# Extending ROOT through Modules

*Oksana* Shadura[1,*], *Brian Paul* Bockelman[1,**], *Vassil* Vassilev[2,***]

[1]University of Nebraska Lincoln, 1400 R St, Lincoln, NE 68588, United States
[2]Princeton University, Princeton, New Jersey 08544, United States

**Abstract.** The ROOT software framework is foundational for the HEP ecosystem, providing multiple capabilities such as I/O, a C++ interpreter, GUI, and math libraries. It uses object-oriented concepts and build-time components to layer between them. We believe that a new layering formalism will benefit the ROOT user community.
We present the modularization strategy for ROOT which aims to build upon the existing source components, making available the dependencies and other metadata outside of the build system, and allow post-install additions on top of existing installation as well as in the ROOT runtime environment. Components can be grouped into packages and made available from repositories in order to provide a post-install step of missing packages. This feature implements a mechanism for the more comprehensive software ecosystem and makes it available even from a minimal ROOT installation. As part of this work, we have reduced inter-component dependencies in order to improve maintainability.
The modularization effort draws inspiration from similar efforts in the Java, Python, and Swift ecosystems. Keeping aligned with modern C++, this strategy relies on forthcoming features such as C++ modules. We hope formalizing the component layer provides simpler ROOT installs, improves extensibility, and decreases the complexity of embedding ROOT in other ecosystems.

## 1 Introduction

One of the advantages of object-oriented systems is the ability to have abstraction layers that provide separation between the user interfaces and implementations. Ultimately, it is an important tool to help to scale projects to larger code bases. Another technique for scaling projects is to use a modularization. Modularisation provides a grouping of functionality into distinct units with clear points of interaction. While ROOT [1] has a strong history in object-oriented programming, it doesn't have as a strong a concept of modular components. Thus, we propose to add new concepts to the ROOT ecosystem:

1. *Component*: A set of interdependent classes implementing coherent functionality and providing well-defined APIs. A *library* could be defined as a component or set of related components that expose functionality and which can be invoked by a program or another library.

*e-mail: oksana.shadura@cern.ch
**e-mail: bbockelm@cse.unl.edu
***e-mail: vvasilev@cern.ch

2. *Package*: A distinct, self-describing resource (file, URL) that provide one or more components (such as a library and associated non-code resources).
3. *Package database*: A record of all packages currently available in a ROOT installation.
4. *Package manager*: An actor that can locate and install packages into a ROOT installation from a package reference (such as a package URL), along with their transitive dependencies.

Similar to ROOT, other large object-oriented software systems consist of a number of interdependent and loosely-coupled classes and are mostly organized as a set of libraries and build targets. In these cases, classes are often used as the lowest level of granularity and can serve as a unit of software modularization.

In ecosystems such as Java, Python, and C++, a further package structure can allow software developers to organize their programs into components. A good organization of classes into identifiable and collaborating packages simplifies the understanding and the maintenance of the software. Yet another technique to improve the quality of the software modularization is to provide technical mechanisms for encouraging coders to utilize these package structures.

## 2 Motivation

Software modularization defines a way of grouping functionalities. It outlines groups in the form of components, which identify a particular piece of functionality that solves a specific problem. Often, it provides a way for coders to carefully specify the visibility of interfaces and differentiate this between inside the package and outside. In general, modularization helps reducing management, coordination and development costs. We aim to define a set of mechanisms that enable a modular version of ROOT, centered around C++ modules [2] and the concept of software packages.

ROOT libraries have a very complex set of interdependencies. By introducing a component layer on top of these libraries, we provide better boundaries between components, allowing ROOT to scale as a project. We hope this can also make it easier for external communities to develop packages on top of ROOT and easily distribute them – without needing to get their work into ROOT's official codebase.

A first step is to define a *minimal ROOT* package, removing as many dependencies as possible, until only the simplest user functionality remains; we decided the minimal functionality should effectively be the I/O libraries and C++ interpreter. By having a smaller size and set of external dependency, we increase the chances that its functionality can be embedded in other contexts and enables ROOT users to interact with the broader data science ecosystem.

Packages and package management provides a mechanism for ROOT users to socialize and reuse projects built on top of ROOT. We aim to help make ROOT more flexible and open it to the new customers. This feature would allow ROOT to serve as a community nexus. In particular, it provides the ROOT team with an improved mechanism to say "no" to new components within the ROOT source itself as users can directly share their packages among each other or in a conventional store such as GitHub.

## 3 ROOT components and packages

The ROOT code [3] is organized into a set of sub-directories per component, with the exception of the ROOT Core library, which has a more complex structure of subfolders.

