Geant4 CMake Updates

Ben Morgan

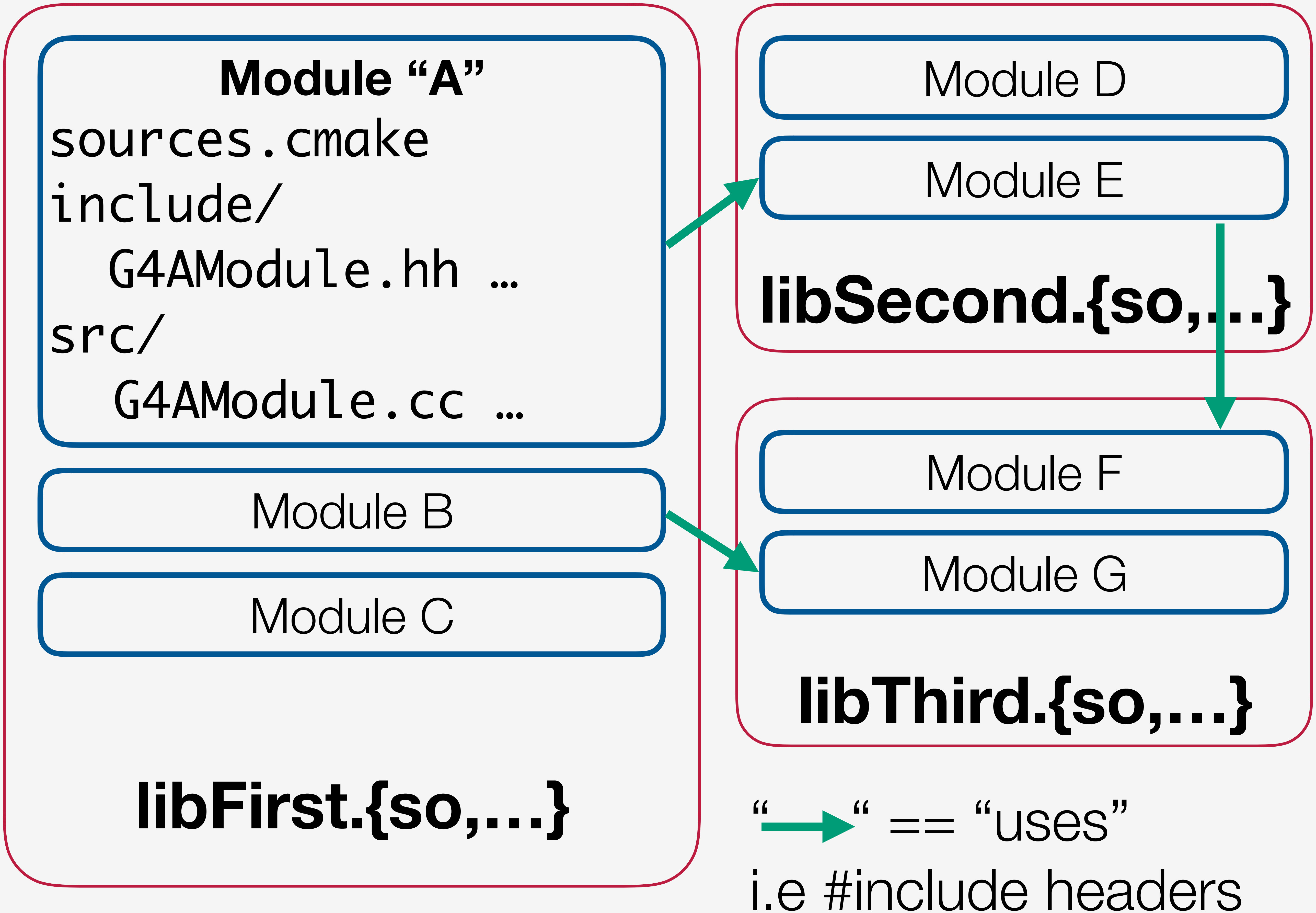
THE UNIVERSITY OF
WARWICK

Prototype Code on CERN Gitlab

```
$ git clone https://gitlab.cern.ch/bmorgan/geant4
$ cd geant4
$ git checkout -t origin/feature/modular-cmake
$ git diff --name-only master
```

- Branched from 10.3-beta
- Most changes in cmake category, aim is for backward compatibility as far as possible so no changes to category/module buildscripts needed.

