Migrating large codebases to C++ Modules

Yuka Takahashi - The University of Tokyo
Princeton University
Oksana Shadura - UNL
Vassil Vassilev

Agenda

1. Motivation of C++ Modules
2. C++ Modules in ROOT
3. C++ Modules in CMSSW
4. CMS Performance Results
5. Conclusion

Motivation of C++ Modules

Motivation of C++ Modules

C++ Modules technology:

- Cache parsed header file information
  - Avoid header re-parsing
  - Avoid runtime header parsing (In ROOT)
- Part of C++20

Motivation of C++ Modules

```
#include <vector>
```

Motivation of C++ Modules

```
#include <vector>
```

Textual Include	Precompiled Headers (PCH)	Modules
✗ Expensive ✗ Fragile	✗ Inseparable	✓

Motivation of C++ Modules

```
#include "TVirtualPad.h"
#include <vector>
#include <set>

int main() {
…
```

original code

Preprocess

Textual Include

```
                                        ……
                                        ……
# 286 "/usr/include/c++/v1/vector" 
namespace std { inline namespace __
template <bool> class __vector_base
{
   __attribute__
((__visibility__("hidden"),
  always_inline__)) __vector_base_c
                                        ……
# 394 "/usr/include/c++/v1/set" 3
namespace std {inline namespace __1
template <…> class set {
public:
    typedef _Key key_type;
                                        ……

int main {
                                        ……
```

TVirtualPad.h

mmon

vector

on()

set

one big file!

Parse

Compile

.o

Motivation of C++ Modules

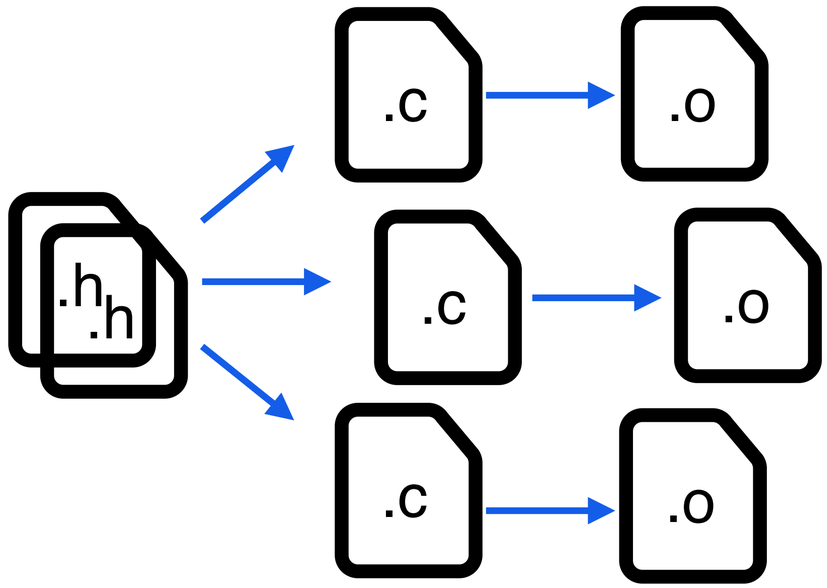
Textual Include

1. Expensive

   Reparse the same header

2. Fragile

   Name collisions



Rcpp library

```
#define PI 3.14
…
```

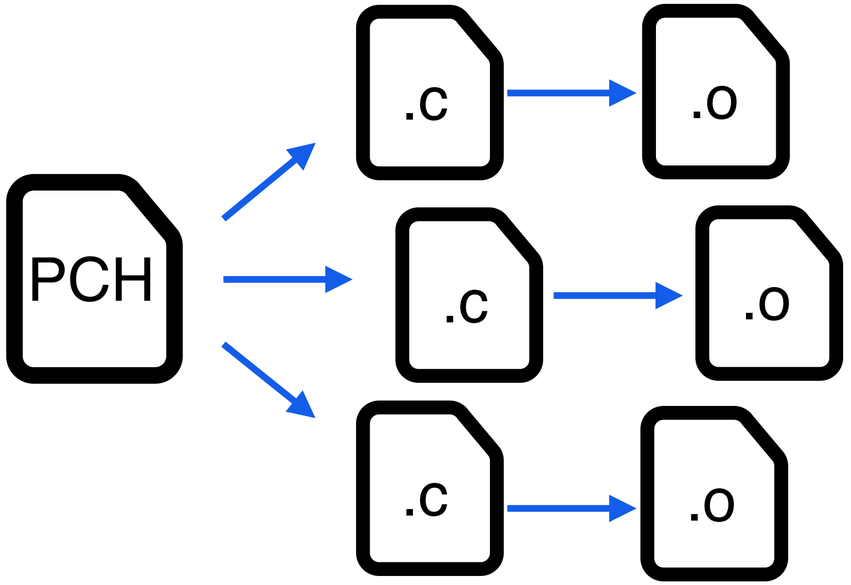
Users' code

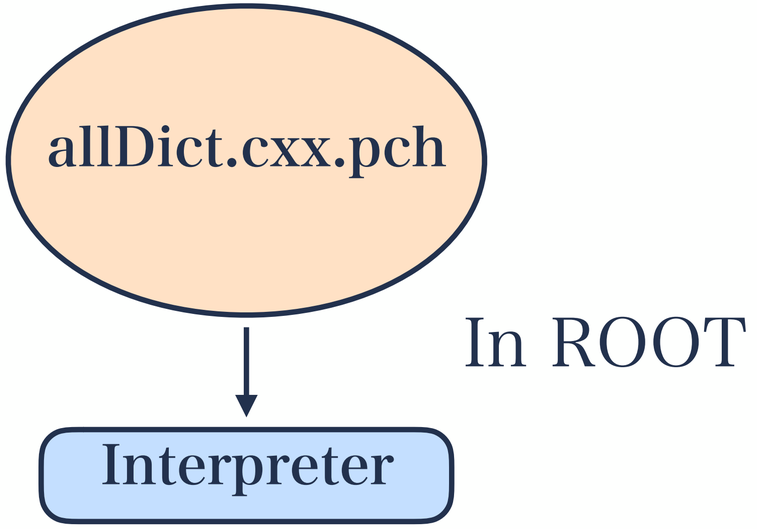
```
#include <header.h>
                …
double PI = 3.14;
// => double 3.14 = 3.14;
```

Motivation of C++ Modules

PCH (Precompiled Headers)

1. Storing precompiled header information (same as modules)
2. Stored in one big file
   - Monolithic



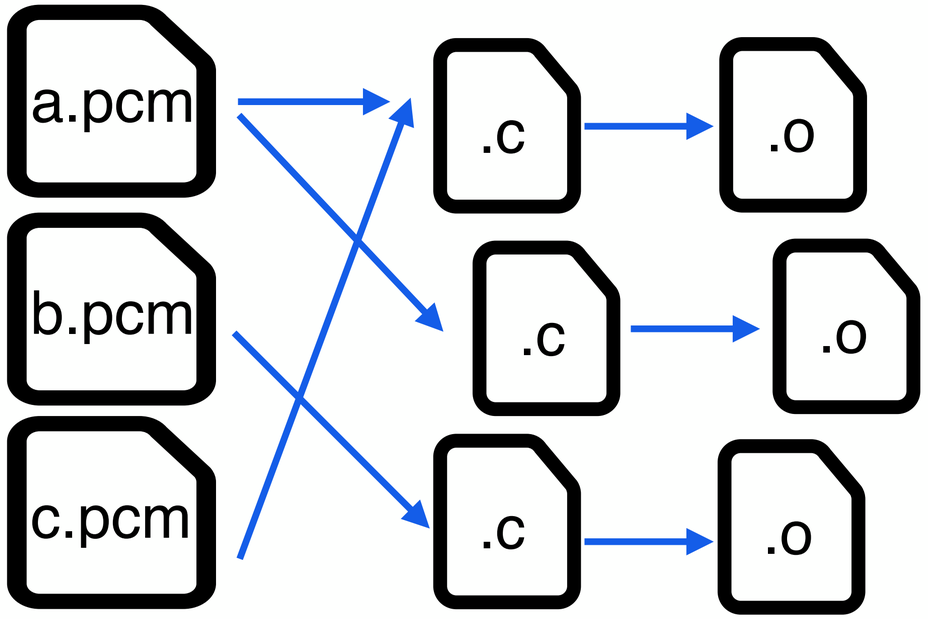


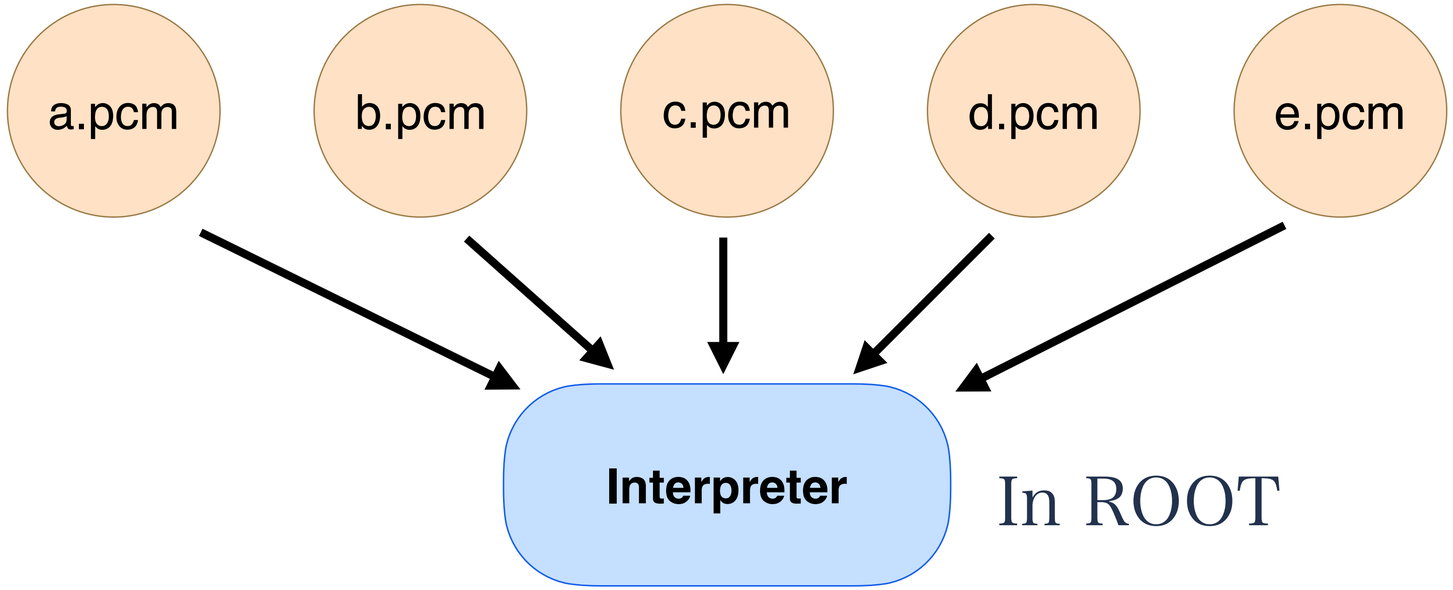
Motivation of C++ Modules

Modules

- Module **files** contain parsed header information
- PCMs are **separated**

Each PCM file (a.pcm) corresponds to a library (liba.so)

