ROOT package management: “lazy install” approach

Oksana Shadura
ROOT Monday meeting

Outline

- How we can improve artifact management (“lazy-install”) system for ROOT
- How to organise dependency management for ROOT
- Improvements to ROOT CMake build system
- Use cases for installing artifacts in the same ROOT session

Goals

- Familiarize ROOT team with our planned work
- Explain key misunderstandings
- Give a technical overview of root-get
- Explain how root-get and cmake can work in synergy

Non Goals

We are not planning to replace CMake

No change to the default build system of ROOT

No duplication of functionality

We are planning to **“fill empty holes”** for CMake

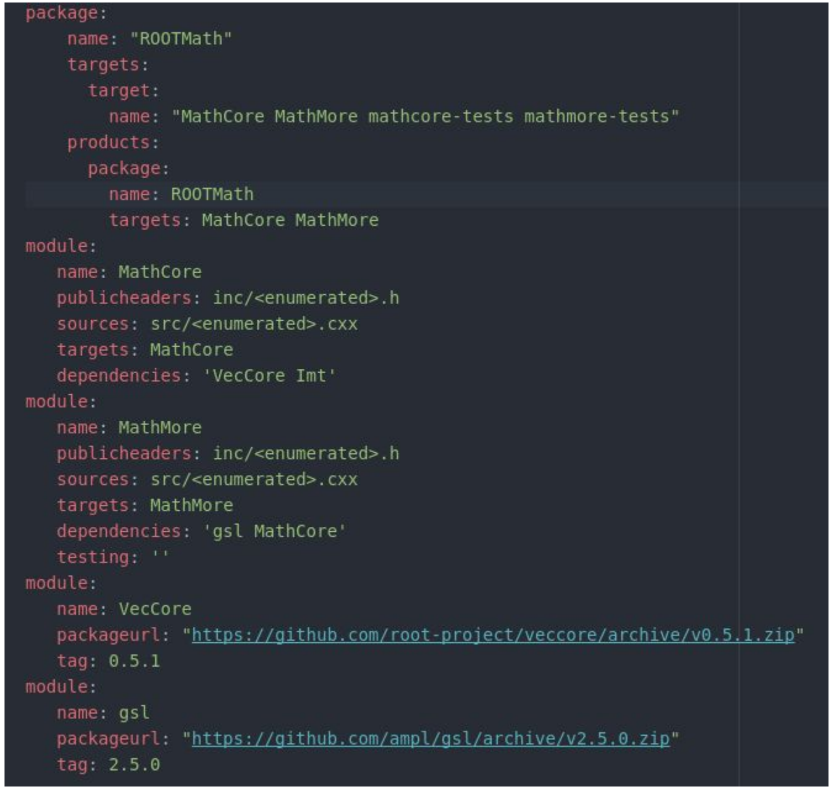
General overview

Manifest - why we need it?

- Easy to write
- Easy to parse, while CMakeLists.txt **is impossible to parse**
- **Collect information from ROOT’s dependencies + from “builtin dependencies” + OS dependencies + external packages to be plugged in ROOT (to be resolved after using DAG)**
- It can be easily exported back as a CMakeLists.txt
- It can have extra data elements [not only what is in CMakeLists.txt, but store extra info]
    - Dependencies description (github links, semantic versioning)
        - url: "ssh://git@example.com/Greeter.git",
        - versions: Version(1,0,0)..<Version(2,0,0)

Manifest is a “dump” of status of build system (BS), where root-get is just a helper for BS

Manifest - Sample



```
package:
    name: "ROOTMath"
    targets:
      target:
        name: "MathCore MathMore mathcore-tests mathmore-tests"
    products:
      package:
        name: ROOTMath
        targets: MathCore MathMore
module:
   name: MathCore
   publicheaders: inc/<enumerated>.h
   sources: src/<enumerated>.cxx
   targets: MathCore
   dependencies: 'VecCore Imt'
module:
   name: MathMore
   publicheaders: inc/<enumerated>.h
   sources: src/<enumerated>.cxx
   targets: MathMore
   dependencies: 'gsl MathCore'
   testing: ''
module:
   name: VecCore
   packageurl: "https://github.com/root-project/veccore/archive/v0.5.1.zip"
   tag: 0.5.1
module:
   name: gsl
   packageurl: "https://github.com/ampl/gsl/archive/v2.5.0.zip"
   tag: 2.5.0
```