A component is a single unit of code distribution or set of classes for a framework or an application that is built and ready to be shipped. Another C++ module [4] or another component can import it with a hook, such as an “include" keyword in case of headers (or the “import" keyword in case of C++ modules).

As an example, Figure 1 shows how a package can be created based on multiple components. Here, a hypothetical *ROOT-ML package* consists of the TMVA (“Toolkit for Multivariate Data Analysis") component and ROOT RDataframe as its dependencies.

The ROOT modularization work is well-aligned with the effort to integrate C++ modules work going [2] in ROOT. C++ modules provide a simple way to produce software libraries with improved compile-time scalability and management of the API of a library. C++ modules improve encapsulation and outline a clear relationship between public and private part of the code, splitting the two into implementation and interface. For ROOT, we are working on an ecosystem to improve versioning and binary distribution.

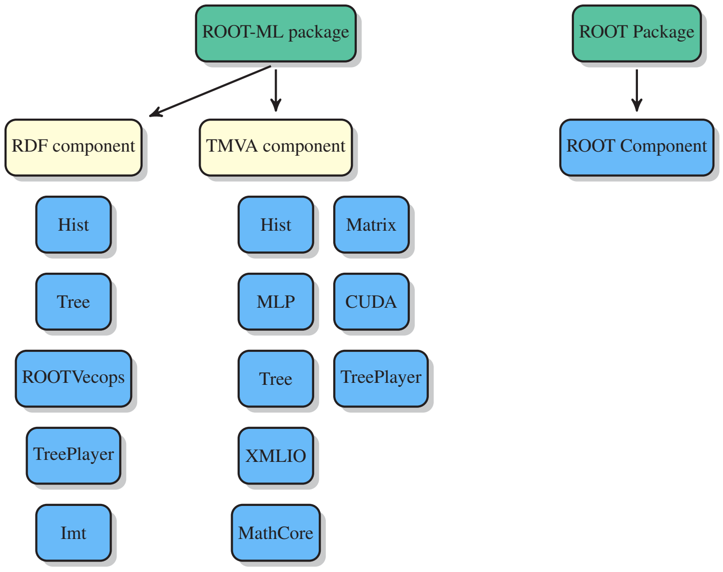

**Figure 1:** Example of ROOT component and package.

One of the main challenges is to define the package granularity; best practices here remain an open question. Too large number of components in packages defeats the purpose of modularization. Similarly, packages should not contain too many small components as this may introduce significant package management overhead.

A first ROOT package to define is *ROOT Base*, which includes cling (the C++ interpreter), ROOT I/O, and other ‘Core’ components. *ROOT Base* is a fundamental part from which we start to modularize ROOT framework. Another example of a ROOT package would be a *ROOTMath* package that consists of multiple *math* related components, such as ROOT libraries MathCore, MathMore and VecCore (see Listing 1).

We define a package to be a grouping of software and associated resources intended for its distribution and reuse. In order to create a package, we assume a specific organization of code for build and deploy steps. The package’s definition, versioning, metadata, content description, and build information is contained in a manifest file. An example of our manifest file format is shown in Listing 1. Package resources can include build byproducts such as a shared library or an executable, or the package documentation and unit tests. The manifest example was inspired by Swift manifests [5], and is written in YAML [6].

Entities who may interact with manifest files may include:

```
package:
  name: "ROOTMath"
  targets: "ROOT::MathCore ROOT::MathMore VecCore::VecCore gsl::gsl mathcore-tests
        mathmore-tests"
  products:
    package:
    name: ROOTMath
    targets: ROOT::MathCore ROOT::MathMore VecCore::VecCore ROOT::Imt gsl::gsl
  module:
    name: MathCore
    publicheaders: inc/<enumerated headers>.h
    sources: src/<enumerated source files>.cxx
    targets: ROOT::MathCore
    dependencies: VecCore Imt
    tests: mathcore-tests
  module:
    name: MathMore
    publicheaders: inc/<enumerated headers>.h
    sources: src/<enumerated source files>.cxx
    targets: ROOT::MathMore
    dependencies: gsl MathCore
    tests: mathmore-tests
  module:
    name: VecCore
    packageurl: "https://github.com/root-project/veccore/archive/v0.5.1.zip"
    targets: VecCore::VecCore
    tag: 0.5.1
  module:
    name: gsl
    packageurl: "https://github.com/ampl/gsl/archive/v2.5.0.zip"
    targets: gsl::gsl
```

**Listing 1:** Draft version of a YAML-based manifest file for the ROOTMath package.

1. *ROOT subsystem developer (such as an I/O developer)*: Here, the purpose of the manifest is strictly informational. The information for manifest is generated from the build system; the build system will help to produce the manifest file.