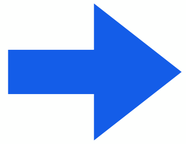




Motivation of C++ Modules

Modules

- Modules **files** contain parsed header information
- PCMs are **separated**



✓ Compile-time scalability

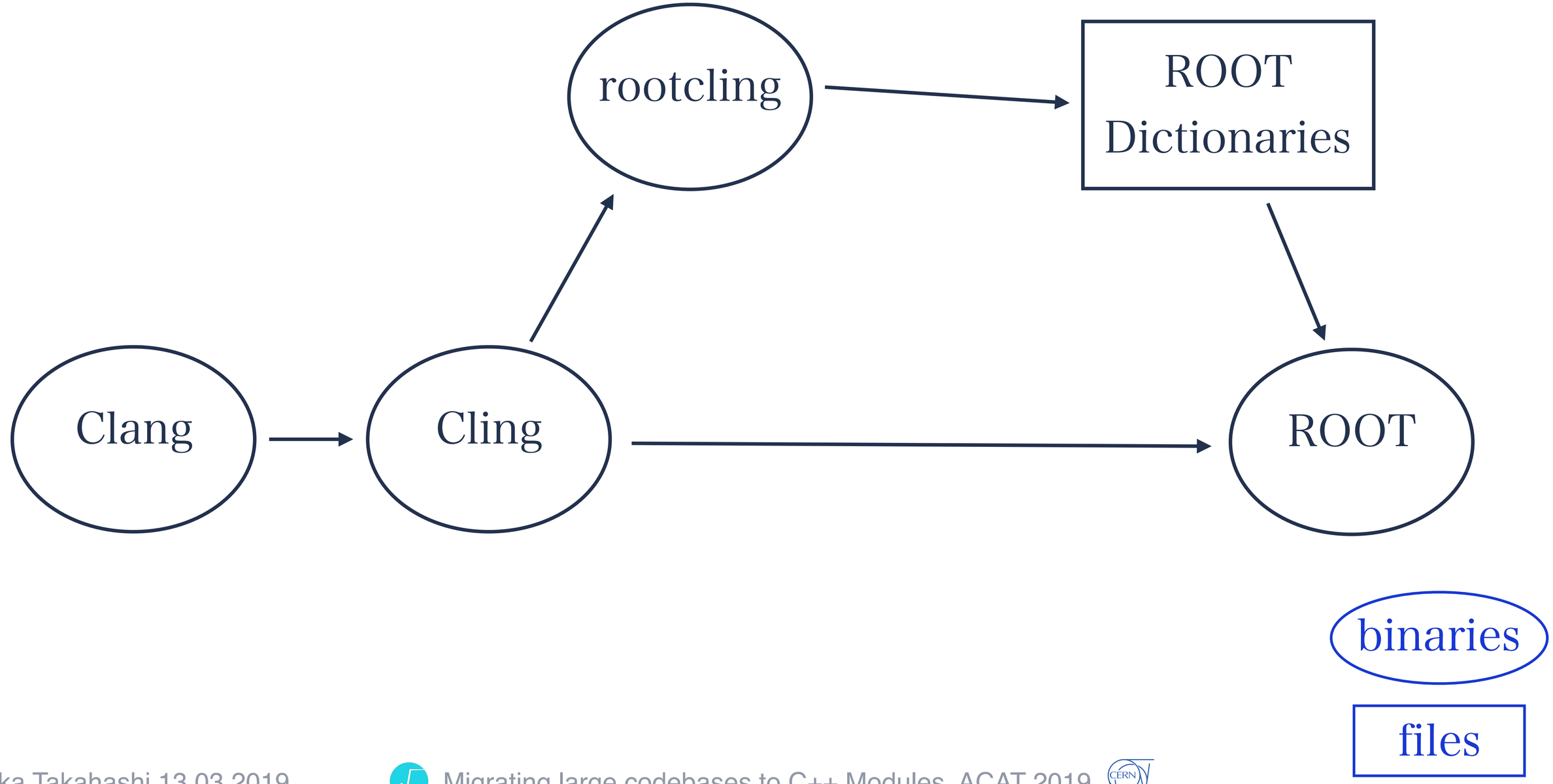
✓ No Fragility

✓ Separable

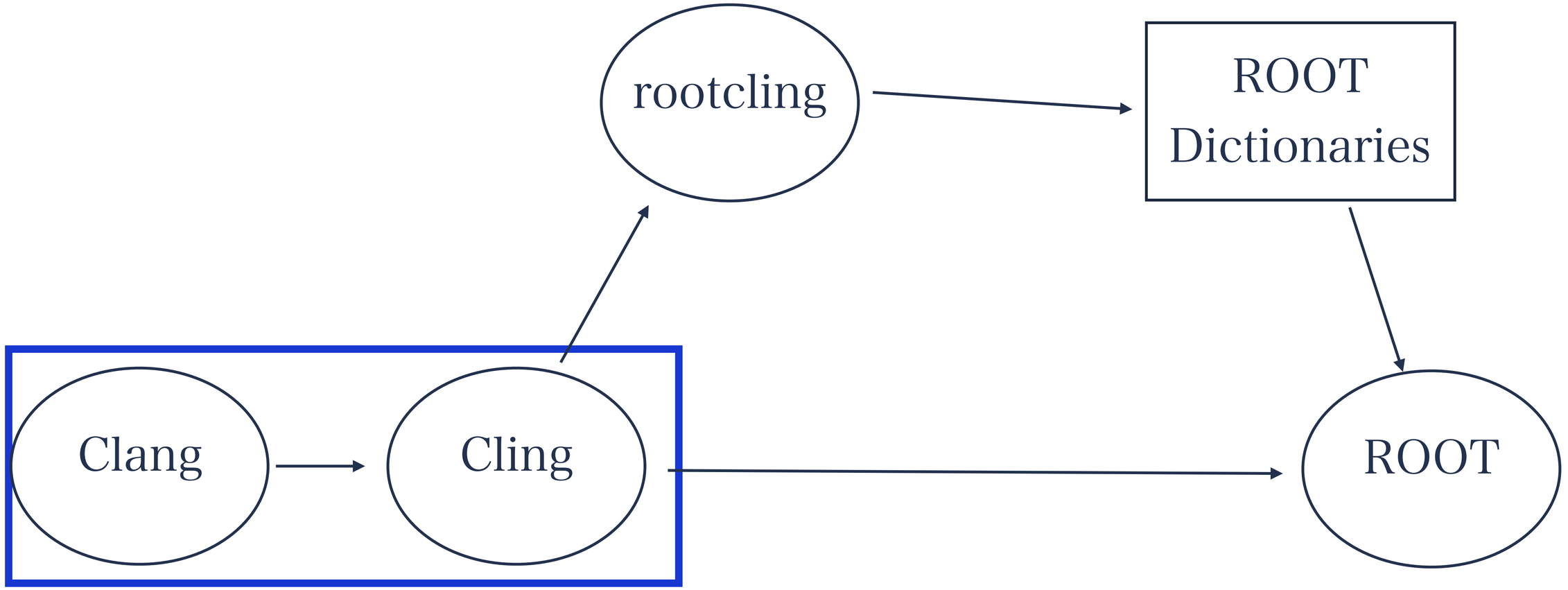
C++ Modules in ROOT

Technology Preview released in ROOT 6.16

Overview - Dependency Graph



Overview - Dependency Graph

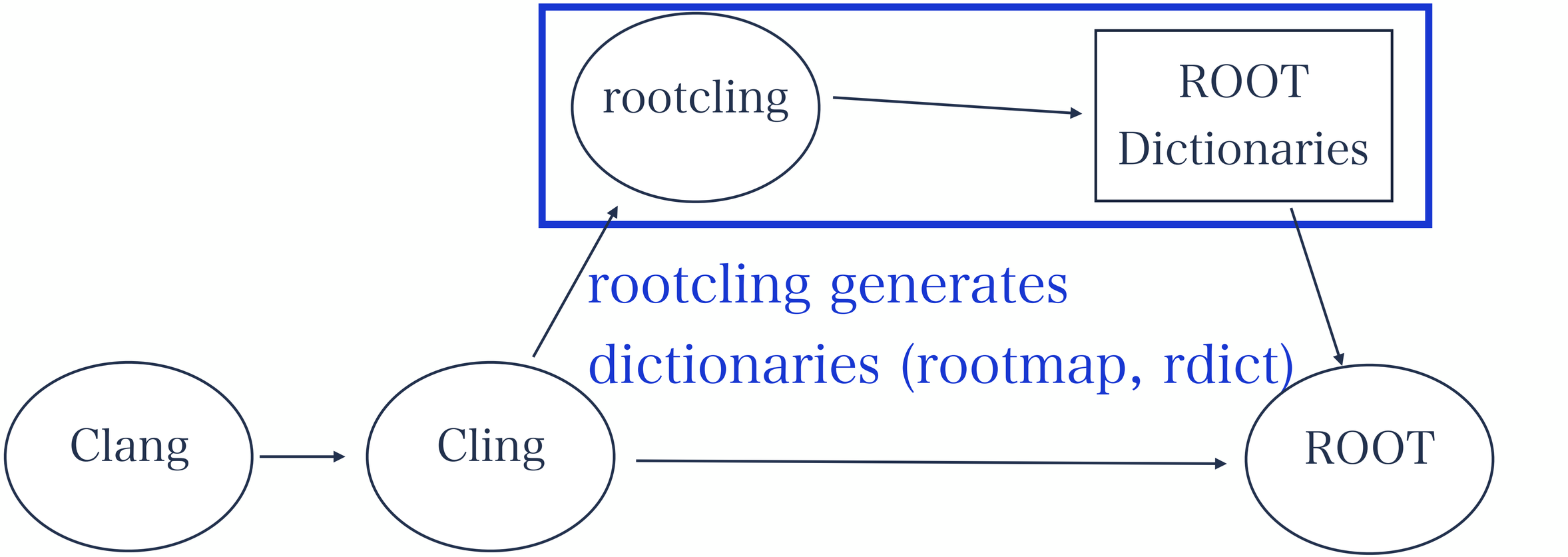


Cling calls Clang API

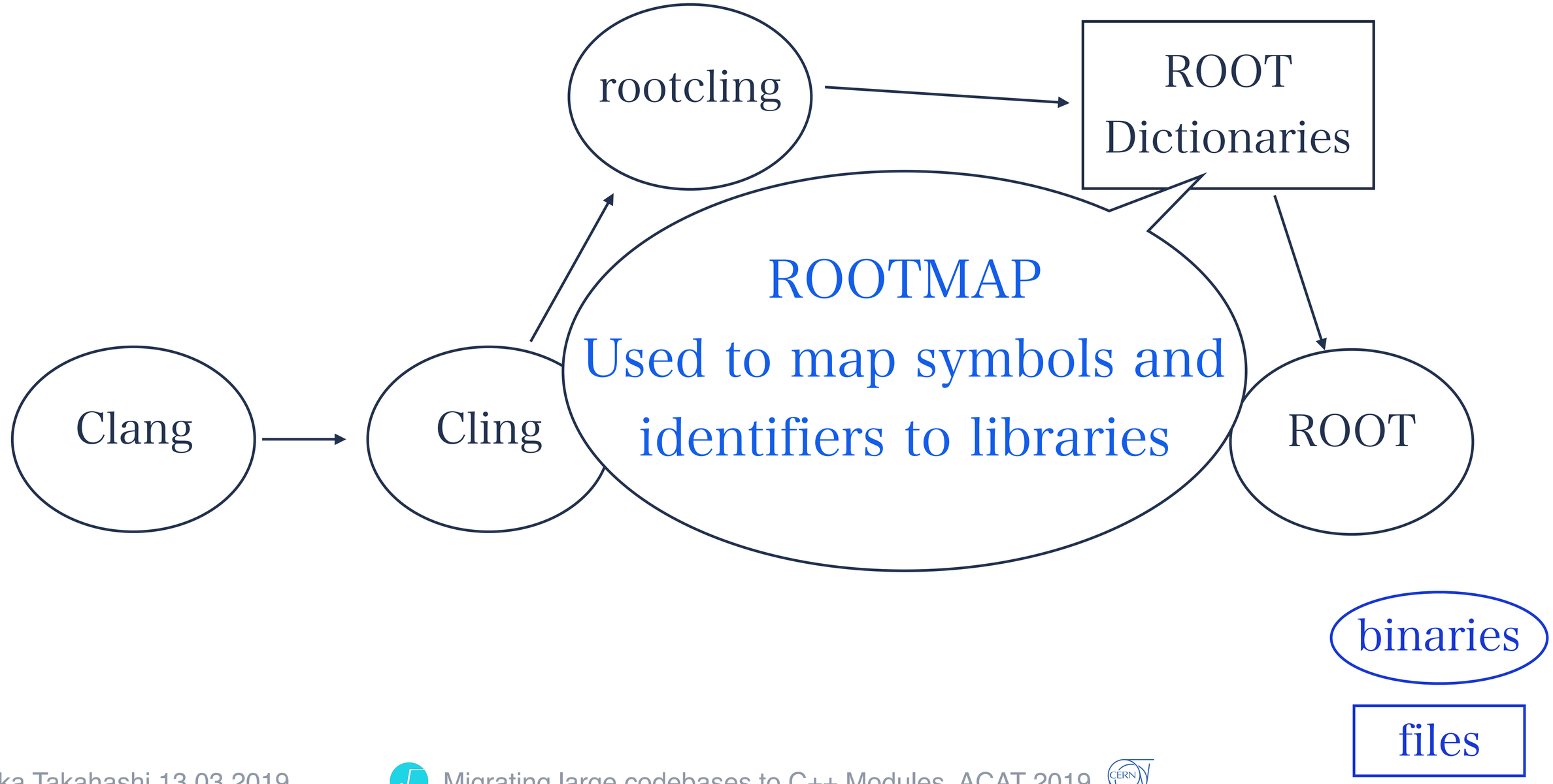
binaries

