++ and Fortran software, the compiler version, language standard, and the operating system version are critical for ABI compatibility. For *Python* software, byte-code compiled libraries are compatible across operating systems, but not generally across *Python* interpreter versions. Often, ABI mis-matches can lead to linking failures. Sometimes they lead to more subtle errors, and are very difficult to diagnose.

*API*: An *application programming interface* (API) is the source-code level interface between two program modules. It is related to source code version. API mis-matches typically lead to compilation failures.

*active product*: An *active product* is one that is currently being used. For executables, this means they are found on the `PATH`; for libraries, that they will be loaded when needed, *etc.* Only one variant of a given product can be active at one time.

*available product*: An *available product* is one that can be made active using the product management tool's command(s) to activate the product. Many variants of a given product may be available at the same time.

*closed link*: A *closed link* is the linking of a dynamic library leaving no unresolved symbols.

*evaluation of a test*: *Evaluation of a test* means determining whether the given test succeeds or fails.

*external product*: An *external product* is one for which the builder of the product is not in control of the software or its build system.

*integration build area*: An *integration build area* is a collection of *build areas*, managed in coherent fashion.

*locally-developed product*: A *locally-developed product* is one whose source is under the control of the developer or release manager.

*package (v.)*: To make a product ready for distribution in its already-built form.

*product*: See *software product*.

*product build area*: A *build area* is a directory tree into which is put the files generated by the building of *standard build targets* for a single *software product*.

*product source area*: A *product source area* is a directory tree containing the source code for a single software product. Such a product source area should always be under the control of a *source code management system*.

*release*: A particular coherent set of products of known version.

*setup (v.):* Produce an environment in which the software can be built or in which a consistent group of products can be used.

*software product:* A *software product* (also shortened to *product*) is an identifiable, separately packaged body of software. It can include such software entities as libraries, executables, C and C++ headers, Fortran module files, documentation and executables. A product is built and delivered as a unit, and is the smallest unit of versioning.

*source code management system:* A *source code management (SCM) system* is a system used for version control of source code. These systems are also called *source code repositories*.

*standard build targets:* A build system typically supports the following list of high-level targets, which we call *standard build targets*. These include:

*default:* The execution of targets that produce files. These include executables, libraries, and test programs. Also known as the *build stage* or (for historical reasons), the *all* target.

*test:* Runs user-provided integration and unit tests.

*install:* Installation of executables, libraries, headers, *etc.*, in the appropriate location(s).

*package:* The production of an “installation kit,” suitable for distribution of the build products, to be used without requiring the ability to build the product.

*clean:* The removal of produced files.

*help:* List available individual component targets, such as libraries, executables and object files.

*target:* A *target* is an identifiable sub-component of a development build procedure that usually (but not always) corresponds to a generated file. Examples would be a particular executable, library or test.

*toolchain:* A term for the collection of tools used to build a software product, such as compilers, code generators and build management tools.

*umbrella product:* An *umbrella product* is a software product that contains only dependencies on other products, and has no additional code, libraries, data files, or executables of its own.

*variant:* A *variant* of a product is a particular version and built configuration of a product. Here, “built configuration” could include such attributes as the platform, compiler used, debug or optimization level, or optional feature set.

*version:* The *version* of software specifies the source code text of the software. It determines the API to which users of a product program, *e.g.* the number and types of the arguments for a specific function.

## 2 Ecosystem Overview

The envisaged ecosystem is intended to satisfy the following overarching requirements:

1. To provide a coherent high level system for describing how to build software products (including those comprising the toolchain) which does not prescribe the build system (*e.g.* autotools, CMake, *etc.*) used for those packages.

2. To provide a system for managing dependencies between products, including those which depend on particular “variants” of a given software package (compiler, option sets, etc).
3. To allow for distribution of internally-consistent sets of pre-built products, with a means to make those products functional in their new locations.
4. To provide an environment for the simultaneous development of multiple interdependent products.

