Extending ROOT through Modules

Brian Paul Bockelman (University of Nebraska Lincoln)
Oksana Shadura (University of Nebraska Lincoln)
Vassil Vassilev (Princeton University)
Introduction
Introducing modules/packages to the ROOT follows the design principle: **do not pay for what you don’t use.**

**Realistic goal:** instead of downloading more than 1GB of full ROOT sources or pre configured ROOT binaries, you can decide to start with minimal set ~50 Mb and expand with any customization you want.
ROOT dependency map

..and even much more complex..
Motivation for ROOT Package Manager (PM)

- **Better layering:** introducing a module layer, that allow ROOT to scale as a project.
- **Better boundaries:** by making the boundaries and relationships more explicit through modules - we can better define “minimal ROOT,” that enables ROOT users to interact with the wider data science ecosystem.
- **Sharing between users:** package management provide a mechanism for ROOT users to socialize and and reuse projects built in the context of ROOT, that allow ROOT be more flexible and open for new customers.

These items allow ROOT to continue successfully to serve as a HEP community nexus.
Main motivation for application in HEP and beyond

Need of an improved ROOT granularity

CMSSW: ROOT is one of core dependences, any changes are causing to rebuild almost whole stack of packages

Easy versioning of dependencies (ROOT externals and builtins)

ROOT Packages intend to introduce a different flow of design & development of software around ROOT's ecosystem
Background studies on PMs
Classification in package management

- **Operating system or system package manager (SPM):** dpkg, apt-utils, yum, pacman, homebrew, nix..

- **Language package manager (LPM):** npm, pip, cargo, maven, swift...

- **Project/application dependency manager (PDM):** Cocaopods, Unity3
  - System for managing the source code dependencies of a *single project* in a particular language.

To avoid possible confusion: ROOT PM is not SPM!
We consider ROOT PM as PDM system.

https://medium.com/@sdboyer/so-you-want-to-write-a-package-manager-4ae9c17d9527
There is a lot of interest on trying to resolve “Package Management” problem for C++ projects. On recent C++ meeting was introduced a new SG15 Tooling group and was presented candidates (PMs) for C++ projects: [vcpkg, build2, spack, conan]. But nothing that could fit for ROOT use case. Since ROOT has its own interpreter, we can achieve much more out of it!

We got inspired by:

- build2 | C++ Build Toolchain
- CONAN.io
- Swift Package Manager
- (COCOAPODS)
Implementation and details
**ROOT package & module**

- **ROOT module** is a single unit of code distribution (set of classes) for a framework or an application that is built and shipped and that can be imported by another module with a hook (such as `#include` or an import keyword). A program may have all of its code in a single module (or inside of more modules), or it may import other modules as dependencies.
  - *Example: XMLIO module or HDFS module for ROOT IO*

- **ROOT package** is defined as a grouping of software for data analysis and associated resources, intended for its distribution (extension or upgrade of ROOT functionality).

- The definition of package assumes a contract for code organization in order to simplify the build and deploy steps. The contract defines a *manifest file* and particular organization of each module.
  - *One of the possible examples of ROOT package is abstract “Math” package*, that could consist of multiple math related modules (e.g. TMVA, MathCore) depending on package vendor.
Usage scenarios and benefits of manifest files

1. I am a ROOT subsystem user or developer (e.g. io). The manifest file is generated by the info in the build system.

2. I am a third party developer (PhD student) who has 5 files and does not know anything about build systems and alike -- I 'just' describe in a human form what my package does and what ROOT components it depends on.

3. I am experiment librarian and I know what exactly I need -- writing manifest file or some other configuration to tell ROOT what packages I need is the ideal scenario. The other scenario could be to describe a pre-built package.