files

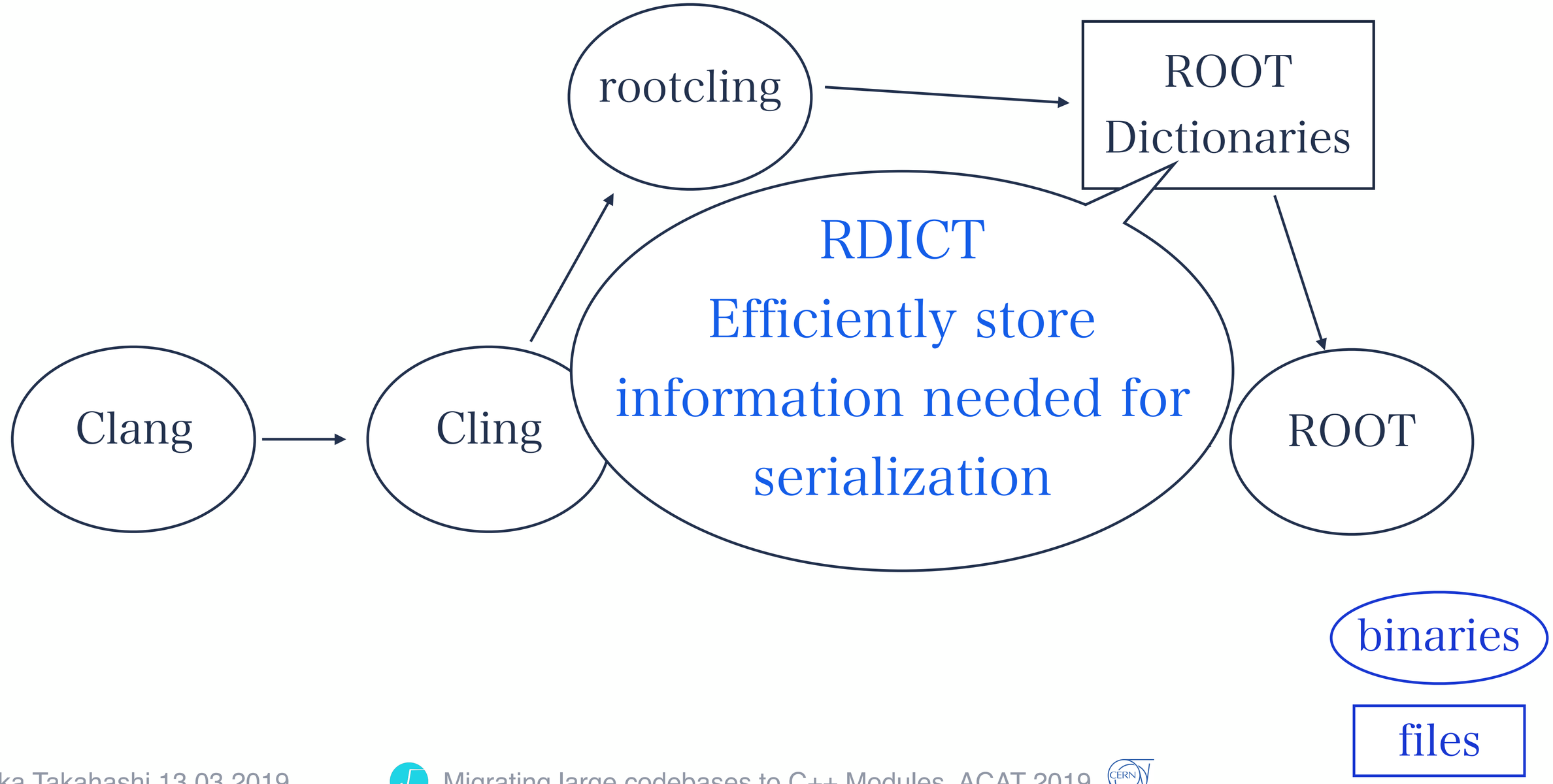
Overview - Dependency Graph



Overview - Dependency Graph

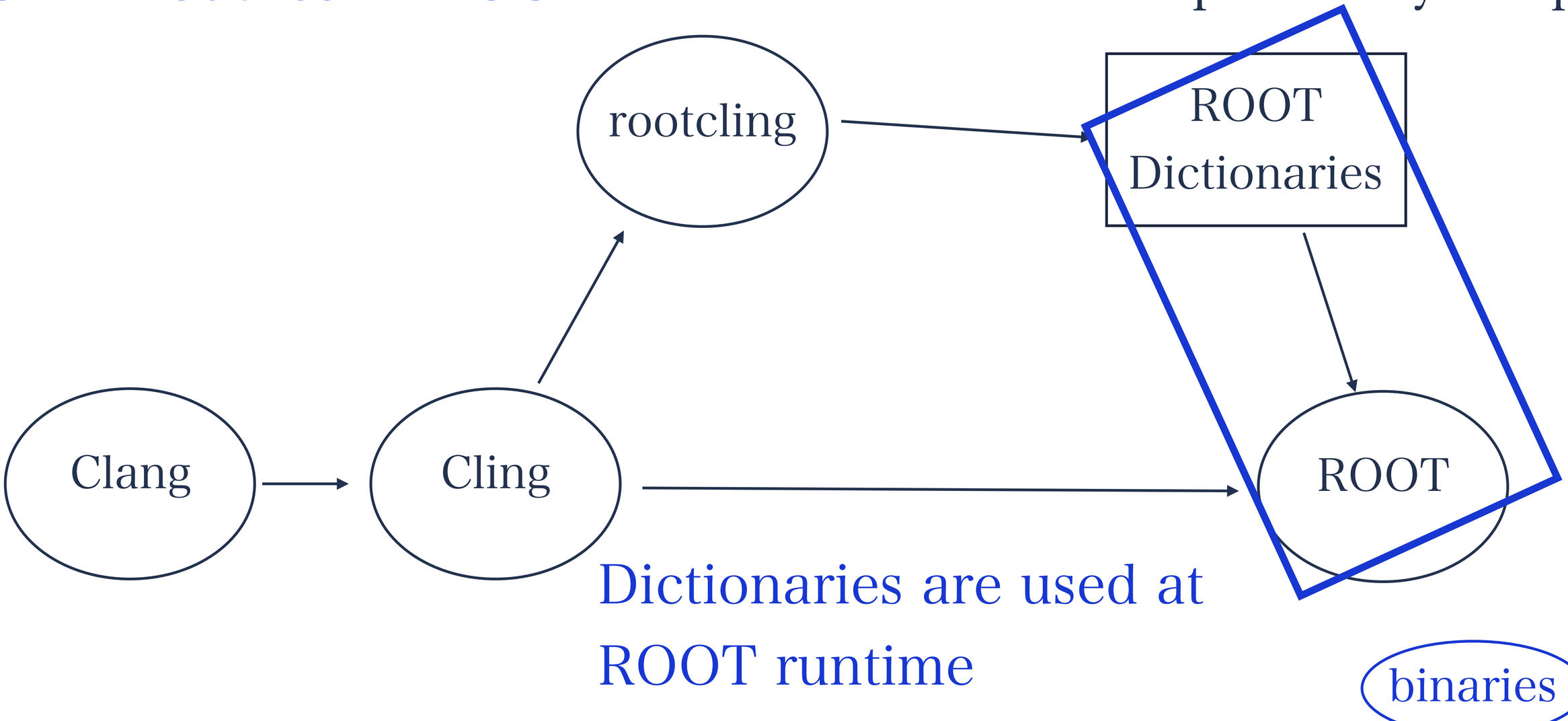


Overview - Dependency Graph

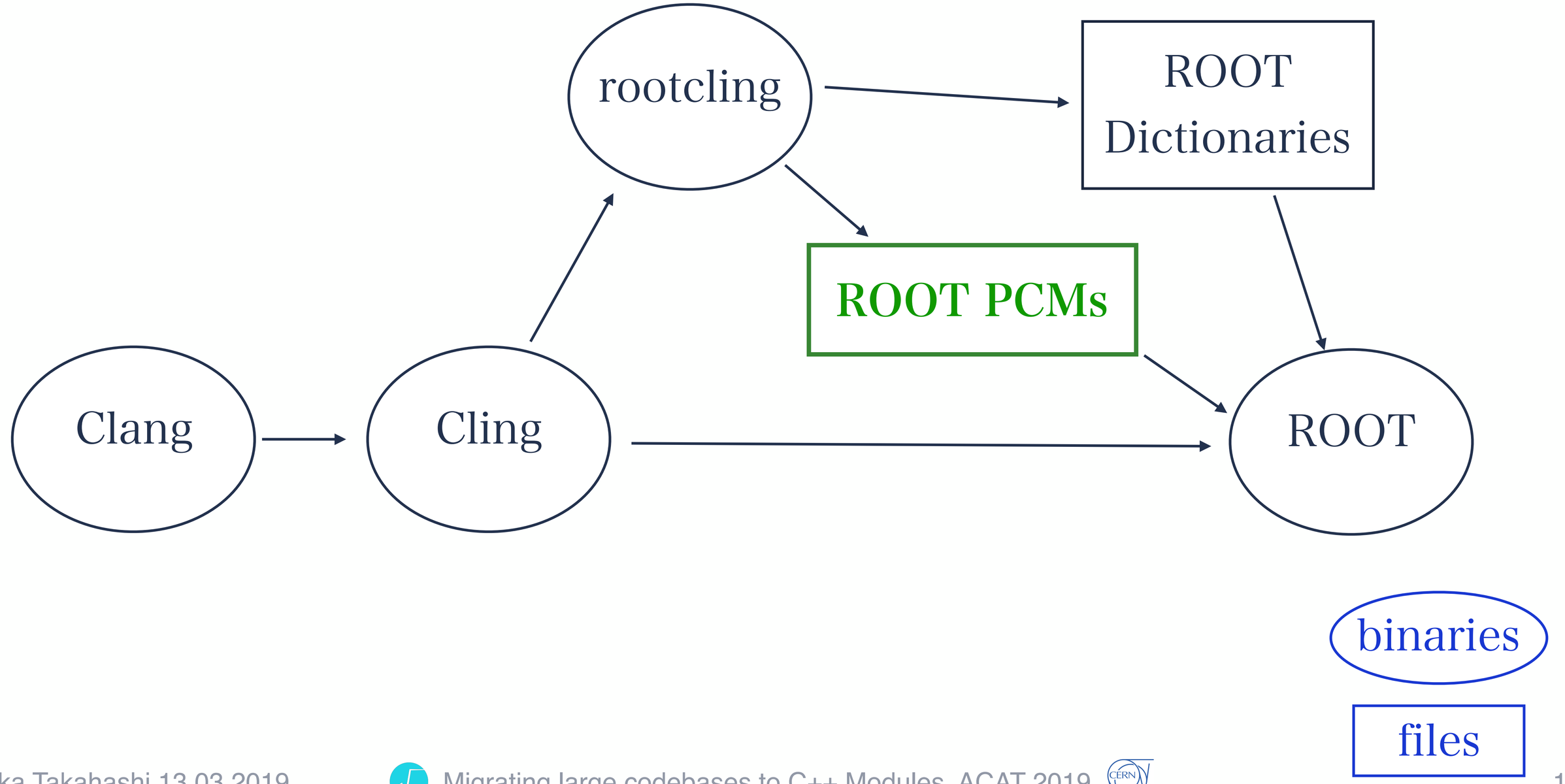


C++ Modules in ROOT

Overview - Dependency Graph

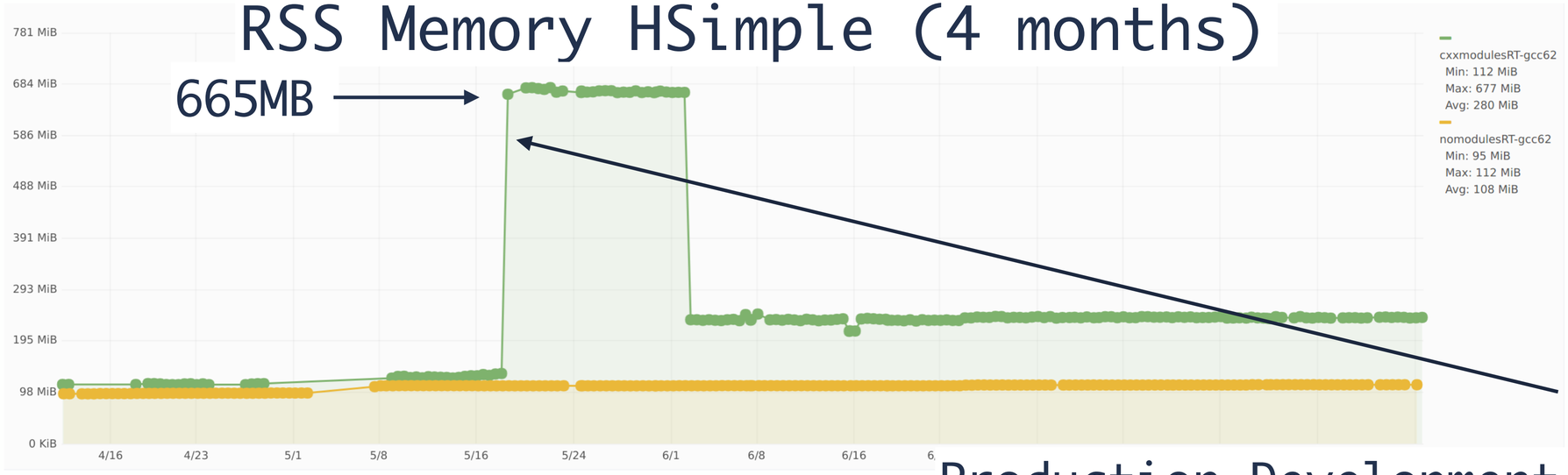


Overview - Dependency Graph



Preloading of modules

- Replace some functionality of RDICT and ROOTMAP with a more stable implementation
- Load all ROOT modules at the startup time



http://root-bench.cern.ch (login required)