Optimising Geant4 Library Structure: CMake Functionality and Migration



Modules To Libraries in Geant4

Library Structures in Geant4

- **“Granular” structure == 1 module -> 1 library**
  - ~144 libraries: Lack of coherence, complex deps
- **“Global” structure == N modules -> 1 library**
  - ~30 libraries: Large variance in size, lack of modularity
- ***Solution: Move to single structure optimised for coherence, modularity, performance.***
- ***Current CMake usage hard codes Global/Granular…***

Involved CMake Scripts

```
+- geant4
   +- cmake/
   +- source/
      +- CMakeLists.txt
      +- first_category
      |  +- CMakeLists.txt
      |  +- A_module
      |  |  +- sources.cmake
      |  |  +- CMakeLists.txt
      |  |  +- include/
      |  |  +- src/
      |  +- B_module/
      +- second_category/
```

source/CMakeLists.txt: Adds categories. Builds/Resolves .so/links

first_category/CMakeLists.txt: Adds Modules. Declares Global Library

sources.cmake: Declares Module

A_module/CMakeLists.txt: Granular Library

New CMake System: Back Compatibility

- New option `GEANT4_USE_NEW_CMAKE`: when set, use new implementation, otherwise existing system.

- Both systems provide **same** CMake function interfaces for module and library declaration:

  - `geant4_define_module (use in sources.cmake)`

  - `geant4_global_library_target (category CMakeLists)`

- **New system just a different implementation, so category/module scripts do not have to change and get current Global Structure as starting point**

```
$ git diff --name-only master
CMakeLists.txt
cmake/Modules/G4CMakeMain.cmake
cmake/Modules/G4DeveloperAPI.cmake
cmake/Modules/Geant4BuildProjectConfig.cmake
cmake/Modules/Geant4MacroDefineModule.cmake
cmake/Modules/Geant4MacroLibraryTargets.cmake
cmake/Modules/Geant4OptionalComponents.cmake
cmake/Modules/documentation/CMakeLists.txt
cmake/Modules/documentation/G4CMakeDocumentation.cmake
cmake/Modules/documentation/Modules/G4DeveloperAPI.rst
cmake/Modules/documentation/cmake.py
cmake/Modules/documentation/conf.py.in
cmake/Modules/documentation/index.rst
cmake/Modules/documentation/inventory.py
cmake/Templates/Geant4Config.cmake.in
source/CMakeLists.txt
source/analysis/parameters/sources.cmake
source/externals/zlib/sources.cmake
source/global/management/include/G4GlobalConfig.hh.in
source/global/management/sources.cmake
```

Yes, changes really are minimal!

geant4_define_module

```
# source/first_category/A_module/sources.cmake
# No longer use this information but can be left
include_directories(… path to E module headers …)
# … need to know dependencies of dependencies of.. …
include_directories(… path to F module headers …)
…
include_directories(${ZLIB_INCLUDE_DIRS})

geant4_define_module(NAME A
  HEADERS G4AModule.hh …
  SOURCES G4AModule.cc …
  LINK_LIBRARIES ${ZLIB_LIBRARIES}
  GRANULAR_DEPENDENCIES E
# … Need to know library structure(s) …
  GLOBAL_DEPENDENCIES Second
 )
```

Under the Hood…

```
# source/first_category/A/sources.cmake
geant4_define_module(…)
  # Implement “Module” like CMake Targets:
  # Build a module composed from these headers/sources…
  geant4_add_module(A PUBLIC_HEADERS … SOURCES …)

  # When building OR USING this target, add these as -I
  # NB: this is called inside geant4_add_module
  geant4_module_include_directories(A
    PUBLIC ${CMAKE_CURRENT_LIST_DIR}/include
    )

  # When building OR USING this target, link these libs
  geant4_module_link_libraries(A
    PUBLIC E ${ZLIB_LIBRARIES}
    )
```

geant4_global_library_target

```
# source/first_category/CMakeLists.txt
# No longer use this information but can be left
add_subdirectory(A)
add_subdirectory(B)
add_subdirectory(C)

if(NOT GEANT4_BUILD_GRANULAR_LIBS)
  geant4_global_library_target(NAME First
    COMPONENTS
      A/sources.cmake
      B/sources.cmake
      C/sources.cmake
  )
endif()
```

Under the Hood…

```
# source/first_category/CMakeLists.txt
geant4_global_library_target(…)
  # Load the module declarations
  include(A/sources.cmake)
  include(B/sources.cmake)
  include(C/sources.cmake)
  # Implement “Library” like CMake Targets
  # Declare a library to be composed from these modules
  # It’s this function that forms the Library Structure
  geant4_add_library(First MODULES A B C)
  # 1: Modules must exist and not be used elsewhere
  # 2: Marks “parent library” of A, B, C as “First”
  # NB: No actual CMake target created here…
```