Usage scenarios and benefits of manifest files: LLVM/Clang

LLVM use CMake as a build system

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
```

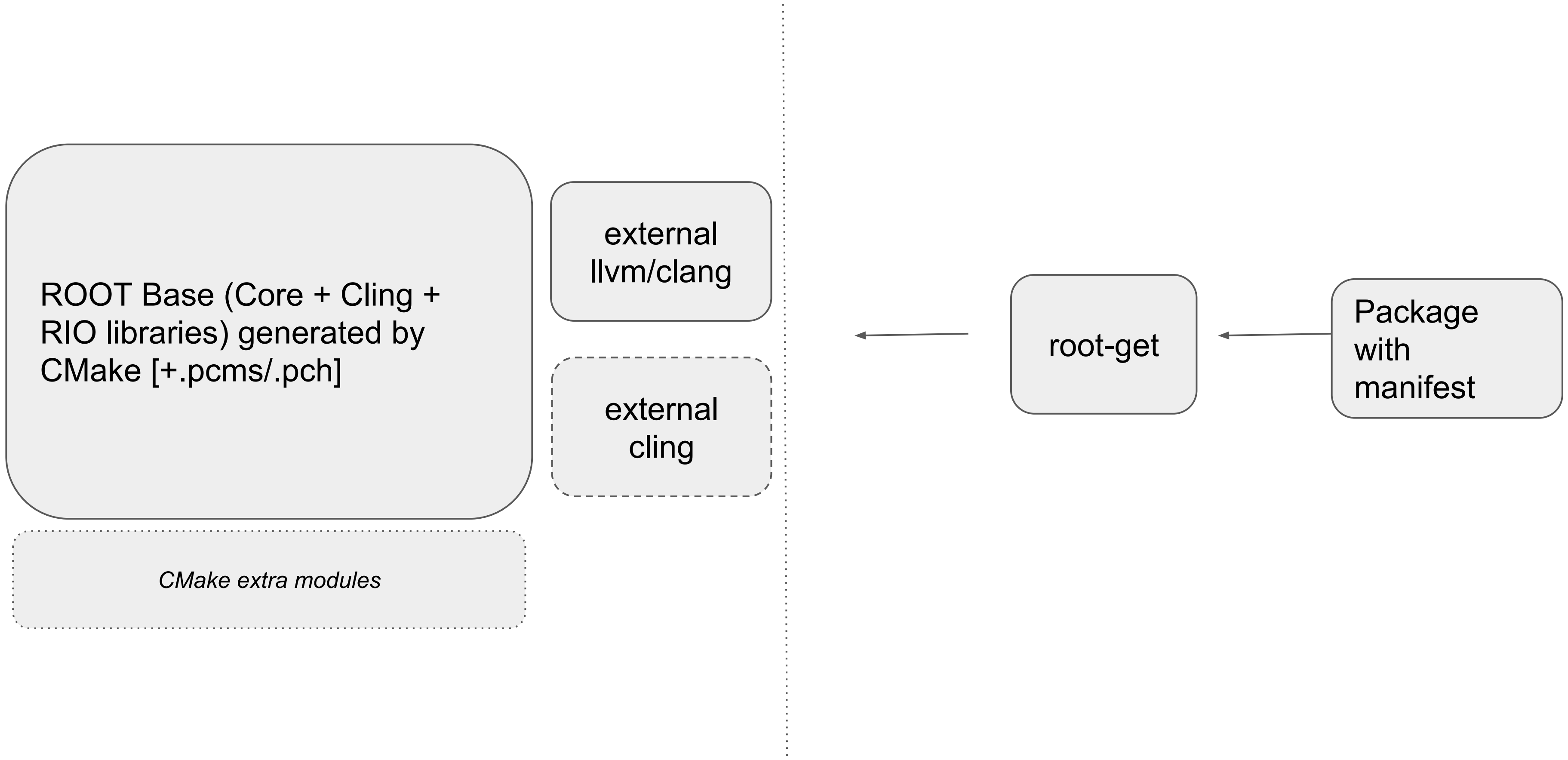
Project Dependency Manager (PDM) & “compiler, phase zero”

- 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.”

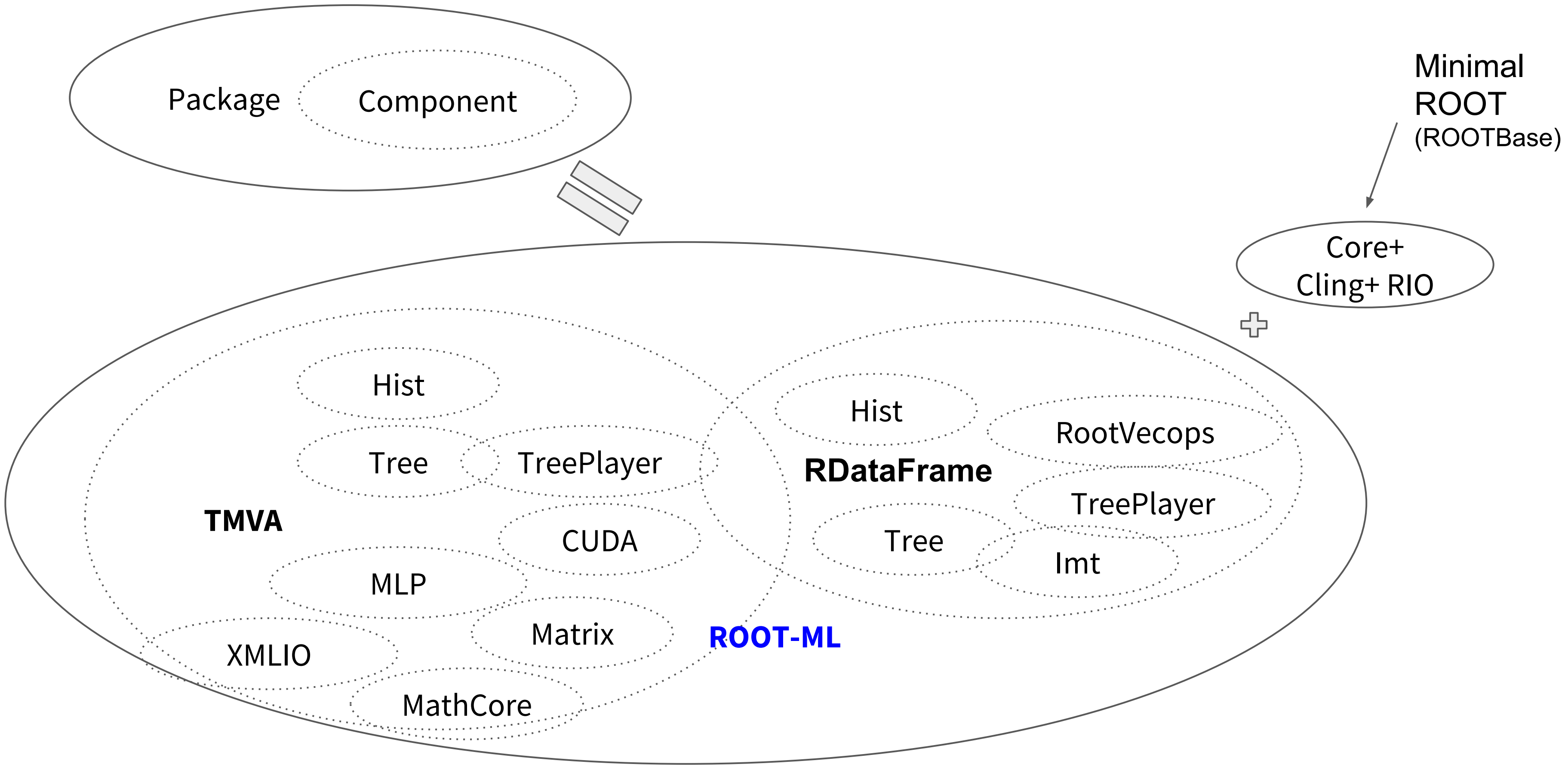