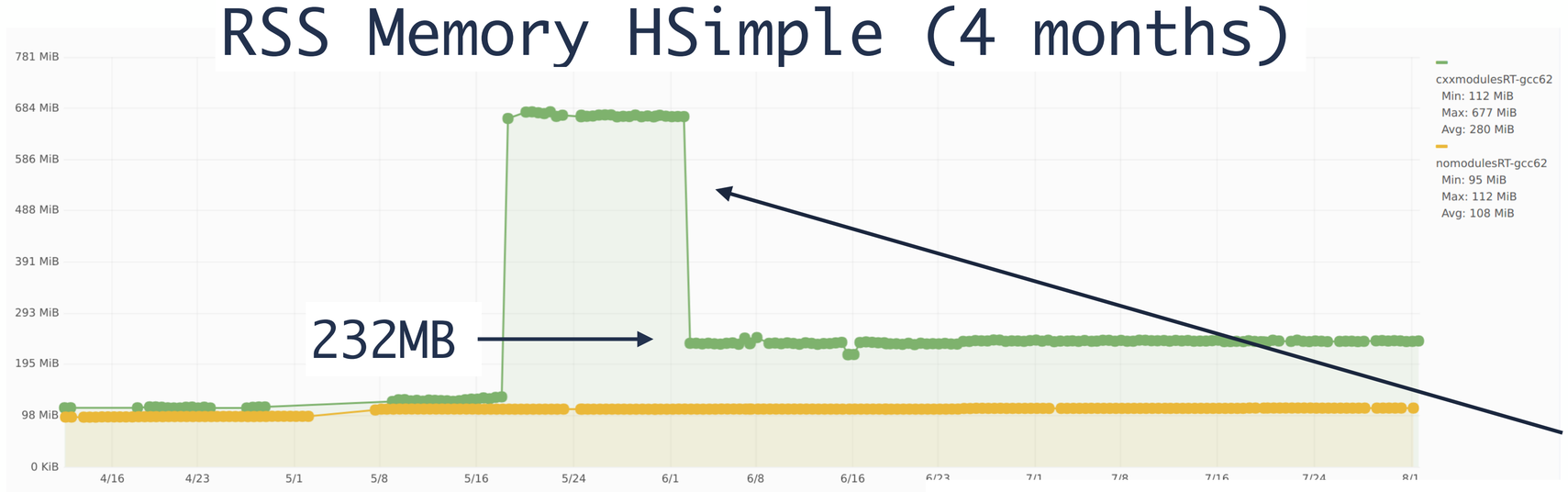
Preloading of all modules

Global Modules Index

- Remove further overhead in ROOT
- Mechanism to create the table of symbols and PCM names
  - ROOT will be able to load corresponding library when a symbol lookup failed
  - The prototype shows promising results

Bloom filter

- Hash tables of symbols in .gnu.hash section in shared object files (further read)
- ROOT can skip unnecessary libraries by reading it



Bloom filter

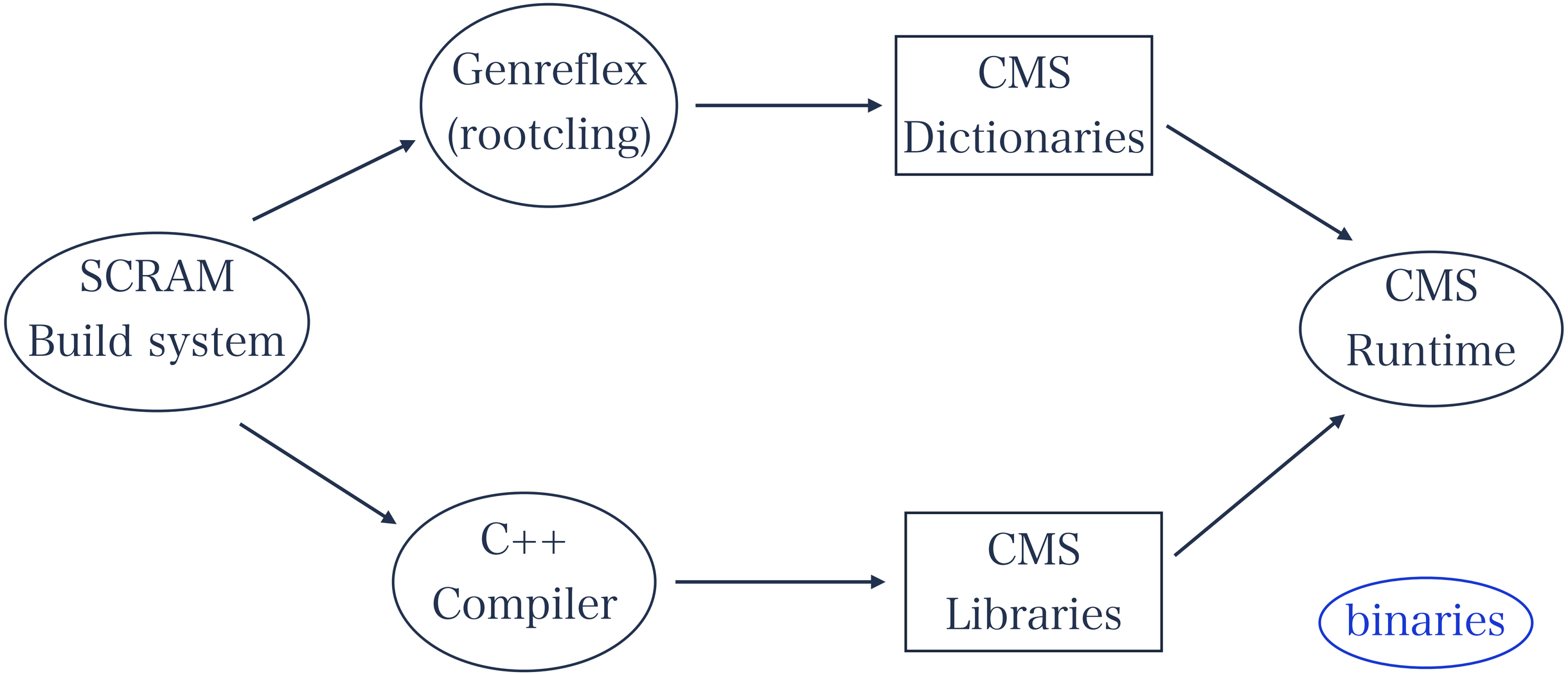
http://root-bench.cern.ch (login required)