Library Structure Hard Coding: Granular

```
~~# source/first_category/A/CMakeLists.txt~~

~~if(GEANT4_BUILD_GRANULAR_LIBS)~~
~~  geant4_granular_library_target(COMPONENT~~
~~    sources.cmake~~
~~  )~~
~~endif()~~
```

"Granular" library interfaces no longer exists directly

geant4_compose_targets

```
# source/CMakeLists.txt
# Add each category, declares modules, “libraries”
# Identical to what we do now
add_subdirectory(First)
add_subdirectory(Second)
add_subdirectory(Third)

# Now have a new element…
if(GEANT4_USE_NEW_CMAKE)
  geant4_compose_targets()
endif()
```

Under the Hood...

```
# source/CMakeLists.txt
geant4_compose_targets()
  foreach(library L declared)
    add_library(L “”)
    foreach(module M in L)
      target_sources(L sources of M)
      target_include_directories(L inc dirs of M)
      # Simplified - argument list arrived at in
      #             several steps!
      target_link_libraries(L
        <<Resolve Module links back to Parent Library>>
        <<Direct link anything that’s not a Module>>
        <<Remove Duplicates>>
        )
    endforeach()
  endforeach()
```

Stock CMake Library Commands

Changing The Library Structure

- Still have Global Structure hard-coded via category level calls to geant4_add_library
- In progress: Override by supplying a “structure file”:
- Simply contains a list of geant4_add_library commands

```
# Structure1.cmake
geant4_add_library(First MODULES A B C)
```

```
# Structure2.cmake
geant4_add_library(First   MODULES A B)
geant4_add_library(First_C MODULES C)
```

Remaining Tasks

- Optional modules/libraries (GDML is canonical case)
  - Suggest that only **libraries** be optional, as then existence of library is guarantee of functionality
  - Suggest that optional libraries not be depended on by anything in the toolkit, otherwise need additional dependency resolution step(s).
- Full export of targets etc to Geant4Config.cmake and geant4-config, validate that tests and examples build!

New CMake Implementation Rollout Proposal

- Review and finalise CMake functionality (now).
- Import new CMake functionality into source tree (**Jan 2017?**).
- Fix any issues like old/missing granular dependencies
- Provide single testing platform/instance that builds/tests with `GEANT4_USE_NEW_CMAKE` to ON
- Ensure tests pass, libraries and performance are identical!

Library Restructure Tasks

- **Only after reproduction of identical Global libraries!**
- Initial work will be to optimise grouping of current modules into libraries *inside categories*, but we should think about restructuring beyond this
- Merge some Categories into same library?
- Move some Modules into a different Category?
- Move some Module code to other Modules (e.g. base vs concrete classes)?

Documentation of Geant4 CMake Options, Functions and Usage

Why Document Geant4's CMake System?

- Developers need to have a working knowledge of the system to add new code, define dependencies on other parts of Geant4, and integrate tests.
- Developers/Users need to know about the various options available to configure Geant4 (optimization level, MT, optional components)
  - In Installation Guide, but options often added during development
- Developers/Users need to know how to locate and use a build/install of Geant4 using CMake or other buildtool
  - Not CMake specific, but tools/support scripts are generated by it, so responsibility to document falls on CMake category…