In addition, it is desirable to:

1. Leverage tools and product build specifications provided by the wider HEP and HPC communities.
2. Minimize as far as possible the amount of in-house development and maintenance required, both for the products comprising the ecosystem and those products built and packaged using it.

Spack[8] is intended to be the centerpiece of this ecosystem: this tool provides the ability to produce build specifications for products using a well defined Domain Specific Language (DSL) implemented within *Python*. These specifications describe the possible variants and dependencies of each product, and provide a prescription for exactly how the product should be built and installed. When a specification is executed, all required dependencies are built if not found on the system. This can lead to large product sets (if left to its own devices, Spack will build *binutils*, *make* and *CMake*, for example), but mechanisms exist to mitigate this issue, such as `packages.yaml`.

A recent Fermilab contribution to Spack was the ability to create caches of pre-built binaries which, when installed in their final position, are updated to be able to find their dependent libraries.

*Spackdev* [9] is a small collection of utilities to allow the source checkout and simultaneous development of multiple interdependent packages for building via *spack*.

*cetbuildtools* [3] is a collection of utilities and *CMake* code to make the building of HEP products, especially those using the *art* framework, more straightforward. In its current form several aspects of its operation assume the use of products made available using *relocatable UPS*. A *cetbuildtools* enhanced to no longer require *relocatable UPS* would provide the ability to build easily HEP products under *Spackdev*.

*Lmod* [10] is a backwards-compatible next-generation replacement for the venerable HPC tool *Environment Modules* [11]. Both tools manipulate the user’s environment to make products available (e.g. by amending `PATH`), but the latter is reputed to be able to deal with variants and consistent sets of interdependent products. *Spack* is able to produce *Lmod* configurations for the products it builds. More study is required to understand

if *Lmod* (and *Spack*'s use thereof) will satisfy our need for a coherent user environment when using multiple *Spack*-built products together.

## 3 *Spack*

### 3.1 Product Description

*Spack* is a product management tool written in *Python*. It has a concept of a repository of, “package files,” also in *Python*, which describe how products should be built, including their dependencies and any variants such as compiler or product-specific feature selections. When asked to install a product, *Spack* will, “concretize” its dependencies based on the product’s requirements and decide what (if anything) needs to be installed first. *Spack* may be configured to satisfy dependencies from its primary installed area, other *Spack* installed areas, external areas such as `/usr` or `/usr/local`, or from a binary, “buildcache,” in preference to building dependencies from scratch.

*Spack* is quickly becoming an HPC-standard tool with a large scientific community contributing both to the product itself and to its default supply of available “package files.”

### 3.2 Technicalities, Questions and issues

- It is unclear exactly what fraction of the available package files will be useful to us in unaltered form, given *e.g.* Fermilab-specific patches or variants, *etc.*
- The `packages.yaml` facility is currently not capable of the flexibility required to allow easily certain products such as *Lua* or *Python* to be used from the system.
- *Spack*'s `spack info` gives inconsistent variant information for products mentioned in `packages.yaml`. It is unclear what implications this might have for builds of products based on them.
- The buildcache facility is new, basic and untested with a complex set of products including dependencies satisfied from the system and those built with different compilers. Further enhancements are likely to be necessary.
- *Spack*'s dependency system is currently unable to deal directly with (valid) dependencies on products that were built with a different compiler. Multiple *Spack* installation areas are a possible solution, but the particulars and logistical issues with this are currently uncertain and require investigation.
- The issue of making products available for use is delegated by *Spack* to a module system such as *Lmod*. Whether this satisfies the requirement of being, “safe” from the point of view of ensuring consistent product sets is currently uncertain, may depend upon exactly which module backend is in use, and requires investigation.