Production Development

C++ Modules in CMSSW

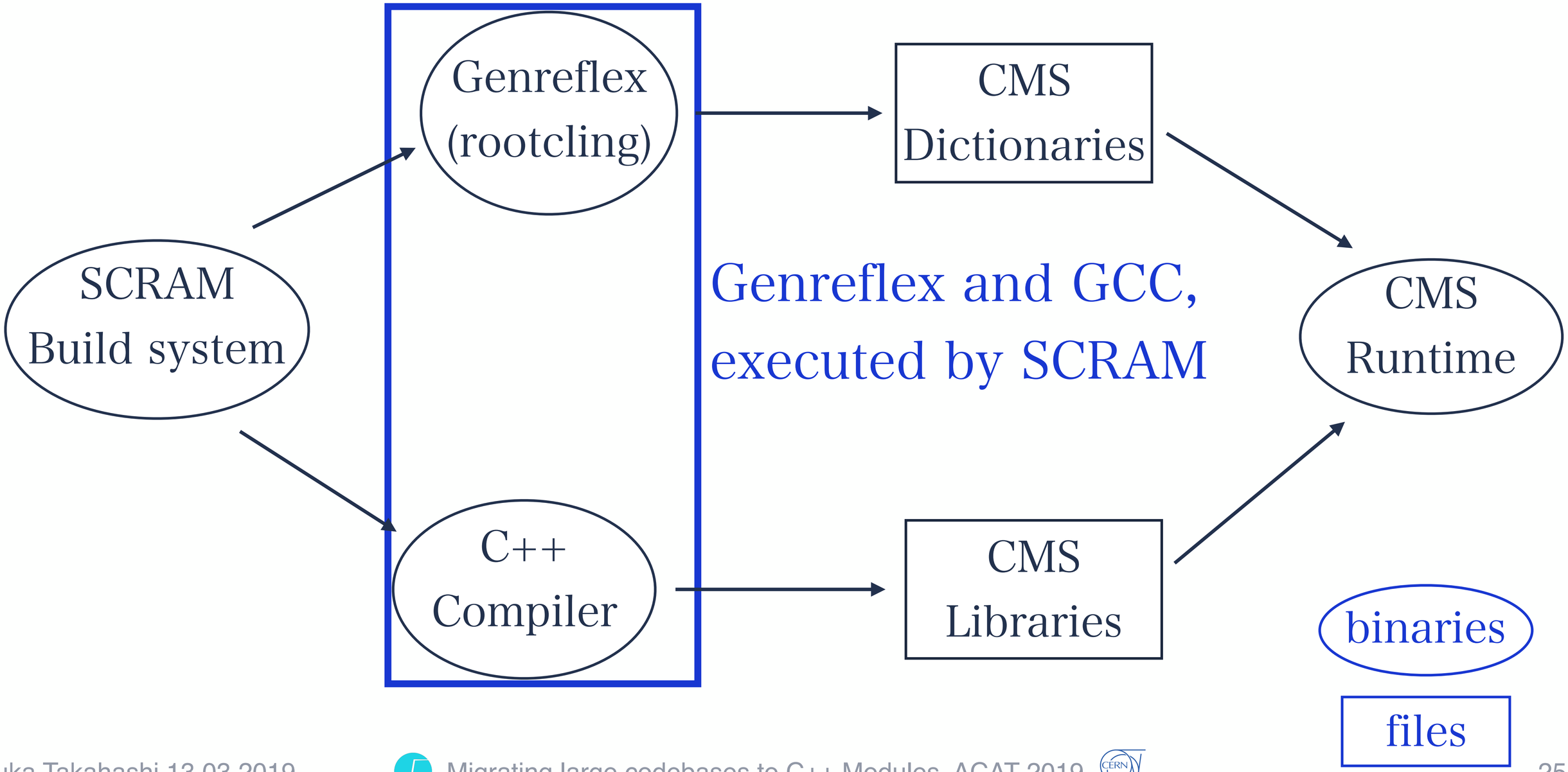
Available in CMS CXXMODULE IB

Overview - Dependency Graph

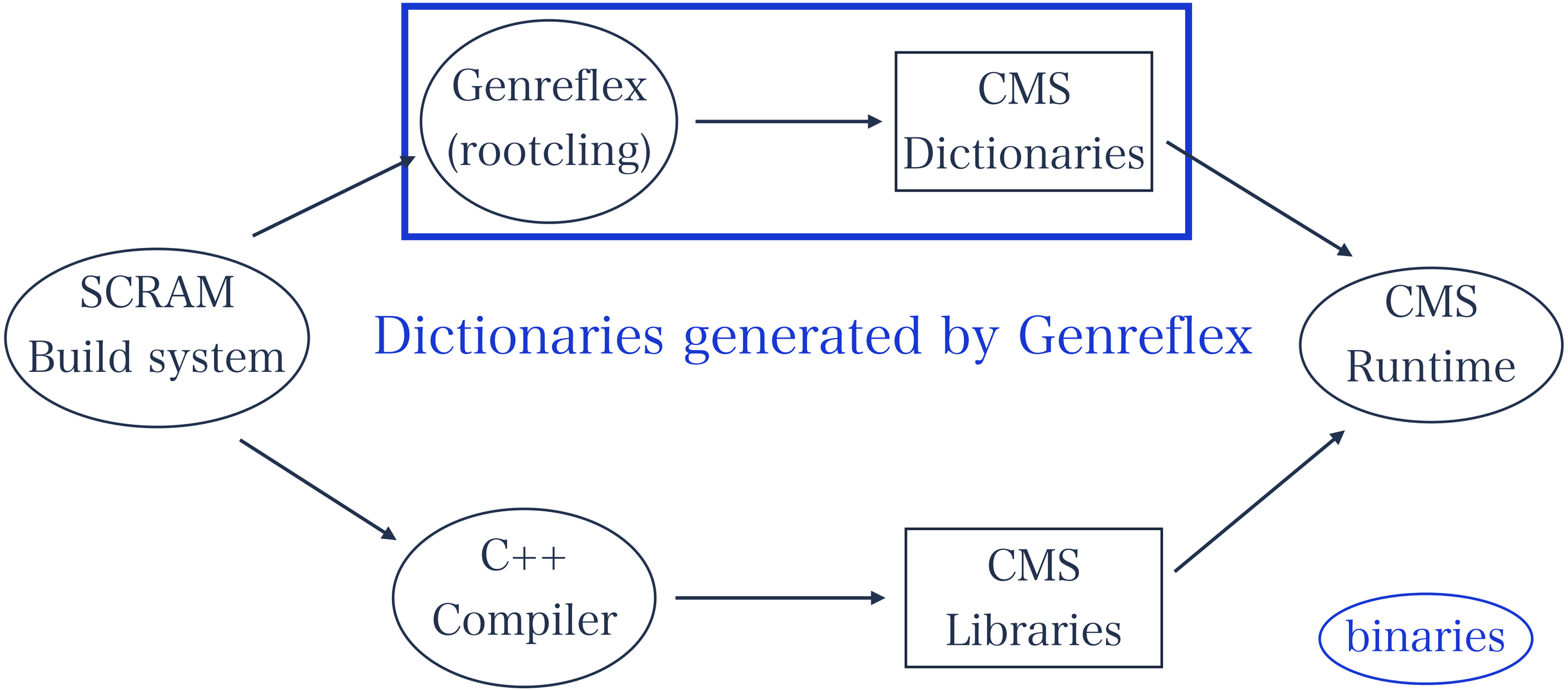


C++ Modules in CMSSW

Overview - Dependency Graph

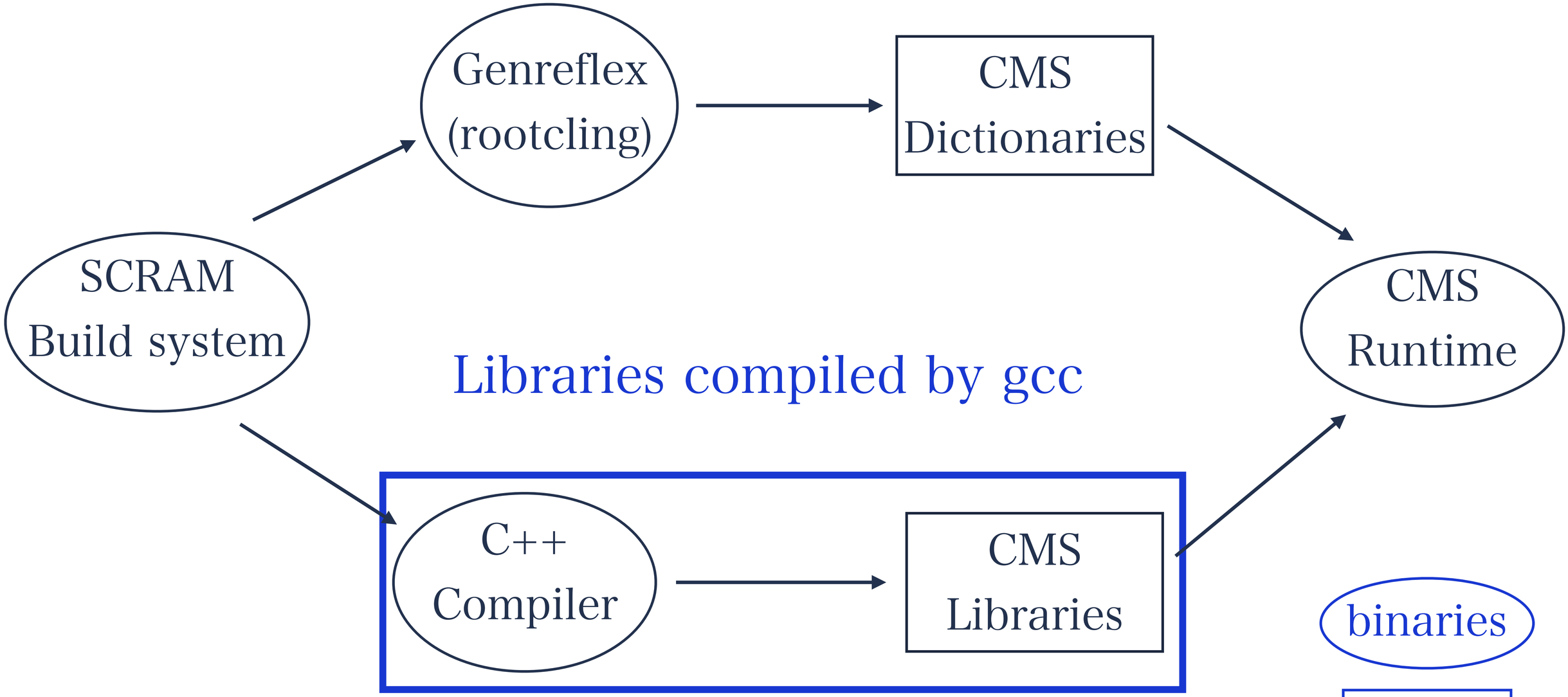


Overview - Dependency Graph

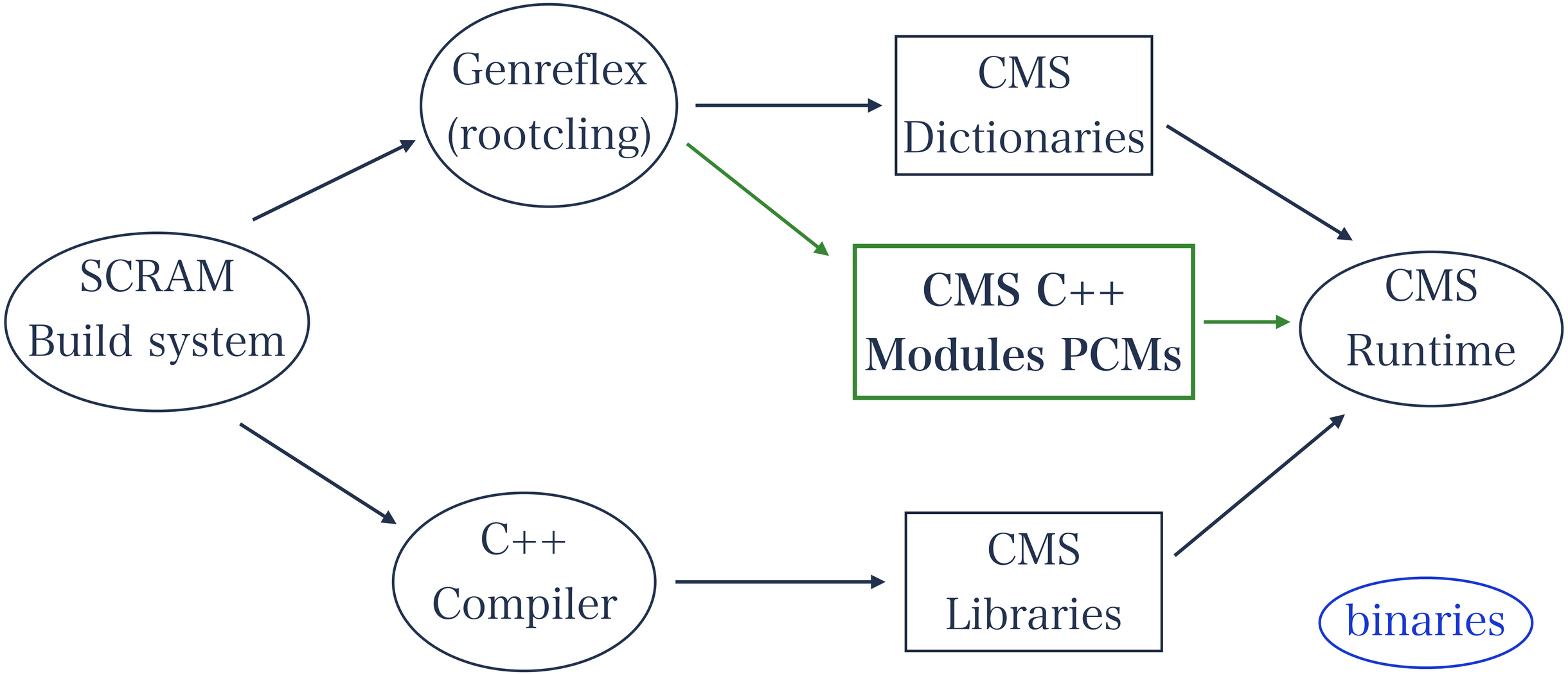


C++ Modules in CMSSW

Overview - Dependency Graph

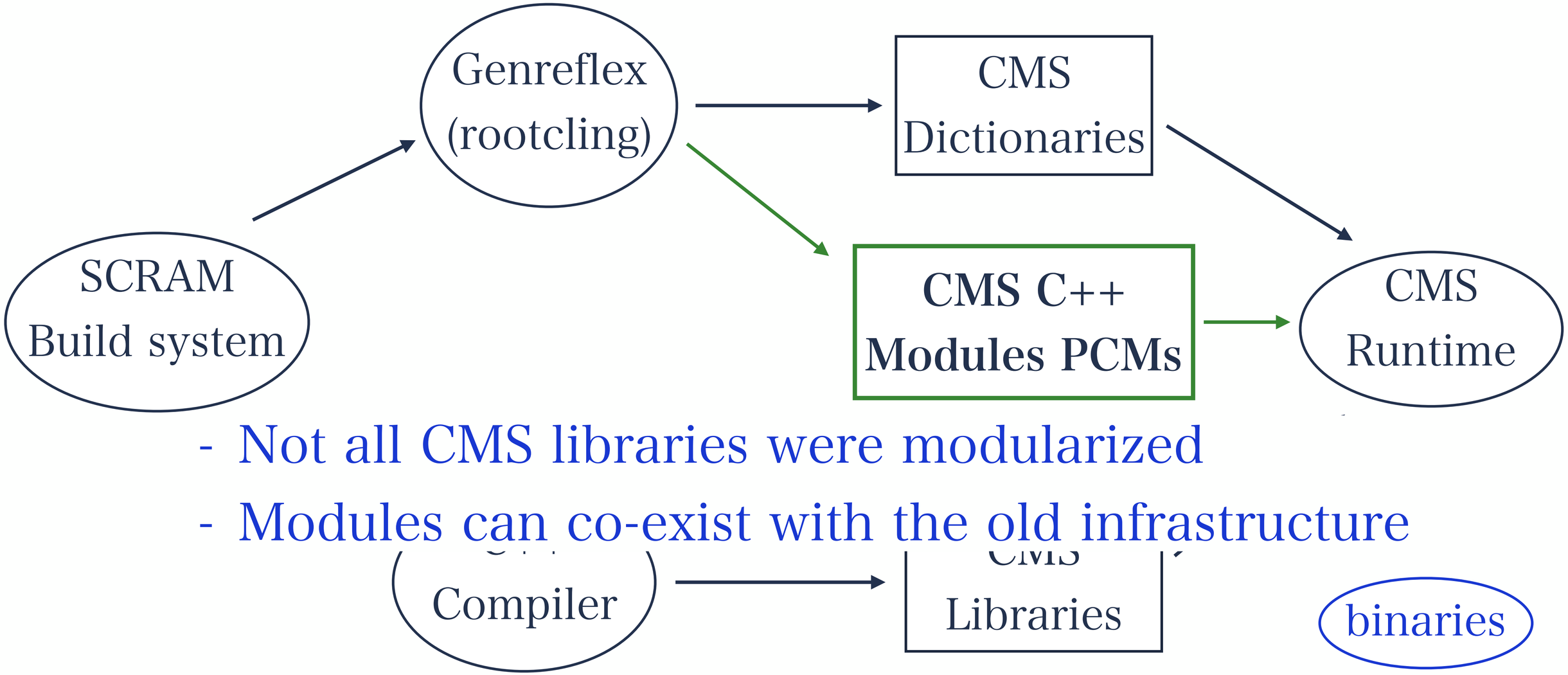


Overview - Dependency Graph



C++ Modules in CMSSW

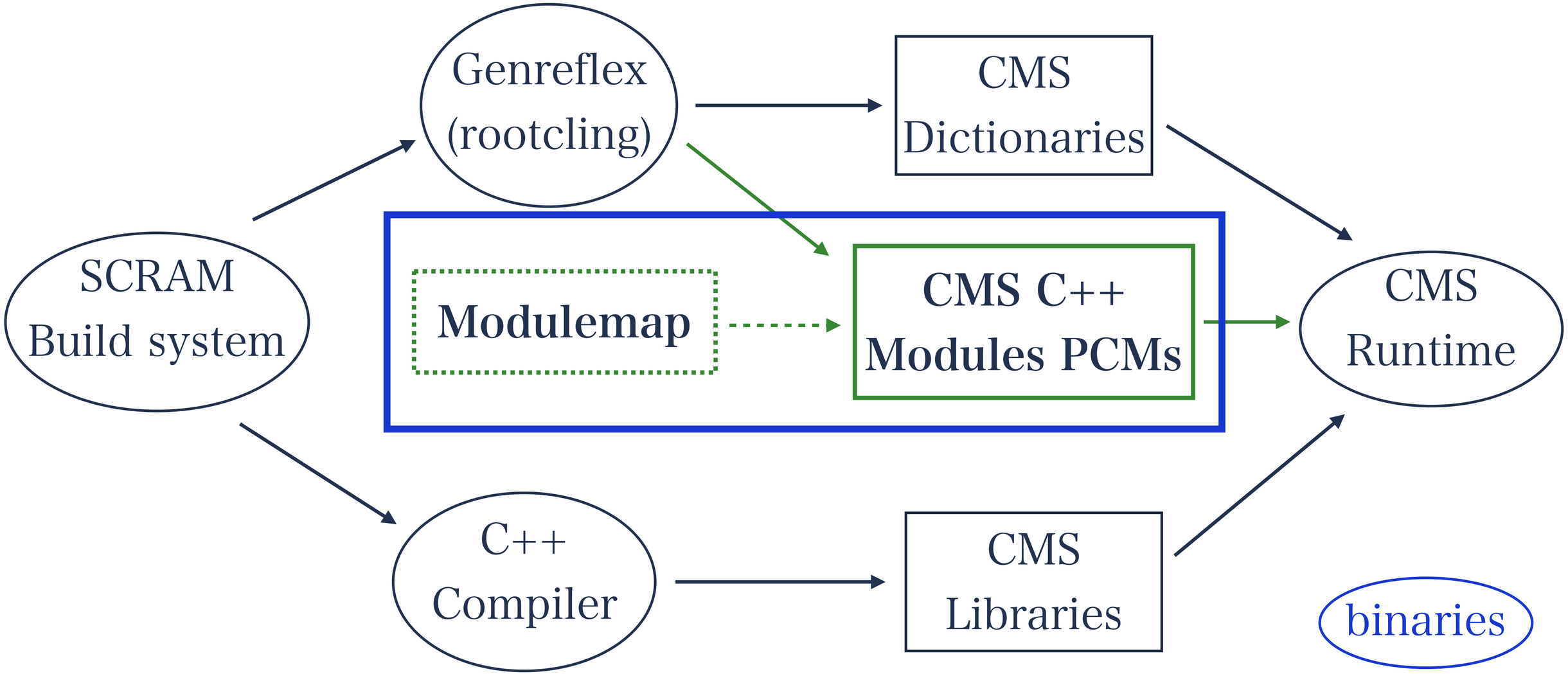
Overview - Dependency Graph



- Not all CMS libraries were modularized
- Modules can co-exist with the old infrastructure

C++ Modules in CMSSW

Overview - Dependency Graph



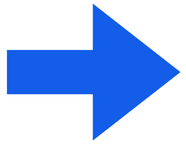
Explicit PCMs in CMSSW

module.modulemap

- Definition file of headers to build a PCM in Clang
- Contain all “interface” headers, which are used by libraries

```
module "MathCore" {
  module “TComplex name” { header "TComplex.h" export * }