2. *The third-party developer*: For example, a Ph.D. student who wrote a new ROOT package as a part of work on the thesis and would like to describe in a human-readable from a build description of his package and which ROOT components it depends on.

3. *A member of an experiment physics group*: Provides the ability to build a particular library on demand or to share their developments.

## 4 Package manager design prototype

As a part of the package manager prototype was defined two different future work areas: ROOT Base package as a fundamental part of ROOT, and development of ROOT package management tool.

### 4.1 Evolving "Minimal ROOT" to "ROOT Base"

While generating the ROOT libraries dependency graph, we can quickly notice that it is hard to visualize a small core of ROOT as desired for our ROOT Base package.

ROOT's build system provides a "minimal ROOT" option that is supposed to build only the essential functionality of ROOT required for basic I/O operations. At the outset of this project, nearly fifty components were built when this *minimal* option was enabled; we believe that "Minimal ROOT" has migrated away from its original goal of being only a *core-like*

ROOT. We have worked to trim the minimal build down include three components, Core, I/O, and Cling libraries, and the smallest number dependencies required.

Our *ROOT Base* is formed by taking these three components and their transitive dependencies using CMake-based introspection.

### 4.2 ROOT Package Manager - root-get

#### 4.2.1 Design specifications

A package manager is a standalone tool for managing the distribution of software code. Package managers are classified into the three sub-types: operation system package manager (OSPM), language package manager (LPM) and project or application dependency manager (PDM). We will focus on PDMs since operation system package managers and language package managers are out of the scope of the goal of this paper. A project dependency manager is an interactive system for managing the source code dependencies of a single project written in a particular language offering multiple operations on the dependent code of the project. Project dependency manager's output is a self-contained and precisely reproducible source tree that acts as the input to a compiler or interpreter. It could also be referred to as a *compiler, phase zero* [7].

The ROOT package manager best fits as a project dependency manager. Since ROOT has incorporated C++ interpreter in its code source, the project dependency manager's functionality could be expanded to be made available at both ROOT build time and runtime. This enables the possibility to develop hooks allowing one to use the package manager during ROOT runtime. Using the package manager during runtime is where a pure CMake implementation would fall short as CMake does not have any support for steps happening after build or install time.

An essential part of the ROOT package manager's design is the use of C++ modules technology. C++ modules can optimize header parsing, providing a possibility of loading on-demand code. In this article, we will be talking about ROOT C++ modules, which are C++ modules generated by internal Clang compiler, available in ROOT and heavily relying on the Clang implementation of C++ modules.

Since ROOT has own interpreter, ROOT C++ modules can be used during ROOT runtime and in this case we will reference them as a ROOT runtime C++ modules or implicit C++ modules. We will reference ROOT C++ modules used during ROOT compile time, as explicit C++ modules.

ROOT implicit C++ modules solve the limitation that the ROOT compiled header (PCH) is non-separable or non-modular and PCH must be regenerated each time when a new ROOT library is configured. Implicit C++ modules are available in the ROOT 6.16.00 release binaries, as a technology preview, which allows us to test them during the package manager development.

#### 4.2.2 Requirements for the ROOT package manager

The ROOT package manager should satisfy a set of functionality requirements expected by ROOT users. That includes more comprehensive sets of operations with the manifest files, sanity checks of the content of the manifest files, and execution commands. At the current stage of research, it is essential to define minimum requirements for functionality of the package manager, definitions of entities (such as a component and package), and the format of the manifest file.

For our purposes, the minimal requirement for the ROOT package manager is to be able to define and resolve dependencies and versions through the manifest of the package. These should be done either for ROOT packages or its external dependencies.

Ideas for future extensions are inspired by the functionality of Swift Package Manager [8], which is supported by a large community:

1. Automated testing;
2. Support of cross-platform packages;
3. Support for operating system package managers (homebrew, etc.);
4. Support for version control system;
5. Standardized licensing;
6. Introduction of a package index;
7. Importing dependencies by source URL;
8. Component inter-dependency determination;
9. Complex dependency resolution.

Using benefits from the introduction of C++ module infrastructure for ROOT, we followed the ideas of Swift Package Manager and implemented a standalone tool - ‘root-get’ which provides functionality meeting the a minimal requirements described above.