Examples from industry: LLVMBuild utility that organize LLVM in a hierarchy of manifest files of components to be used by llvm-build, that is responsible for loading, verifying, and manipulating the project's component data.

```
LLVMBuild.txt:
[component_0]
type = Tool
name = llvm-diff
parent = Tools
required_libraries = AsmParser BitReader IRReader
```
Connect the PM to ROOT's runtime

- This is where CMake falls short as it does not have any support for steps happening after build/install time
- PM allows bootstrapping minimal ROOT and installing packages automatically on demand
- It provides a basic interpreter functionality, which will allow to call:

```cpp
#include "TMVA/DataLoader.h"

error: TMVA/Dataloader.h not found.

note: TMVA/Dataloader.h is part of TMVA package, do you wish to install it?[Y/n]

auto dataloader = new TMVA::DataLoader("test"); // works without quitting ROOT.
```
Demo is in Backup slides (only in PowerPoint version) or https://asciinema.org/a/jUkLgJjxpUZISFrh9Cv9dkeUz
Demo is in Backup slides (only in PowerPoint version) or https://asciinema.org/a/jUkLgJjxpUZISFrh9Cv9dkeUz
Demo is in Backup slides (only in PowerPoint version) or https://asciinema.org/a/jUkLgJjxpUZISFrh9Cv9dkeUz
Generic approach for PM

- Project code: module/package code
- Manifest file is file generated by static code analysis tools or provided by user
- Lock file: it is a file with project dependencies generated from manifest
- Dependency code - “input” to interpreter, generated from lock file
ROOT Package manager flow

Project code

Manifest file

Project manifest file

- src/
- inc/
- test/
- CMakeList.txt with
  `ROOT_STANDARD_LIBRARY_PACKAGE()`

or

- module1/
- module2/
- module3/
- ..

module:
  name: "Rint"
  Packageurl:
  "https://github.com/root-project/root.git"
  Path:
  /$SOURCE/core/rint
  publicheaders: *.h
  sources: *.cxx
  targets: Rint
  deps: Core

package:
  name: "X"
  targets:
    target:
      name: "X"
  products:
    package:
      name: X
    module:
      name: "Y"
      ..
Package manager flow

Generating a dependency graph (DAG) and resolving dependencies via various strategies), listed in the project’s manifest

Generating and deploying X.zip:
- inc/
- libY.so
- License file
- Manifest.yml
- Y_rdict.pcm
- Y.rootmap

- All of the source code from lock file, arranged on disk in a such way that the compiler/interpreter can use it as intended, but isolated to be avoid mutation.
- Modules/packages can be installed in any location, even outside of the install path of ROOT, all you need is to have ROOTBase and root-get installed in system.
ROOT package manager: ingredients

root-get

Analyzer
Downloader

Generator of manifests

Resolver (DB + DAG)

Builder of package

Integrator for package

ROOT

ROOT package map

CMake handles to build packages externally

ROOT base part

root-get
R&D: how to ship ROOT C++ runtime modules

- For more information about ROOT C++ modules check Yuka Takahashi’s poster “Optimizing Frameworks’ Performance Using C++ Modules-Aware ROOT” [https://indico.cern.ch/event/587955/contributions/2937639/]

- ROOT PM is trying to solve other complex questions for distribution of C++ runtime modules in a ROOT:
  - How we can define order for translation units to be compiled?
  - Build system, does it need to have a naming policy to be able to discover binary module interfaces (BMI)?
  - How to introduce mapping between translation units and identifiers?
  - How to introduce versioning of the same components?
Conclusions

- We described how could look like package management ecosystem for ROOT
- During research was defined the basic concepts needed for a implementation
- Also during work was defined additions to CMake build system for generating packages from a ROOT build
- All ideas was adopted in a preliminary prototype (root-get) that can download and install packages. A prototype could be connected to ROOT runtime and serve as a runtime dependency management tool [work in progress]

A lot of thanks to our GSOC 2018 student Ashwin Samudre!
Thank you!
Backup slides
Demo

https://asciinema.org/a/jUkLgJjxpUZISFrh9Cv9dkeUz
ROOT build system

- Configuration: CMake
  - Make files were deprecated in February 2018
- Codebase: C++
  - + PyROOT bindings: Python
- Also Python is used for generating pch & rootmap and command line utils

- /build/unix/git_coreteam.py
- /build/unix/makepchininput.py
- /build/unix/rootmapcat.py