  module <name of the file> {
    header <relative path to the header file location> }
}
```

modulemap will contain all interface header files



Autogeneration of modulemap

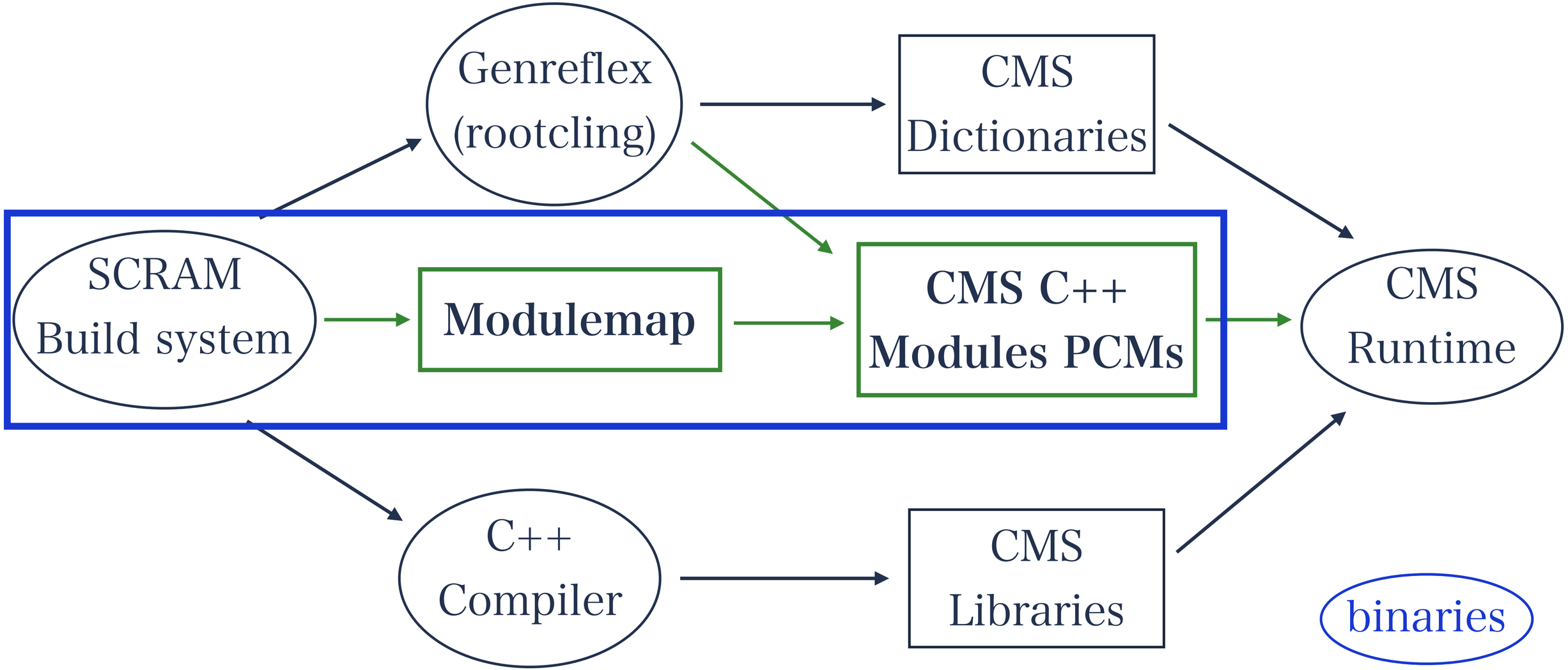
Explicit PCMs in CMSSW

Autogeneration of modulemap

- CMSSW has “**interface**” headers
  - Exposed to libraries outside
- **Automatically generate the modulemap** by adding interface headers
  - Modulemap needs to be generated before the execution of **genreflex**
  - **Build system** is responsible for the autogeneration

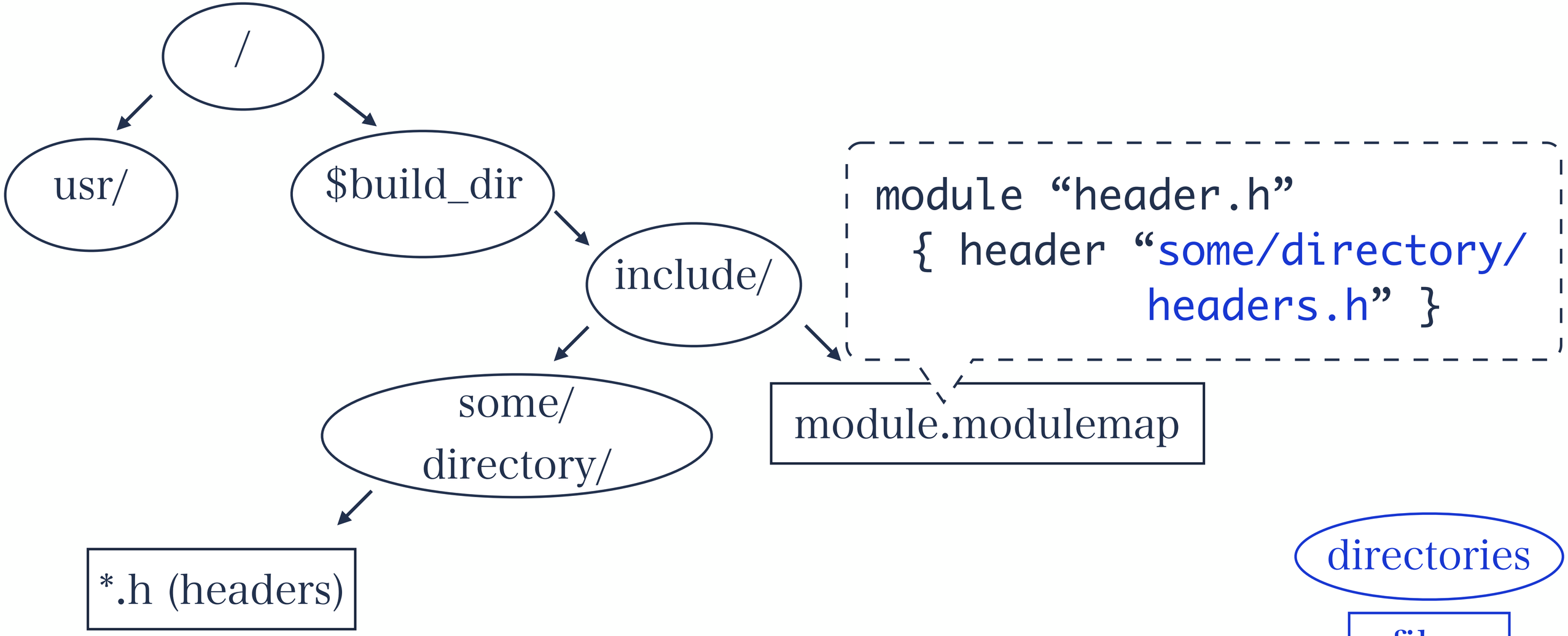
C++ Modules in CMSSW

Explicit PCMs in CMSSW



Mechanism of the modulemap

modulemap, modulemap overlay file, virtual modulemap overlay



Mechanism of the modulemap

modulemap, modulemap overlay file, virtual modulemap overlay

Modulemap for **system headers**

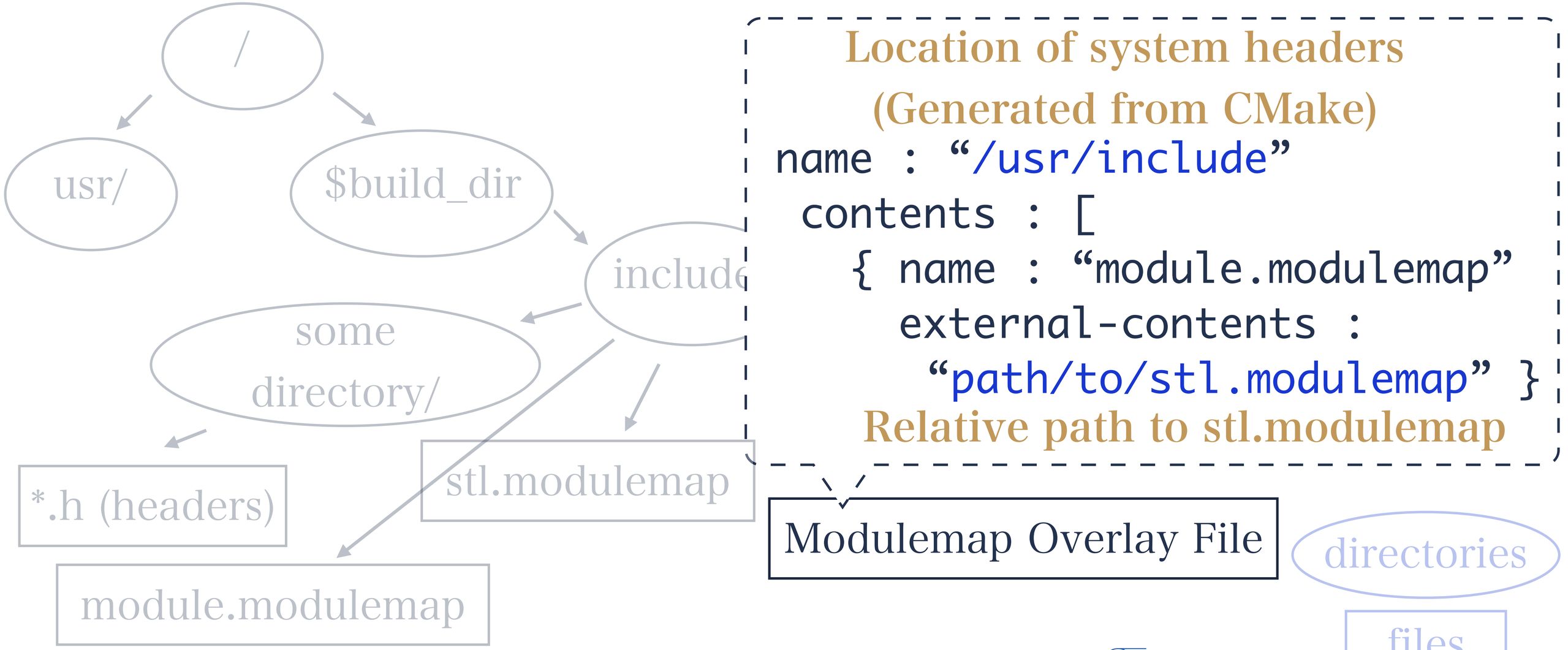
- Modulemap needs to know the **location** of headers
- System headers' location **cannot be hardcoded**