### 3.3 Snapshot: Limitations of Spack and their Consequences

As of 2018-04-12, the develop branch of *Spack* has the following limitations with consequences for a *Spack* and *Spack*-based build system:

### 3.3.1 Overly-rigid treatment of compiler as a dependency

### 3.3.2 Inability to propagate variants in a general manner

### 3.3.3 Issues with inheritance of specs

## 4 Spackdev

### 4.1 Product Description

*Spackdev* is a *Python* utility that works in conjunction with an existing *Spack* installation to create and manage areas in which multiple products can be developed simultaneously. *Spackdev* handles the checkout of specified product sources into an area (including any intermediate packages necessary for a coherent build), and the creation of *CMake* glue to allow the multiple products to be built together.

*Spackdev* also has the ability to create a limited `packages.yaml` file based on certain products found on the system, to create a relatively uniform environment from which to develop without building the world from the ground up.

### 4.2 Technicalities, Questions and issues.

- *Spackdev* appears to assume that all products' source is available via *Git*. It is unclear whether there is a mechanism for incorporating manually-checked-out product sources, for instance using *Subversion* or *CVS*.
- Likely use cases include the checking out of multiple product sources with different branches or version tags. It is unclear what if any enhancements would be required to support this, or if branch / tag adjustments would be done manually after the fact.
- *Spackdev* assumes that the build-system to be used is *CMake*-based, as does *MRB* in the current ecosystem and is limited to the *GNU Make* or *Ninja* generators. In time, support for the *XCode* generator on *MacOS* may be a reasonable enhancement.
- How *Spackdev* may interact with a complex *Spack* configuration such as multiple repositories, install areas and buildcaches may need investigation and adjustment.
- *Spackdev*'s ability to produce and/or manipulate a `packages.yaml` file will need to be enhanced as our understanding of *Spack*'s features and our needs improves, and as they evolve.

## 5 cetbuildtools

### 5.1 Product Description

*cetbuildtools* is a collection of utility scripts and *CMake* files to facilitate the building of HEP packages, especially but not exclusively those intended to work with the *art* framework. Features include the ability to set up the dependent products required to build a particular product, generate *relocatable UPS* table files and *CMake* config

files for product installation, build *ROOT* dictionaries and other plugin libraries, and a comprehensive utility for declaring and configuring *CMake* tests. It is also able to integrate with the *MRB* system, which provides support for *CMake*-based multi-package development.

## 5.2 Technicalities, Questions and issues.

- In addition to the *relocatable UPS* table file generation functionality, there are many places in *cetbuildtools* which rely on environment variables to obtain information about products that are currently provided by *relocatable UPS*, such as `XXX_VERSION`, `XXX_INC` and `XXX_LIB`. These uses would have to be replaced by the equivalent calls to obtain the information from *Spack*.
- Functionality would need to be added to replace the current *relocatable UPS* table file generation and dependency setup facilities with equivalent facilities to generate *Spack* package files from a template and make dependent packages available (e.g. via `spack load`).

## 6 *Lmod*

### 6.1 Product Description

*Lmod* is the latest implementation of a tool called *Environment Modules*, long used in the HPC community, used for managing availability to the user of product with different versions. It operates using *modulefiles*, which describe what variables should be added or altered in the user's environment to make a product available for use. Modulefiles are written in the *Lua* scripting language; or Tcl modulefiles created for the earlier-generation *Environment Modules* may be used for backward compatibility reasons.

### 6.2 Technicalities, Questions and issues.

- *Spack* commands such as `spack load` may be configured to use *Lmod*, but it is unclear how different variants of a product are dealt with since vanilla *Lmod* only understands the concept of one or two numeric version numbers in order to decide what product to set up..
- *Lmod* does not appear to have the ability to throw an error when a product is set up which is inconsistent in its dependencies, either internally or with products already set up in the current environment. Instead, *Lmod* simply overrides the earlier product version with the later-specified one.
- *Lmod* operates by adding variables to the environment, including variables describing the location of libraries and modifying `PATH`. While *Lua* adds a smaller number of new variables per product to the environment than *relocatable UPS*, it is still a significant addition to the environment when tens or even a small number of hundreds of products are involved.