*“compiler, phase zero’ idea is similar to JIT’ing. In addition to putting the code on disk, the PDM typically needs to override the interpreter’s code loading mechanism in order to resolve includes correctly (PDM is producing a filesystem layout for itself to processed.)”

[1] https://www.tutorialspoint.com/compiler_design/compiler_design_phases_of_compiler.htm

Components of PDM

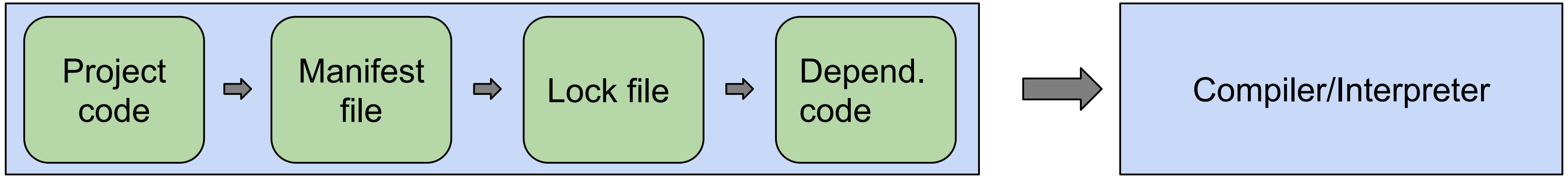


ROOT package & component



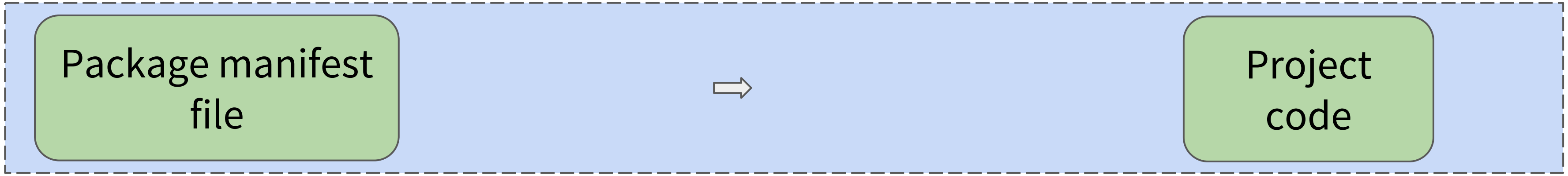
root-get

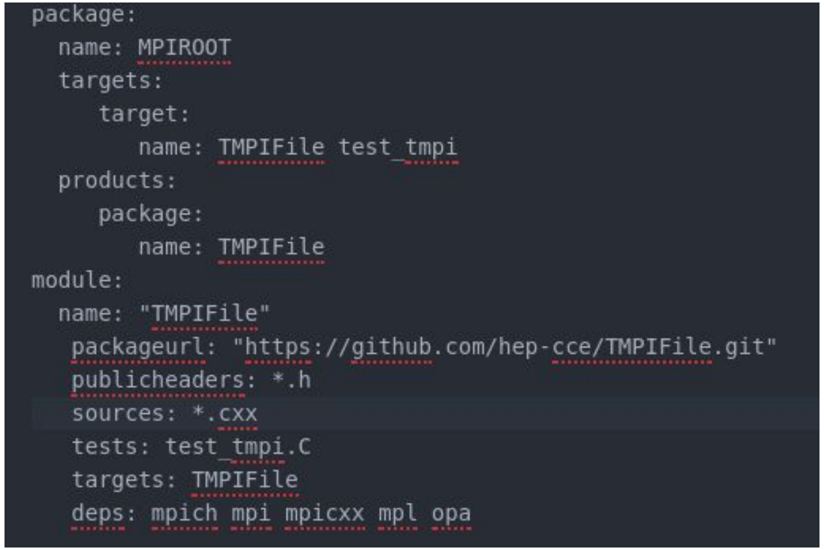
PM or PDM state of the art approach



- Eg. Swift Package Manager etc..
  - 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

Package manager flow (prototype)



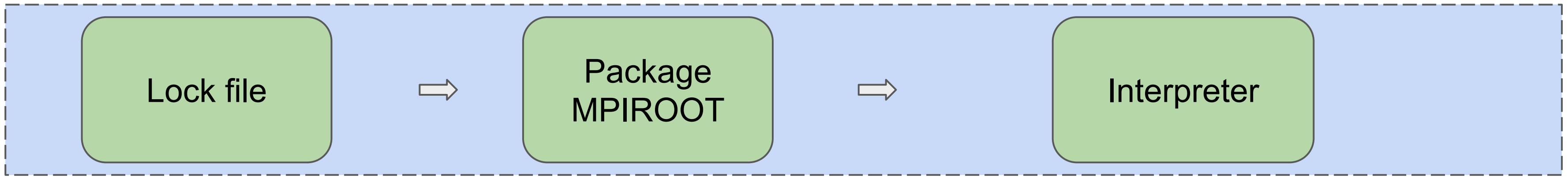