➡ **Modulemap Overlay File**

C++ Modules in CMSSW

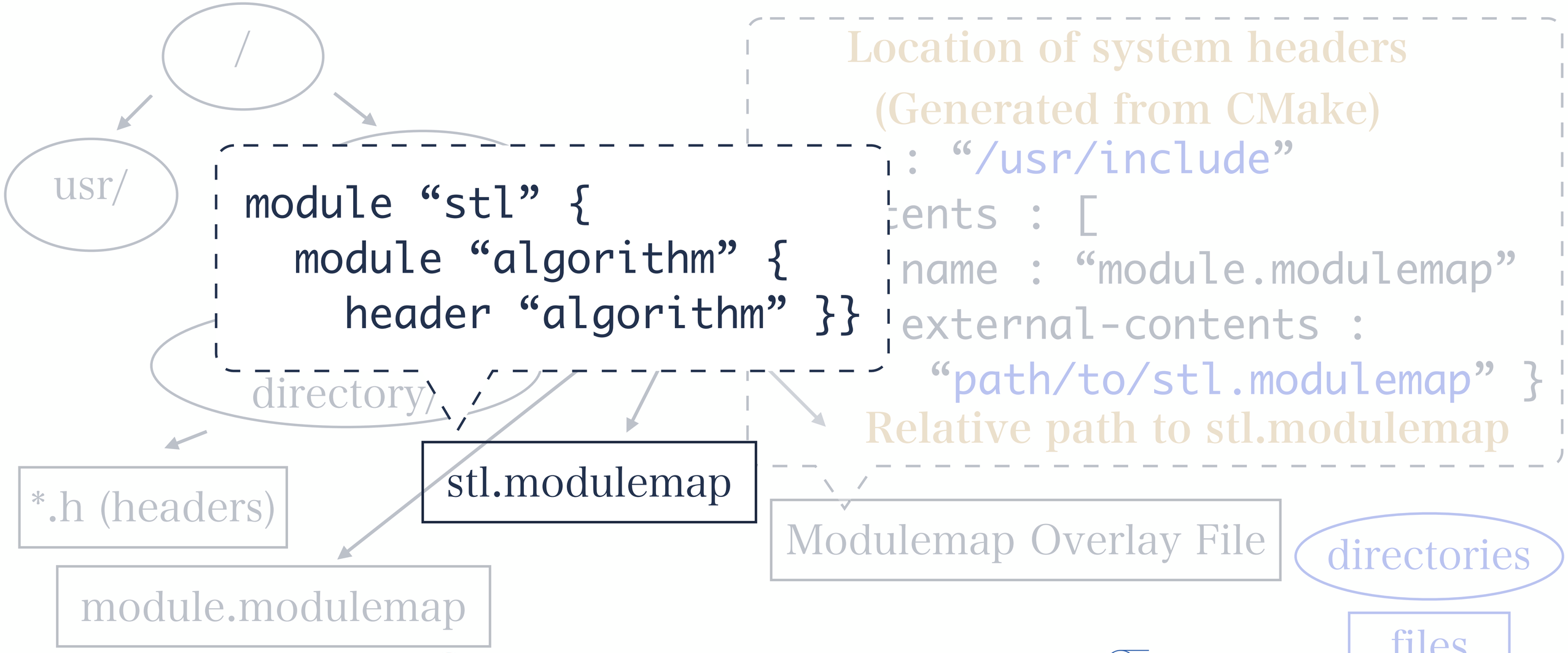
Mechanism of the modulemap

modulemap, modulemap overlay file, virtual modulemap overlay



Mechanism of the modulemap

modulemap, modulemap overlay file, virtual modulemap overlay



Mechanism of the modulemap

modulemap, modulemap overlay file, virtual modulemap overlay

Clang interprets those information as

- module.modulemap **exists in the location of system headers** (/usr/include, in this example)
- module.modulemap **has the contents of stl.modulemap**

→ **Modulemap Overlay File can handle system headers, but it needs to be generated at configuration time**

Mechanism of the modulemap

modulemap, modulemap overlay file, virtual modulemap overlay

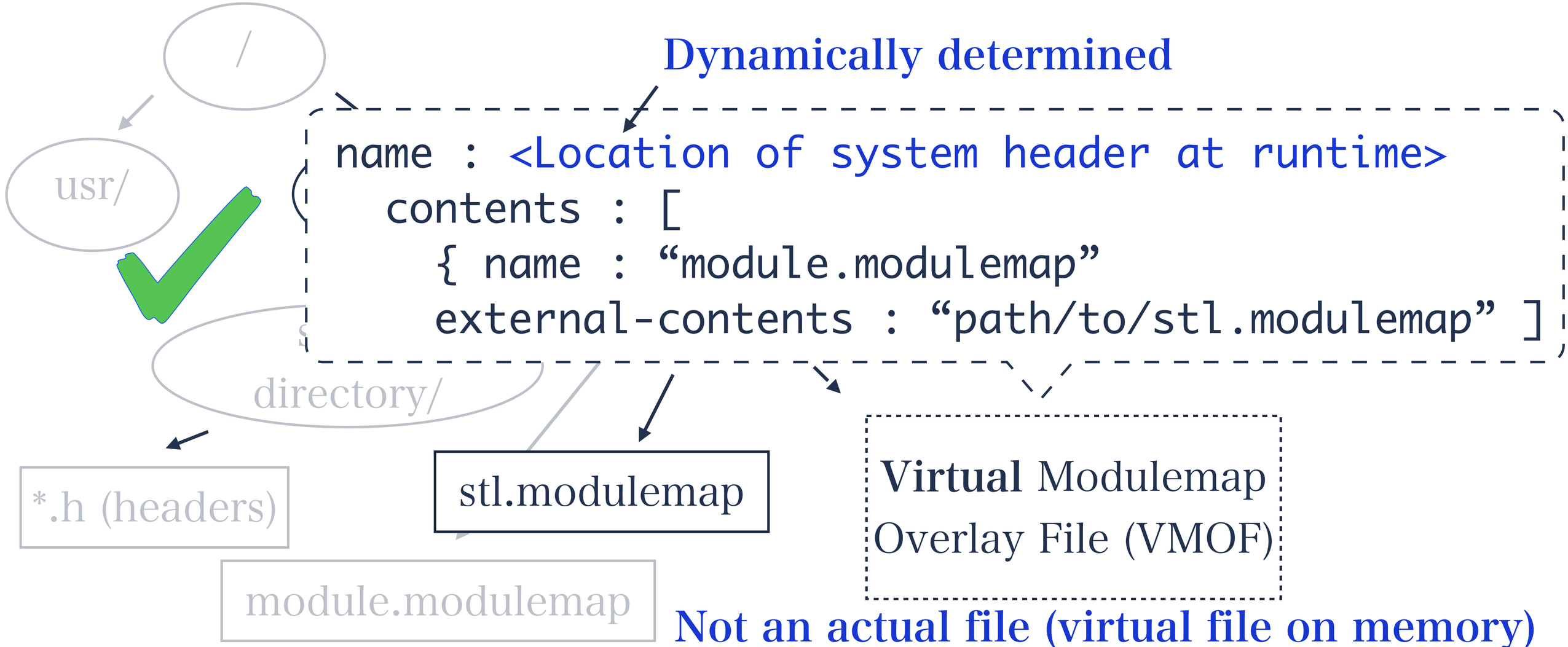
The location of system headers needs to be generated at **CMake (configuration) time**

- **Not binary distributable**
- **Not relocatable**
- CMS builds and distributes binary to other locations

→ **Solution: Virtual Modulemap Overlay File**

Mechanism of the modulemap

modulemap, modulemap overlay file, virtual modulemap overlay



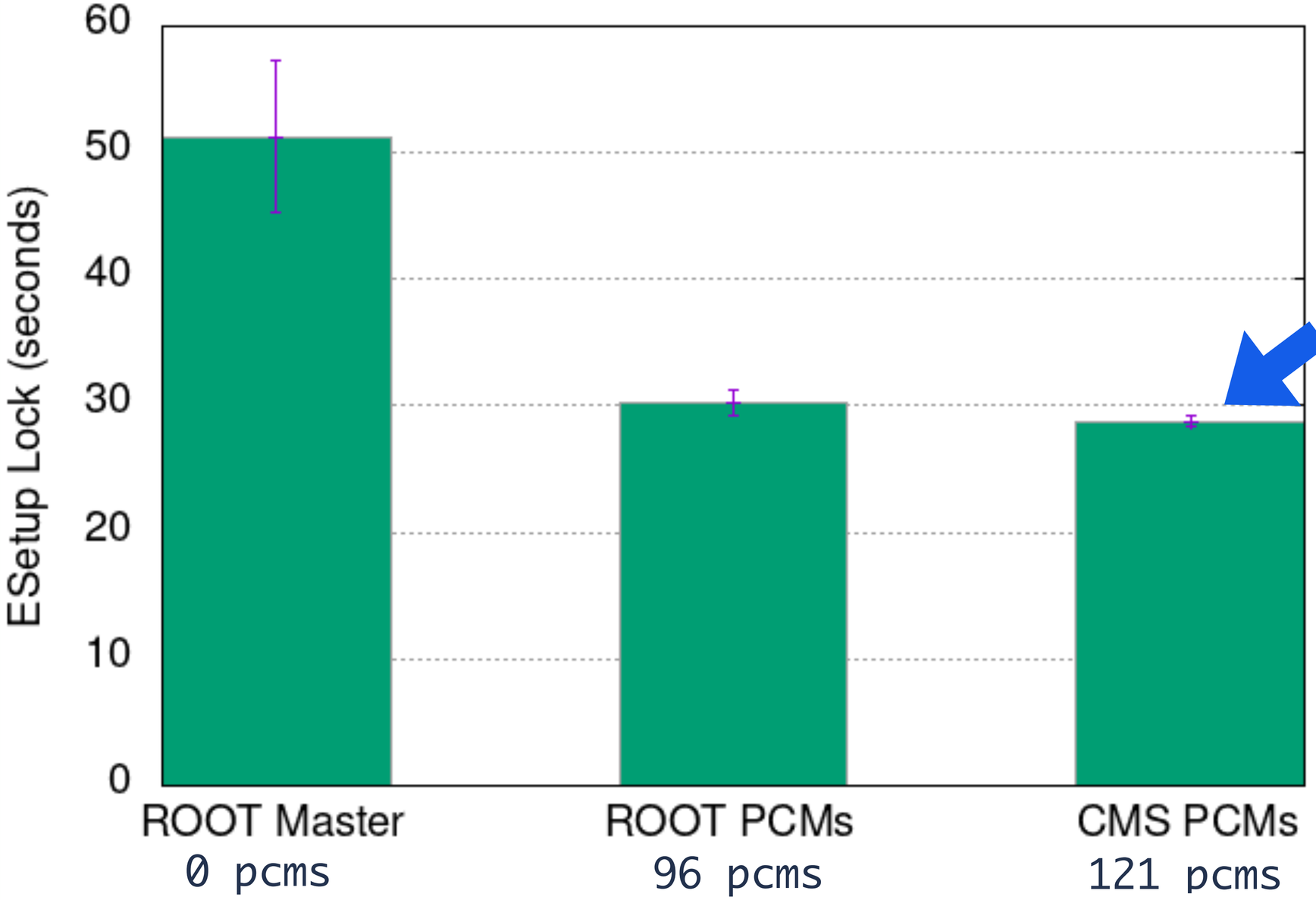
C++ Modules in CMSSW

Summary

- C++ Modules integration in CMSSW
  - **Genreflex** generates pcm files
  - **Autogeneration** of modulemap from build system
- Modulemap for system headers (libc, stl)
  - Virtual Modulemap overlay file