#### *4.2.3 root-get prototype*

We have implemented a dependency management tool for ROOT (Figure 2) called ‘root-get’. It consists of the multiple modules that provide the desired package management functionality:

1. **Analyzer** defines environment variables, checks if there are existing manifest/package YAML files, and sets up the environment (discovering the location of components and packages and preparing for the manifest’s generation).
2. **Generator** encapsulates the ROOT CMake functions for generating information for manifest files. It allows one to configure ROOT modules and packages outside of ROOT using special CMake files containing the definitions about ROOT CMake functions and ROOT external dependencies.
3. **Downloader** is a set of helper routines for downloading packages from Github or other location.
4. **Resolver** is a module that provides a package database generation and resolution of dependencies via generated direct acyclic graph (DAG).
5. **Builder** is a module that provides ROOT packaging scripts.
6. **Integrator** is a module that provides installation and deployment routine for ROOT packages.

A root-get is using the generic approach, existing for all "package managers" and works directly with the source code of components. Manifest files, in this case, are usually the files generated by static code analysis tools or provided by the user. The core part of package manager workflow is a "lock file". It is a file containing the project dependencies that are generated directly from manifest files. The expected result after package manager operations is a delivery of the set of artifacts: compiled source code and its dependencies, which can serve as a direct “input” to the interpreter and usually generated from lock file.

All compiled source code, declared by the lock file should be arranged on disk in such way that the compiler or interpreter can use it as intended, but it will be still isolated to avoid mutation. ROOT components arranged as a set of packages could be installed in any location, even outside of the install path of ROOT. All necessary components for the successful setup are ROOT Base installation and root-get installed in the system.

ROOT Base together with root-get is capable of providing proper infrastructure for custom ROOT distribution and natural extension of ROOT functionalities, which includes the possibility to build any ROOT components and also plug-in custom user components.

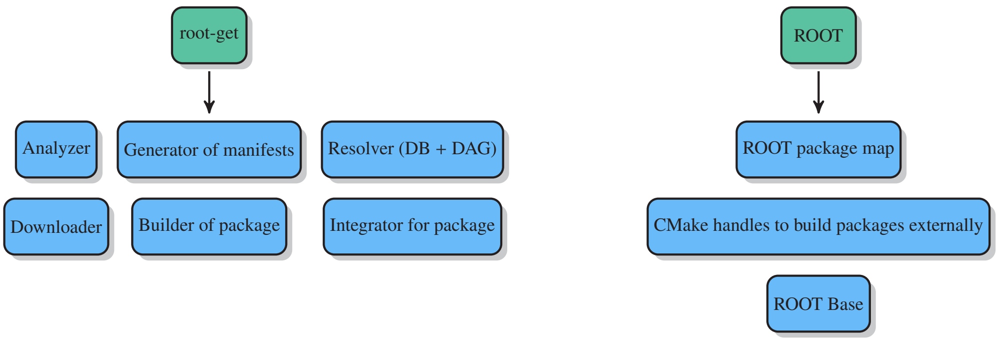

**Figure 2:** Components of root-get prototype.

## 5 Conclusions

We have defined the desired functionality for a package management ecosystem for ROOT. We defined a minimal and extended set of requirements for the ROOT package manager. These ideas have been adopted into a preliminary prototype that can download and install packages. As it matures, the prototype will be connected directly to the ROOT runtime and serve as a runtime dependency management tool.

This work has been supported by U.S. National Science Foundation grants OAC-1450377, OAC-1450323, and PHY-1624356.

## References

[1] R. Brun, F. Rademakers, *ROOT - An Object Oriented Data Analysis Framework*, Nucl. Inst. & Meth. in Phys. Res. A **389** (Proceedings AIHENP’96 Workshop,1997).
[2] V. Vassilev. *Optimizing ROOT’s Performance Using C++ Modules. Journal of Physics: Conference Series*, **898**. 10.1088/1742-6596/898/7/072023. (2016)

[3] GitHub.*GitHub - root-project/root: The official repository for ROOT: analyzing, storing and visualizing big data*, Available at: https://github.com/root-project/root. [Accessed 03 December 2018]. (2018)

[4] *Modules — Clang 8 documentation*. Available at: https://clang.llvm.org/docs/Modules.html. [Accessed 30 November 2018]. (2018)

[5] GitHub. *GitHub - apple/swift: The Swift Programming Language*. Available at: https://github.com/apple/swift. [Accessed 30 November 2018]. (2018)

[6] Yaml.org. *YAML Ain't Markup Language (YAML™) Version 1.2.* [online] Available at: http://yaml.org/spec/1.2/spec.html [Accessed 6 Mar. 2019].

[7] Medium. *So you want to write a package manager – Sam Boyer – Medium.* Available at: https://medium.com/@sdboyer/so-you-want-to-write-a-package-manager-4ae9c17d9527. [Accessed 30 November 2018]. (2018)

[8] GitHub.*GitHub - apple/swift-package-manager: The Package Manager for the Swift Programming Language.* Available at: https://github.com/apple/swift-package-manager. [Accessed 30 November 2018]. (2018)