```
package:
  name: MPIROOT
  targets:
     target:
        name: TMPIFile test_tmpi
  products:
     package:
        name: TMPIFile
module:
  name: "TMPIFile"
   packageurl: "https://github.com/hep-cce/TMPIFile.git"
   publicheaders: *.h
   sources: *.cxx
   tests: test_tmpi.C
   targets: TMPIFile
   deps: mpich mpi mpicxx mpl opa
```

- *Defined by manifest file*

(example - external package https://github.com/hep-cce/TMPIFile)

Package manager flow (prototype)



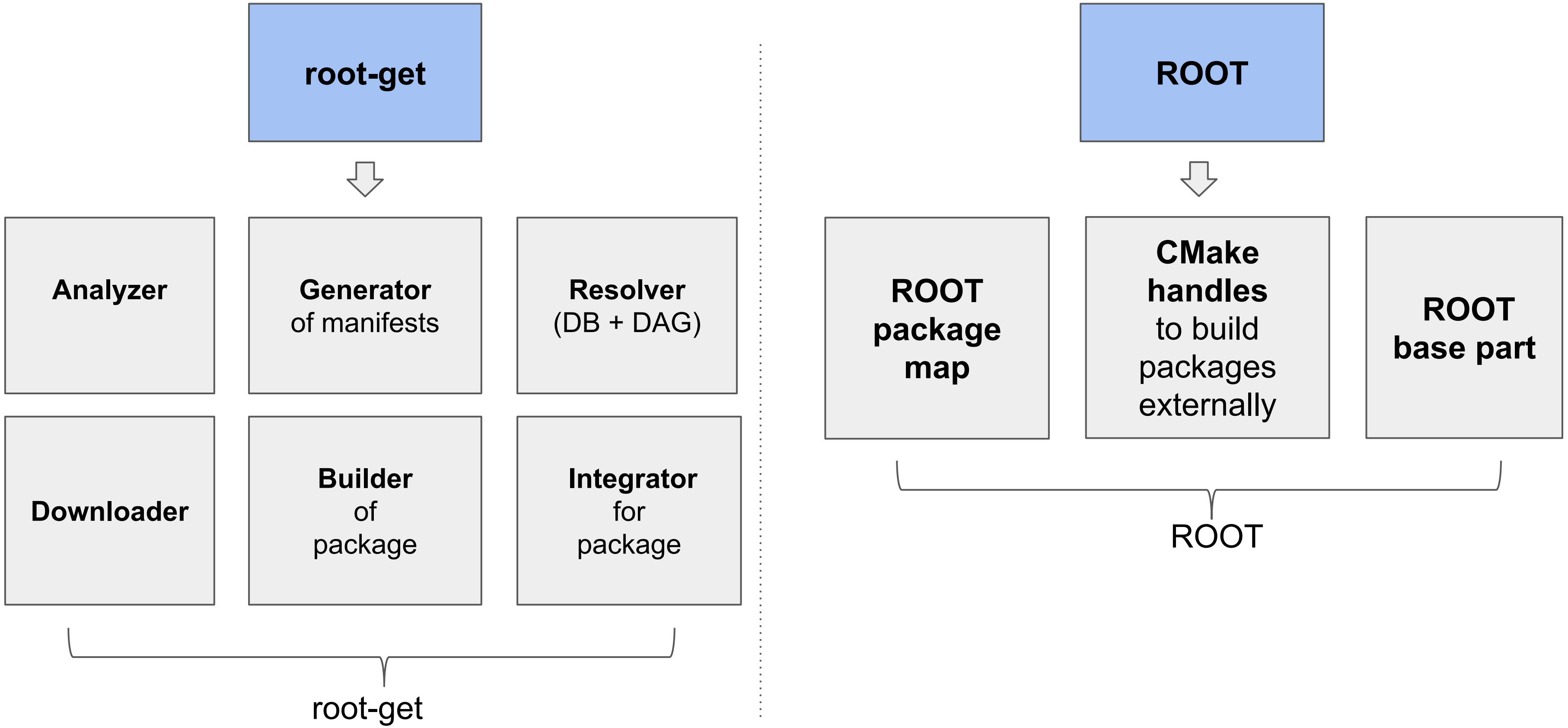
- Generating a dependency graph (DAG)
- Resolve dependencies via various strategies, listed in the project's manifest

Generating and deploying MPIROOT.zip:

- *inc/*
- *libTMPIFile.so*
- *License file*
- *manifest.yml*
- *TMPIFile_rdict.pcm*
- *TMPIFile.rootmap*
- *TMPIFile.pcm*

- 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.
- 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 (prototype)



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 pair of CMake files containing all information about ROOT macro and ROOT external dependencies
    - **We are running CMake to generate ROOT package manifests for ROOT**

ROOT PM: Resolver

- Having manifests files we can resolve package dependencies and update DB for root-get
  - We build and resolve DAG for all our packages
  - It will include management of handles for OS package managers, external builds and ROOT packages

ROOT PM: Builder

- Routine that allow root-get to build and/or test packages (we are invoking make/ninja and/or ctest at this step)

ROOT PM: Integrator

- Routine that install, provide packaging [into zip files] and root-get to deploying packages and its dependency code if needed

ROOT PM: Downloader

- Routine that allow root-get to work with external packages (represented for example only by a link in github)

Improvements in ROOT CMake

CMake improvements

- Help remove globbing of *.h and *.cxx and hide CXX flags in the ROOT_STANDARD_LIBRARY_PACKAGE() macro
- Separate the ROOT Base package (as well as other packages)
- Update ROOT options (separate them as “features” and “actual ROOT components” to be plugged)
- Update ROOT_STANDARD_LIBRARY_PACKAGE() - add_root_component()
- Ship distributed version of CMake setup macros such as add_root_component()
    - root-get add “include(RootFramework)” to each CMakeLists.txt to be able to digest add_root_component()

CMake improvements

- Change a way to discover all components based on enabled set of default ROOT components or extra components that they depend on;
- Add external_dependencies to add_root_component()
- Introduce ROOT package map
- Plug ROOT external map to ROOT package map
- Test separability

ROOT extra features: plug root-get to ROOT RT

CMake: 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:

```
[] #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.
```

Thank you!

FAQ

FAQ: We do not want another build system

- We do not want that either. Root-get’s major advantage will come when deploying binary artifacts.
- Currently we build ROOT libraries because ROOT is non-modular. All this happens in the context of cmake. Progress there is little due to various constraints: complexity of the cmake; long feedback/review loop; etc.

FAQ: Why do we need manifest files

- It is true that almost all of the information is already in cmake. In particular in ROOT_STANDARD_PACKAGE. Unfortunately, this information is available implicitly and it is very hard to understand it.
- Manifest files propose to make this information explicit, in a self-describing, textual file format (yaml or xml). They can contain information which cannot/should not be expressed in cmake. For example, artifact githuburl, or bugtrackerurl.
- If this information is expressed in easy to read and easy to parse textual file, generating build system rules becomes much easier. This approach is becoming more and more popular in modern build systems such as bazel.

FAQ: Why do we need manifest files

- Manifest files will allow us to simplify cmake code at the cost of introducing an JSON parser (relatively flat structure such as our proposal can be handled by regex).
- The long term benefit is that ROOT’s build information becomes build-system agnostic

FAQ: Why don’t we ‘fix’ ROOT_STANDARD_PACKAGE

- We could, as we have been doing last 2 years. Progress seems insufficient.
- Root-get introduces new requirements such as package matrices -- implementation requires more advanced language and library features such as maps. We could probably implement this, however, we believe we are going in the wrong direction and will introduce more technical debt than necessary.

CMake helpers

- https://github.com/toeb/cmakepp -- this seems to be what it would cost to implement stuff in cmake…
- We will need to pay price to maintain it

FAQ: How we should handle external dependencies

- Currently, cmake errors out saying this is not found (eg. x11, opengl). We can do the same but we want to do better.
- We should be able to delegate (after proper mapping) to the underlying system package manager (dpkg -L ‘dependency_header.h’). We can build on the experience of CMSSW and ALICE
- Example x11

FAQ: How we should handle versioning

- At first we want to support strict versioning (based on semantic versioning which is being well-adopted)
- Later, we can define the versioning as ranges of supported versions
- Ideally we would like to define ABI resilience mechanism which can find which components are commonly build together and loosen the requirements on specific versions (see Swift for more information on ABI resilience).

Backup

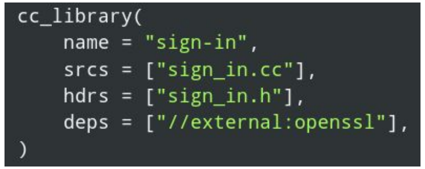
Pro & cons: meta build system CMake

- Pro:
  - Easy to write
  - Easy extensible
- Con:
  - CMake is not enough!
  - Problems:
    - How to manage dependency and its transitive dependency in CMake
    - How to package and deploy
    - Doesn’t have a support of C++ modules

*For third-party dependency: ExternalProject // Findxxx.cmake
*For solution of transitive problem - you need to write separate ExternalProject for each dependencies

Pro & cons: other build systems

- **Bazel**
  - Pro:
    - Bazel takes a lot of attention about external dependencies: https://bazel.build/roadmaps/external-deps.html
  - Con:
    - Complex syntax. Builds were slower than ninja. Transitive dependencies between external dependencies is messy.
- **Gn [meta-build system that generates build files for Ninja]**
  - Pro:
    - **it is extremely fast. Its syntax is extremely intuitive**
    - for every compiled file each build flag can be traced to specific build file and section. The compiler/linker/etc flags are organized into configs that can be easily enabled/disabled for each target.
    - the build is consistent on all platforms vs a lot of effort keeping consistent results with CMake where the build itself was performed by xcodebuild/msvc.
  - Con:
    - complex learning curve
    - complex toolchain definitions [tricky to adopt]