Documenting the CMake Modules/Functions

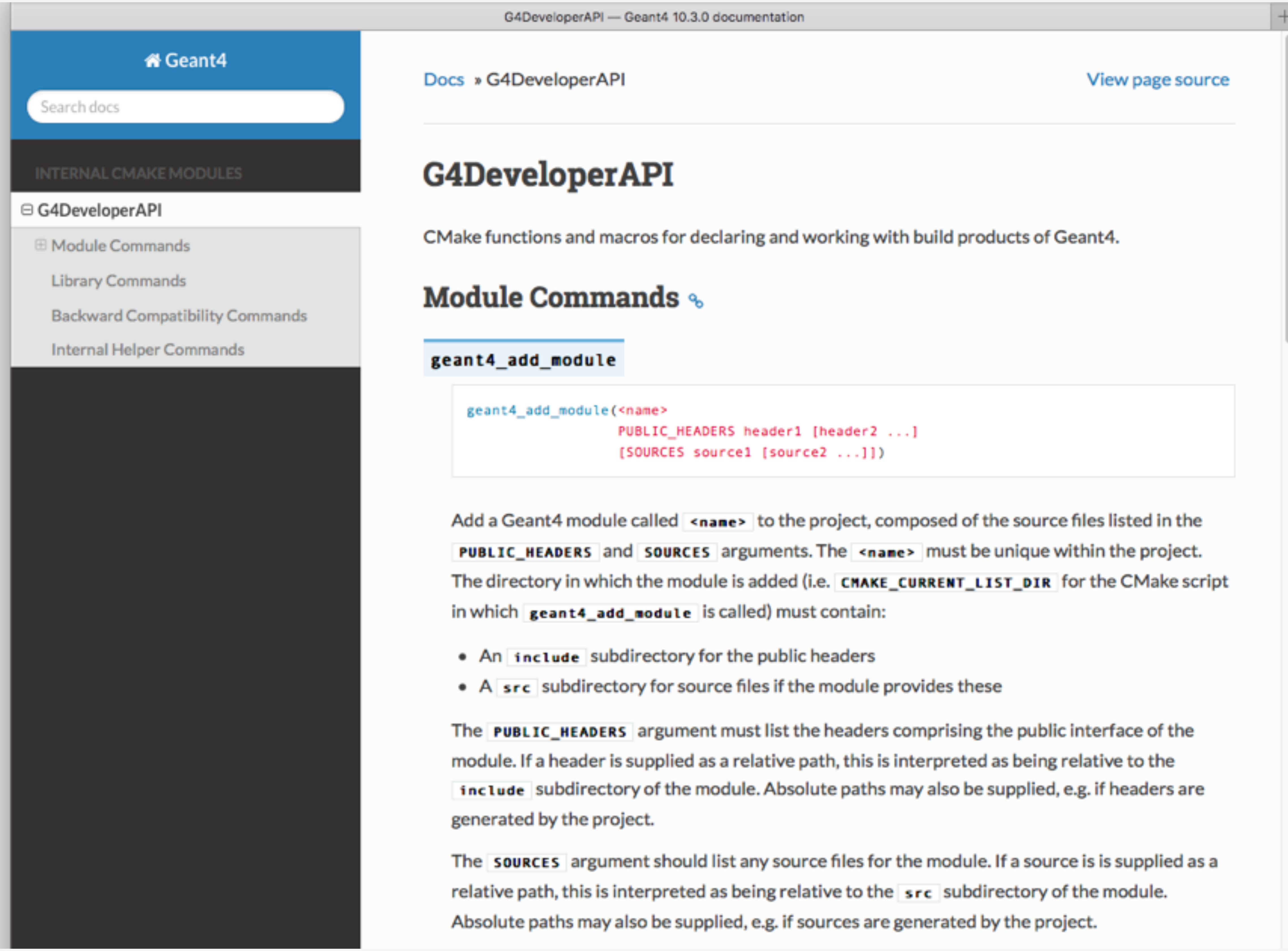
- CMake itself is documented using reStructuredText to markup cmake script (ala Doxygen for C++) and Sphinx to generate HTML/PDF/etc
- Copy CMake's `cmake.py` Sphinx parser across, markup Geant4 CMake modules and build documentation pages for them ("make doc")
- **Discussed in Parallel 2, further Thoughts/Comments/Suggestions?**
  - Advantage: everything in one place, always up to date, can cross-ref CMake's documentation
  - Disadvantage: Have some duplication between Install and Application Guides (but *could* cross-ref)

```cmake

if(NOT __G4DEVELOPERAPI_INCLUDED)
  set(__G4DEVELOPERAPI_INCLUDED TRUE)
else()
  return()
endif()

#-----------------------------------------------------------------------
#.rst:
# Module Commands
# ^^^^^^^^^^^^^^^
#
# .. cmake:command:: geant4_add_module
#
#   .. code-block:: cmake
#
#     geant4_add_module(<name>
#                       PUBLIC_HEADERS header1 [header2 ...]
#                       [SOURCES source1 [source2 ...]])
#
#   Add a Geant4 module called ``<name>`` to the project, composed
#   of the source files listed in the ``PUBLIC_HEADERS`` and ``SOURCES``
#   arguments. The ``<name>`` must be unique within the project.
#   The directory in which the module is added (i.e. ``CMAKE_CURRENT_LIST_DIR``
#   for the CMake script in which ``geant4_add_module`` is called) must contain:
#
#   * An ``include`` subdirectory for the public headers
#   * A ``src`` subdirectory for source files if the module provides these
#
#   The ``PUBLIC_HEADERS`` argument must list the headers comprising the
#   public interface of the module. If a header is supplied as a relative path,
#   this is interpreted as being relative to the ``include`` subdirectory of the module.
```

Example CMake Script with reStructuredText Markup



G4DeveloperAPI — Geant4 10.3.0 documentation

Geant4

Search docs

INTERNAL CMAKE MODULES

⊟ G4DeveloperAPI

⊞ Module Commands

Library Commands

Backward Compatibility Commands

Internal Helper Commands

Docs » G4DeveloperAPI

View page source

G4DeveloperAPI

CMake functions and macros for declaring and working with build products of Geant4.