CMS Performance Results

“Fast simulation test” ESetup Lock (seconds)

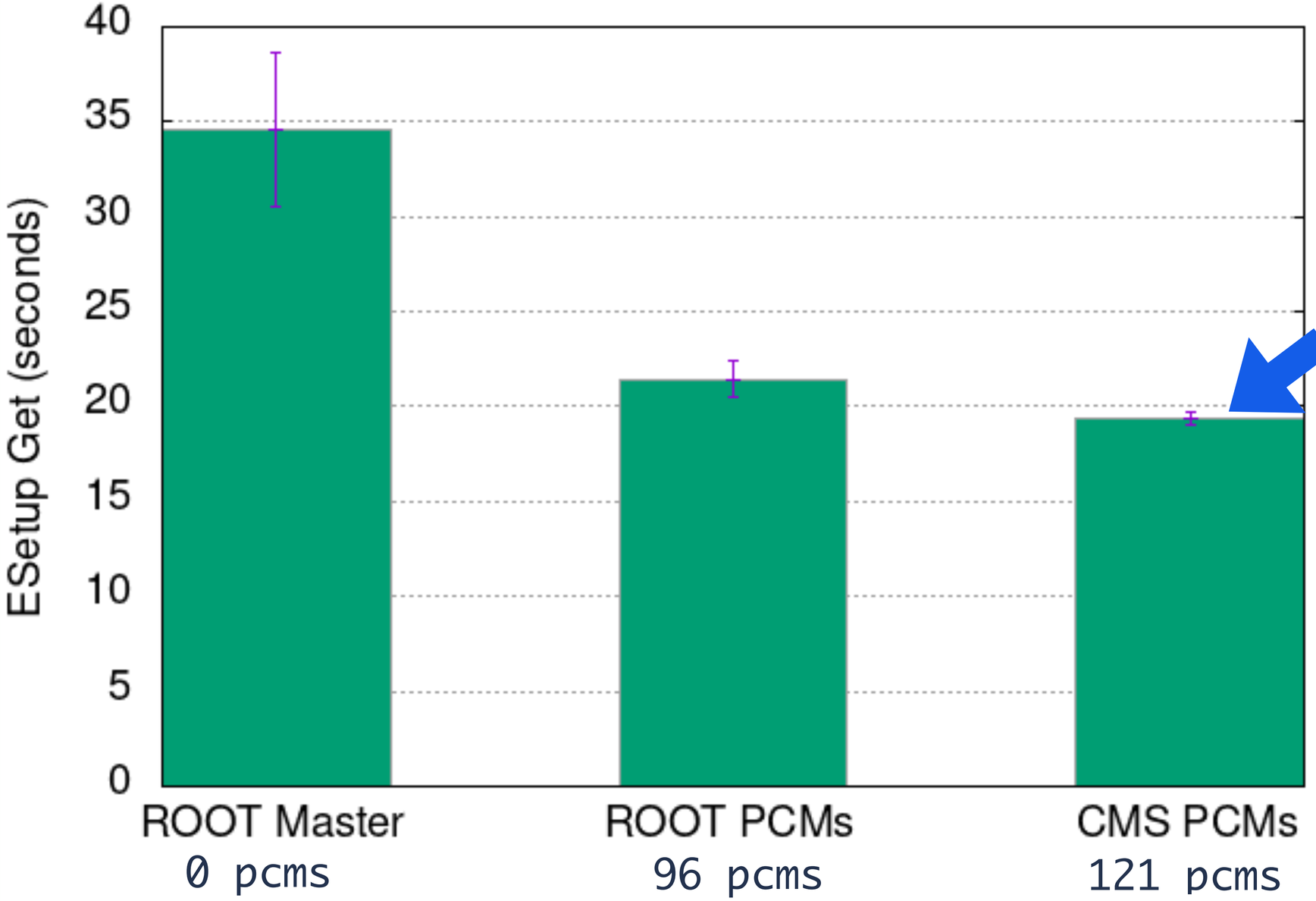


22.5 seconds better than ROOT Master

random 100 events
15 times execution

Standard error of mean

“Fast simulation test” ESetup Get (seconds)

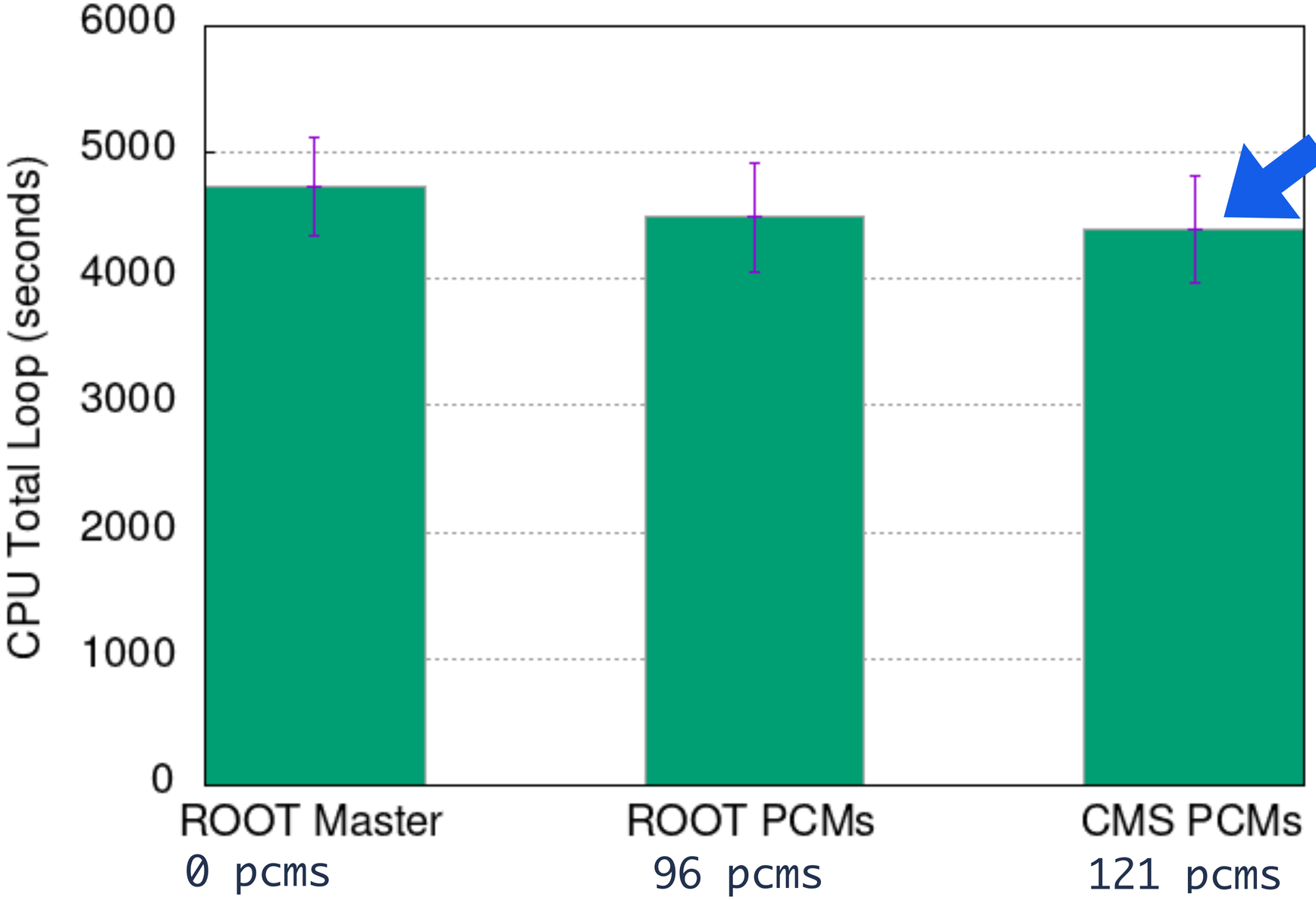


15.2 seconds better than ROOT Master

random 100 events
15 times execution

Standard error of mean

“Digitization test” CPU Total Loop (seconds)

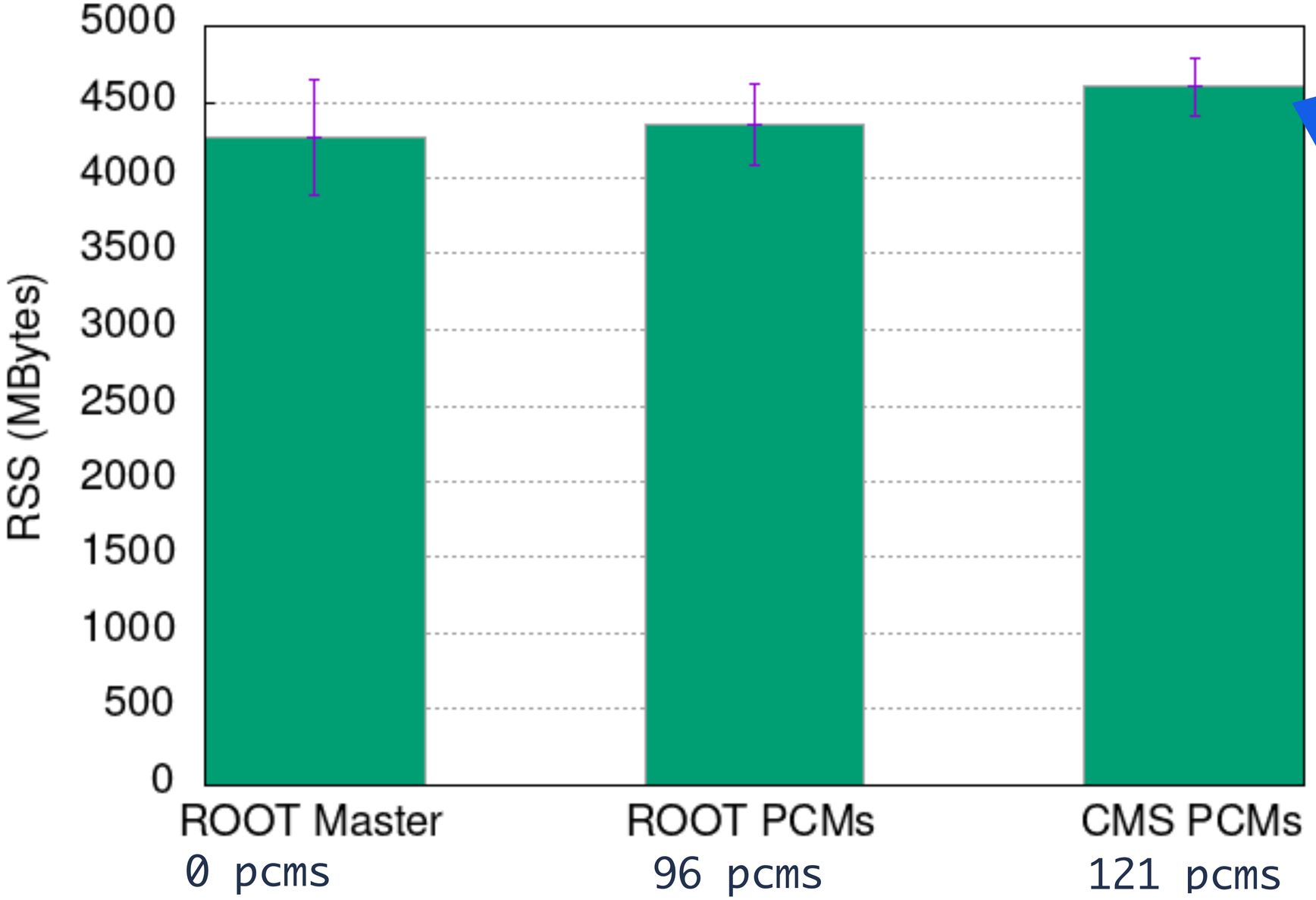


331 seconds better than ROOT Master

random 100 events
15 times execution

Standard error of mean

“Digitization test” RSS Memory (MBytes)



143 Mbytes worse than ROOT Master

random 100 events
15 times execution

Standard error of mean

CMS Performance Results

Summary

- Results suggests the performance benefits at **runtime**
  - Especially at the **initialization time**
- ~150 MBytes **RSS overhead**
  - Investigation is ongoing

Conclusion

Conclusion

- C++ Modules were implemented and tested in ROOT and in CMSSW
  - Improves the header modularity of libraries
- Preliminary performance study suggested the performance improvement at runtime
  - Work on performance improvement is ongoing

Thank you for your attention!

Backup slides

Implicit PCMs in CMSSW

Implicit pcms

Implicitly generated **without** modulemaps

- Add all possible header files needed for the generation of the dictionary
  - **Huge header duplication**

Explicit pcms

Explicitly generated **with** modulemaps

- Only add defined headers to the PCM
  - **Reduce header duplication**