```
cc_library(
    name = "sign-in",
    srcs = ["sign_in.cc"],
    hdrs = ["sign_in.h"],
    deps = ["//external:openssl"],
)
```

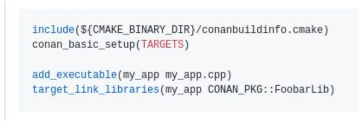
Pro & cons: dependency management. CMake package manager - Conan

Pro:

- One of not so many C++ dependency managers & package manager

Cons:

- Non-intrusive CMake integration is still intrusive by default
- “Doesn't seem as easy to add the new packages that you don't own, but doesn't seem that hard either”



```
include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
conan_basic_setup(TARGETS)

add_executable(my_app my_app.cpp)
target_link_libraries(my_app CONAN_PKG::FoobarLib)
```

Package managers (in particularly PDM)

	Package Manager	*PM's implementation language*
C++	Conan (PDM)	Python
Rust	Cargo	Rust
Swift	Swift Package Manager	Swift
Swift	Cocoapods (PDM)	Ruby
OS	Homebrew	Ruby
OS	Portage	Python
C++	Spack	Python

⇨ We are not be limited by choice of language!

We need to have a support for parsing & network operations such as download as well as for git manipulations

Separability for ROOT PM [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

Shipping ROOT C++ runtime modules

- 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?

Separate ROOTBase package (as well as other packages)

```
#---------------------------------------------------------------------------
# add_subdirectory(name)
# add_subdirectory(name)  was originally add_root_subdirectory(name)
#---------------------------------------------------------------------------
# Custom add_root_subdirectory wrapper
# Takes in the subdirectory name, and an optional
# path if it differs from the name.
macro(add_subdirectory name)
  # Make sure that we have a generic lower-case name of the module as an input
   canonicalize_tool_name(${name} nameLOWER)
  # Searching for available packages for the module
  # FIXME: now it is only one dimensional search, since we expect to have one module in one package
  # We need to extend it to have as a multi-dimensions search tool
………....
  foreach(list_name IN LISTS rootpackagemap_requested)
                foreach(value IN LISTS ${list_name})
                    if("${value}" STREQUAL "${name}")
                         …
endforeach()
if(EXISTS ${CMAKE_CURRENT_SOURCE_DIR}/${name}/CMakeLists.txt)
                _add_subdirectory(${CMAKE_CURRENT_SOURCE_DIR}/${name} ${name})
                        ….
                endif()
endmacro()
```

ROOTFeatures.cmake

```
# ROOT options that does not activate/deactivate packages/modules, but qualify the ROOT “build” feature
# There are only ~15 features in ROOT, all others are actually ROOT components

if (CMAKE_SYSTEM_PROCESSOR STREQUAL "aarch64")
   root_add_feature(ccache ON "Enable ccache usage for speeding up builds")
else()
   root_add_feature(ccache OFF "Enable ccache usage for speeding up builds")
endif()
# C++ standards options
root_add_feature(cxx11 ON "Build using C++11 compatible mode, requires gcc > 4.7.x or clang")
...
```

Add external_dependencies to add_root_component()