Module Commands

geant4_add_module

```
geant4_add_module(<name>
                  PUBLIC_HEADERS header1 [header2 ...]
                  [SOURCES source1 [source2 ...]])
```

Add a Geant4 module called `<name>` to the project, composed of the source files listed in the `PUBLIC_HEADERS` and `SOURCES` arguments. The `<name>` must be unique within the project. The directory in which the module is added (i.e. `CMAKE_CURRENT_LIST_DIR` for the CMake script in which `geant4_add_module` is called) must contain:

- An `include` subdirectory for the public headers
- A `src` subdirectory for source files if the module provides these

The `PUBLIC_HEADERS` argument must list the headers comprising the public interface of the module. If a header is supplied as a relative path, this is interpreted as being relative to the `include` subdirectory of the module. Absolute paths may also be supplied, e.g. if headers are generated by the project.

The `SOURCES` argument should list any source files for the module. If a source is is supplied as a relative path, this is interpreted as being relative to the `src` subdirectory of the module. Absolute paths may also be supplied, e.g. if sources are generated by the project.

Example HTML Page as Generated by Sphinx using ReadTheDocs style
(for separate but more complete example, see http://drbenmorgan.github.io/cetbuildtools2

Usage of Preprocessor Directives/Flags in Geant4 and Consistent Builds

Simplifying Usage of -D/Preprocessor Flags

- Many places in Geant4 use preprocessor directives to control functionality, with compiler flags used to activate via definitions
- Standard practice, but have several cases where these -D flags must be consistently applied to both build of Geant4 **and** client code**:**

```
$ c++ -DG4MULTITHREADED … G4Source.cc

$ c++ -DG4MULTITHREADED … UserApplication.cc
```

- This has the potential for confusion and/or inconsistent builds
- **CMake/GMake/geant4-config can transmit flags, but an easier and more robust solution without these tools or user knowledge is possible**

Replace -D flags by #define Directives

- Essentially a hard-coding problem: Macros appear in public headers **and** source files of Geant4.

- But "hard coding" is really requirement to use a `-D` flag to define the macro

```
$ c++ -DG4MULTITHREADED … UserApplication.cc
```

- **Solution: make hard-coding explicit by replacing definition via compiler flags with C++ header(s) containing #define directives.**

- *In Geant4 code, simply* `#include` *this header where required*

- *In Client Code, simply* `#include` *this header where required (or not if the code doesn't directly use the macro).*

Creating “G4GlobalConfig.hh”

- **Use CMake’s** configure_file **and** #cmakedefine **functionality (or** sed/awk **in GMake)**

```
/// “G4GlobalConfig.hh.in”

#ifndef G4GLOBALCONFIG_HH_HH
#define G4GLOBALCONFIG_HH_HH

/// Defined if Geant4 built in MT mode
#cmakedefine G4MULTITHREADED

/// Numeric version of C++ Standard compiled against
#cmakedefine GEANT4_BUILD_CXXSTD "@GEANT4_BUILD_CXXSTD@"

/// Here's something that isn't defined
#cmakedefine GEANT4_DEMO_UNDEF

#endif // G4GLOBALCONFIG_HH_HH
```

Creating “G4GlobalConfig.hh”

- Depending on the settings chosen, the `#define` directives are set accordingly in the generated header:

```
/// “G4GlobalConfig.hh”

#ifndef G4GLOBALCONFIG_HH_HH
#define G4GLOBALCONFIG_HH_HH

/// Defined if Geant4 built in MT mode
#define G4MULTITHREADED

/// Numeric version of C++ Standard compiled against
#define GEANT4_BUILD_CXXSTD "11"

/// Here's something that isn't defined
/* #undef GEANT4_DEMO_UNDEF */

#endif // G4GLOBALCONFIG_HH_HH
```

Using "G4GlobalConfig.hh"

- Geant4 and client code simplify `#include` the header, and the macro definitions are **available and consistent without requiring a compiler flag.**

```
/// "G4Code.hh/cc" or "UserCode.hh/cc"

#include "G4GlobalConfig.hh"

…

#if defined(G4MULTITHREADED)
…
#else
…
#endif
```

Implement After 10.3?

- Need to identify all macros, where they are used, and their build/use time requirements.
- Implementation of config headers in CMake build is easy, need to review for GMake.
- Review whether to explicitly `#undef` macros before `#cmakedefine` to **guarantee** that `-D` flags cannot override.
- Gradual rollout through code based on review - testing should pick up issues via compile or link time errors.

Discussion