## 7 Integration

The integration of the aforementioned tools into a coherent ecosystem will require not only an understanding of the issues mentioned in earlier sections, but also a thorough understanding of exactly what a fully functioning instance of the ecosystem would look like, including possibly multiple *Spack* installations and repositories, some shared across multiple experiments, and providing for multiple versions of multiple product distributions in multiple variants.

Minimum reassurance of a fully functioning system will involve:

- Multiple releases of a distribution with multiple compilers and multiple variants.
- A mix of installed products including those in different spack installation areas, some installed from scratch, and some installed from buildcache after having been compiled elsewhere (and where such products and dependencies are no longer in their originally-referenced locations on the final installation target).
- A set of products satisfied by `packages.yaml` which differs in meaningful respects from the environment in which products installed from buildcache were compiled originally.
- A successful build of a set of products with *Spackdev* built on such a heterogeneous system, followed by successful use of those built products via buildcache on a different system, from a different location.

It remains to be seen whether *Lmod* is a necessary part of the ecosystem, and what the environment would look like in that case.

As described hitherto, there remain significant uncertainties in the operation characteristics of the final system. In addition, the development cycle of the *Spack* product is current fast-moving and somewhat unstable. Consequently, an automated set of tests to detect regressions in the behavior we need as described above would be highly desirable moving forward, even post-development.

## 8 Plan

### 8.1 Statement of Final Goal

The ultimate goal of this project is to have an ecosystem which is able long-term to support the development, distribution and use of Fermilab HEP software projects and experimental collaboration software on all required platforms with only reasonable migration costs for those projects and experiments from their existing systems.

### 8.2 Success Criteria

In order for the *Spack* and *Spackdev*-based ecosystem project to be considered a successful replacement to the old *relocatable UPS*, *cetbuildtools* and *MRB*-based system:

1. At least one software project previously using *relocatable UPS* products, *cetbuildtools* and *MRB* must have migrated its development, build-and-test, and distribution operations to the new ecosystem.

2. We must have buy-in from all the groups who will ultimately be responsible for producing release distributions of HEP software that is currently based on any of the following:
  - Distributions of *relocatable UPS*-packaged products.
  - *cetbuildtools*.
  - *MRB*.

This includes not only experimental collaborations and software projects but also those groups producing release distributions. Ultimately, we must have an agreed-upon end date for Fermilab support for *relocatable UPS* product distributions of experimental and support software, and for *MRB* and the *relocatable UPS*-bound *cetbuildtools*.

It will have to be possible for middleware projects such as *art*, and likely *LArSoft*, to produce both *relocatable UPS* and *Spack* products in the short term, but maintaining both ecosystems in parallel is not sustainable going forward due to the maintenance burden of providing parallel and compatible sets of product stacks including third-party products.

### 8.3 Current Status

- A preliminary implementation of buildcache functionality has been added to *Spack*.
- A preliminary implementation of *Spackdev* is available.
- A very preliminary version of *cetmods*, the *UPS*-free successor to *cetbuildtools* is in process but needs work.
- Basic product stacks are being worked on for **CMS** and *art*, and a stack for **CMS'** *FWLite* already exists.
- Explorations with stacks involving multiple compilers are just beginning.

### 8.4 Milestones

1. Build a basic tool stack comprising two distinct toolchains (discounting the native compiler) on one *Linux* and one *MacOS* platform, making maximum reasonable use of products that may be reasonably expected to be available on each system ([issue #18396](#)).
2. Configure the current environment to be able to use a product's executables and any libraries both at link and execution time on Linux and on a MacOS system with SIP enabled ([issue #18406](#)).
3. "Build" an empty product that has only dependencies ([issue #18397](#)).
4. Set up multiple products for use by referring directly only to an umbrella product ([issue #18408](#)).
5. On both *Linux* and *MacOS*, build a small set of products in one location, transfer it to another via buildcache and successfully build a dependent product against it without the originally-built set being present ([issue #18399](#)).
6. Use successfully a non-trivial product relocated to a system with a materially different `packages.yaml`, *i.e.* where product dependencies were in `packages.yaml` but now are not, or *vice versa*, or were in `packages.yaml` in both installations but in different locations ([issue #18409](#)).