```
function(add_root_component libname)
….
        if(ARG_EXTERNAL_DEPENDENCIES)
          call(find_package_${ARG_EXTERNAL_DEPENDENCIES})
          string(TOUPPER "${ARG_EXTERNAL_DEPENDENCIES}" EXTERNAL_DEPENDENCY)
          set(EXTERNAL_FOUND "${EXTERNAL_DEPENDENCY}_FOUND")
          set(EXTERNAL_DEPENDENCIES_INCLUDES "${EXTERNAL_DEPENDENCY}_INCLUDE_DIR")
          #message(STATUS ${EXTERNAL_DEPENDENCIES_INCLUDES})
          set(EXTERNAL_DEFINITIONS "${EXTERNAL_DEPENDENCY}_DEFINITIONS}")
          if(NOT ${${EXTERNAL_FOUND}})
            message(FATAL_ERROR "Please try to install ${ARG_EXTERNAL_DEPENDENCIES}, we couldn't find using Find${ARG_EXTERNAL_DEPENDENCIES}.cmake
                  we expect to find ${EXTERNAL_FOUND}, ${EXTERNAL_DEPENDENCIES_INCLUDES} and ${EXTERNAL_DEFINITIONS}")
          endif()
          include_directories(${${EXTERNAL_DEPENDENCIES_INCLUDES}})
          add_definitions(${${EXTERNAL_DEFINITIONS}})
        endif()
endfunction()
```

io/CMakeLists.txt:

```
..
if(castor) # will be removed
  add_subdirectory(castor)
endif() # will be removed
..
```

io/castor/CMakeLists.txt:

```
add_root_component(RCastor
                   HEADERS TCastorFile.h
                   SOURCES src/TCastorFile.cxx
                   LIBRARIES ${CASTOR_LIBRARIES}
                             …………………………...
                   EXTERNAL_DEPENDENCIES castor
                   DEPENDENCIES Net RIO Core)
```

Change a way to search external libraries ROOTDependencies.cmake

```
...
function(find_package_unuran)
            message(STATUS "Looking for Unuran")
            find_package(Unuran)
            if(NOT UNURAN_FOUND)
                find_package_builtin_unuran()
            endif()
endfunction(find_package_unuran)
...
```

Search layering:

- *FindXXX.cmake*
- *OS package manager query (based on header search) + version info*
- *Build based on provided manifest +/ header search + semantic versioning*

ROOT external map

- How to divide a ROOT into packages and modules?
  - We introduced ROOTComponentMap.cmake to help us with this task (statically and dynamically)
- We would like to provide a possibility to plug your own map of packages and modules!

Custom_map.yml

```
ML:
   MathCore, MathMore, PyRoot, TMVA, CustomDiplomaOfPHDStudentLib
```

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

(only what will be needed in addition - it is a manifest file of CustomDiplomaOfPHDStudentLib)

ROOTPackageMap.cmake

```
# Initial ROOT package map [dynamic] - minimal requirement is ROOTBase package
set(rootpackagemap_requested BASE)
SET_PROPERTY(GLOBAL PROPERTY rootpackagemap_requested "")

# Package map with all existing ROOT packages, package ALL is trying to build whole set of ROOT libraries
set(rootpackagemap BASE ALL IO)
SET_PROPERTY(GLOBAL PROPERTY rootpackagemap "")

# Define packages
set(BASE interpreter llvm/src cling core io rootpcm main clib clingutils cont dictgen foundation meta metaclin rint textinput thread imt zip lzma lz4
newdelete unix winnt macosx base rootcling_stage1 src)
SET_PROPERTY(GLOBAL PROPERTY BASE "")

# We could try to add package to map
function(ADD_ROOT_PACKAGE_TO_MAP X)
   set(rootpackagemap_requested ${rootpackagemap_requested} ${X})
endfunction()
```

ROOTFramework.cmake

- Distributed version of CMake setup macros such as add_root_component()
- root-get add “include(RootFramework)” to each CMakeLists.txt to be able to digest add_root_component()

Test separability

Vassil: building ROOT base and testing Core and removing all tests which is implicitly depending on other components = introduce layering in testing

```
# ROOT targets that we need for these test: Hist  & Tree

add_test(NAME event-generate   COMMAND ...)
add_test(NAME event-cleanup COMMAND ...)
add_test(NAME event-cms1  COMMAND ...)
add_test(NAME event-cms2   COMMAND ...)

set_tests_properties(event-cms1 event-cms2 PROPERTIES FIXTURES_REQUIRED “Hist;Tree”)
set_tests_properties(event-generate PROPERTIES FIXTURES_SETUP    “Hist;Tree”)
set_tests_properties(event-cleanup PROPERTIES FIXTURES_CLEANUP “Hist;Tree”)
```

```
# We will also need to have exported a full list of ROOT targets for roottest
```