- ./main/python/rooteventselector.py
- ./main/python/rootprint.py
- ./main/python/rootbrowse.py
- ./main/python/rootdrawtree.py
- ./main/python/rootmv.py
- ./main/python/rootcp.py
- ./main/python/rootmkdir.py
- ./main/python/rootslimtree.py
- ./main/python/rootrm.py
- ./main/python/cmdLineUtils.py
- ./main/python/rootls.py

ROOT has ~ 150 components + ~60 built in dependencies!
ROOT has 115 options!
PDM & “compiler, phase zero”

- System for managing the source code dependencies of a single project in a particular language. That means specifying, retrieving, updating, arranging on disk, and removing sets of dependent source code. PDMs reproducible output is a self-contained source tree that acts as the input to a compiler or interpreter. => “compiler, phase zero.”

*“compiler, phase zero' idea is similar to JIT'ing. In addition to putting the code on disk, the PDM typically needs to override the interpreter’s code loading mechanism in order to resolve includes correctly (PDM is producing a filesystem layout for itself to processed.)”

Complex ROOT CMake build system

- SearchSoftware.cmake was searching for all packages based on defined set of ROOT modules plus already enabled by default a set of build options and enabled modules
  - User has a less control if custom package was actually enabled because of complex requirements (ROOT builtins and SearchSoftware.cmake)

- Planned improvements to ROOT CMake build system by ROOT team
  - Introduce ON/OFF plugin through packages and not a modules
  - Update the way how builtins packages are build
  - Update the way how external packages are been searched
  - If user is requesting package, and for some reasons you can't reach requirements for package ROOT build will be failed.
Possible application in HEP and beyond: motivation

- ROOT Packages intend to introduce a different flow of design & development of software around ROOT's ecosystem
- What we are trying to do is to introduce a different way of thinking
- Our tool can help to sanitize a ROOT codebase
- In the same time we can try to define a popularity of ROOT packages
  - Example from Cocoapods: https://cocoapods.org/pods/protobuf-c
- It could be a “custom way” of distribution ROOT
ROOT PM and its relation with ROOT C++ modules

● ROOT C++ modules & runtime modules (ROOT is using Clang C++ modules)
  ○ Clang C++ modules is precompiled headers that optimize header parsing
    ■ Clang can load on-demand code from modules
    ■ It is similar to ROOT PCH
  ○ Due we build interpreter, we are optimizing header parsing at runtime and we call it runtime C++ modules

● ROOT C++ modules will solve problem that ROOT PCH can be only one => important part for ROOT PM design

● In the same time while using C++ modules for PM, we will try to help to solve a global problem of distribution C++ modules

● Status: ROOT is working on enabling C++ modules and runtime C++ modules
  ○ For more details about ROOT C++ modules please check poster of Yuka Takahashi (University of Cincinnati)
ROOT external map and CMake handles

- How to divide a ROOT into packages and modules?
  - We introduced ROOTComponentMap.cmake to help us with this task (static and dynamic)
- We will provide a possibility to plug your own map of packages and modules

custom_map.yml:

ML:

MathCore, MathMore, PyRoot, TMVA, CustomDiplomaOfPHDStudentLib

```bash
cmake ../ -DML=ON or root-get -i ML
```

(only that will be needed is a manifest file of CustomDiplomaOfPHDStudentLib)
ROOT PM: Analyzer

- Defining environment variables
- Checking if we have already existing manifest(package).yml files
- Preparing for generation routine: discovery of path for modules and packages, preparation for manifest’s generation.

ROOT PM: Generator of manifest

- CMake Routine in ROOT for recording info for manifest files
  - We are able to configure ROOT modules and packages outside of ROOT using few cmake files containing all information about ROOT macro (RootFramework.cmake) and ROOT external dependencies (ROOTDependencies.cmake)
ROOT PM: Integrator

- Routine that install, provide packaging and root-get to deploying packages (including regeneration of PCH) and its dependency code if needed

ROOT PM: Builder

- Routine that allow root-get to build and test packages

ROOT PM: Downloader

- Routine that allow root-get to work with external packages