7. Use successfully a non-trivial product relocated to a system where a dependency is a virtual package whose definition and/or location has changed ([issue #18410](#)).
8. Produce a *Spack*-based distribution of the `canvas_product_stack` *art*-precursor distribution on both toolchains and both platforms mentioned above ([issue #18411](#)).
9. Demonstrate (on both *Linux* and *MacOS*) successful use of interactive *ROOT* after relocation, both of itself and (independently, from and to different locations) the dependencies against which it was built ([issue #18412](#)).
10. Produce a successful *Spack* build of the *art* and *gallery* suites passing all tests ([issue #18413](#)).
11. Produce a successful *Spackdev* build of the *art* and *gallery* suites passing all tests ([issue #18414](#)).
12. Produce a successful *Spackdev* build of *critic*, passing all tests, using *art* and *gallery* as installed and relocated ([issue #18415](#)).
13. Produce usable *relocatable UPS* tarballs for products built using *Spack* and *Spackdev* ([issue #18416](#)).
14. Produce a full distribution built using the Jenkins facility, and successful use thereof ([issue #18417](#)).
15. Select and use one of two available *variants* of a release distribution ([issue #18418](#)).
16. Select and use one of two available *versions* of a release distribution ([issue #18419](#)).
17. Produce a release distribution utilizing debug variants of *art* suite and selected third party products ([issue #18420](#)).
18. “Clean up” an old release distribution while retaining those products that are required by other releases ([issue #18421](#)).
19. When building releases against multiple compilers, avoid unnecessary duplication of installations of non-architecture- or non-compiler-dependent products (*e.g.* comprising only headers, or data, or native-compiled executables and/or C-only libraries) ([issue #18422](#)).
20. Demonstrate the ability to install easily and maintain and use a workable installation of multiple release distributions ([issue #18423](#)).
21. Produce a set of automated tests of the ecosystem to guard against regressions introduced during continued development of its constituent tools ([issue #18424](#)).
22. Obtain Fermilab management buy-in on replacement of the old ecosystem ([issue #18425](#)).
23. Obtain agreement in principle from all software projects and experimental collaborations still using one or more components of the old system ([issue #18426](#)).
24. Obtain agreement on an end date from all stakeholders, with agreement from management on the effort required to support both systems during the transition period ([issue #18427](#)).
25. Cease distribution of new software releases using *relocatable UPS*, and of all development of *relocatable UPS*-bound *cetbuildtools*, and of *MRB* ([issue #18428](#)).

## Bibliography

- [1] Garren L, Green C, Kowalkowski J and Paterno M Requirements for software product building and management URL <https://cd-docdb.fnal.gov:440/cgi-bin/RetrieveFile?docid=5380&filename=requirements.pdf&version=1>
- [2] The *art* event processing framework: URL <http://art.fnal.gov>
- [3] The *cetbuildtools* home page: URL <https://cdcvs.fnal.gov/redmine/projects/cetbuildtools>
- [4] Amundson J 2001 *SoftRelTools* version 2 at Fermilab *Computing in High Energy and Nuclear Physics (CHEP 2000): Proceedings. (Computer Physics Communications vol 140)* ed Mazzucato M and Michelotto M pp 731–732
- [5] The *SCONS* home page: URL <http://www.scons.org/>
- [6] The *MRB* home page: URL <https://cdcvs.fnal.gov/redmine/projects/cetbuildtools>
- [7] The *UPS* and *UPD* home page: URL <http://www.fnal.gov/docs/products/ups/>
- [8] The *Spack* home page: URL <https://github.com/spack/spack/wiki>
- [9] The *Spackdev* home page: URL <https://github.com/amundson/spackdev>
- [10] The *Lmod* environment modules home page: URL <https://www.tacc.utexas.edu/research-development/tacc-projects/lmod>
- [11] The *Environment Modules* home page: URL <http://modules.sourceforge.net>