DEVELOPING A MODERN BUILD SYSTEM FOR THE FRAMEWORK MESSy

deRSE25, 27th February 2025

Sven Goldberg, Melven Röhrig-Zöllner

1. What is MESSy?

DLR

MESSy is …

a software framework that combines components, which are numerical representations of our Earth system. Examples of components are atmosphere, land and ocean models, and more.

https://messy-interface.org

- … an abbreviation: **M**odular **E**arth **S**ubmodel **Sy**stem
- … built from ~ 3,500,000 SLOC in Fortran, ~ 500,000 SLOC in C/C++
  - More than 50 executables
- … continuously developed (> 20 years), widely used (2024: 41 publications)
  - Supported by DKRZ, LRZ, MPCDF, JSC, terrabyte cooperation (DLR, LRZ)

2. MESSy‘s previous build system

- Build systems in general: Automate build process
  - Compile code
  - Configure and build executables/libraries

- MESSy: `autoconf` build
  - Requires configure files and makefiles

→ Two types of verbose files need to be maintained

3. Why do we want to use CMake?
Results for MESSy and general advantages

Metric		CMake build	autoconf build
LOC (total)		~ 10,000	~ 100,000
LOC (root files)		650	~ 11,000
Build time (averaged)		180 s	290 s
CMake/configure		30 s	13 s
make		150 s	277 s
Recompile time File with many dependecies		3.5 s	15 s

- Further advantages
  - Out-of-source build → Different configurations in parallel
  - Easy to include new architectures (HPC clusters)
  - Easy integration into other software (→ More users)
  - Interface can be used to directly manipulate options/build variables

4. CMake for MESSy – First steps

- Initial situation:
  - Mostly Fortran90 code (partially C/C++), GNU `autoconf` build system exists
- Start with absolute easiest possible configuration
- How did we know what to build?
- Get overview of mandatory files and libraries for (main) executable
  - Use (root) Makefile (→ `all` target)

4. CMake for MESSy – Understanding autoconf

- MESSy's `all target`:

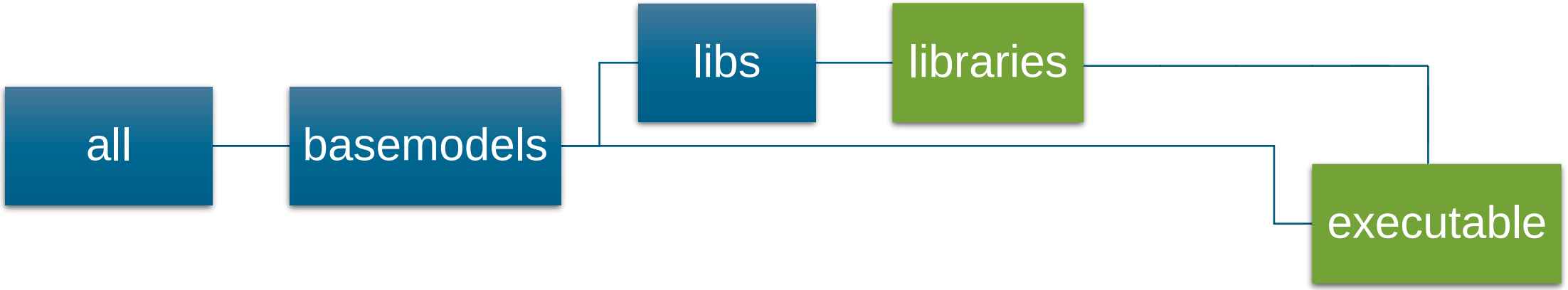
```
.PHONY: all
all: modlog basemodels
```

target: `all` — dependencies: `modlog basemodels` — rule

```
.PHONY: basemodels
basemodels: libs
	@if [ "$$target" != "" ] ; then \
	    ZBASEDIRS="$(subst $(comma),$(space),$(target))" ;\
	    echo "basemodels: $$ZBASEDIRS" ;\
```

recipe

- In the recipe, the `Makefile` of the actual executable (echam5) is called



4. CMake for MESSy – Minimal configuration

Root CMakeLists.txt

```
cmake_minimum_required(VERSION 3.20)
project(MESSy LANGUAGES Fortran C)
set(CMAKE_Fortran_PREPROCESS On)

list(APPEND CMAKE_MODULE_PATH "${CMAKE_CURRENT_SOURCE_DIR}/cmake/")

set(CMAKE_Fortran_FLAGS "${CMAKE_Fortran_FLAGS} -ffree-line-length-none -fallow-invalid-boz")

add_compile_definitions(MPIOM_13B MESSY)

find_package(MPI REQUIRED COMPONENTS Fortran C)
find_package(NetCDF REQUIRED COMPONENTS Fortran C)
find_package(LAPACK REQUIRED)

# Create config.h file needed for echam5/support/util_sysinfo.c
configure_file("${CMAKE_CURRENT_SOURCE_DIR}/config/config.h.in"
        "${CMAKE_CURRENT_BINARY_DIR}/config.h")
include_directories(${CMAKE_CURRENT_BINARY_DIR})

add_subdirectory(libsrc)
add_subdirectory(messy)
add_subdirectory(mpiom)
add_subdirectory(echam5)
```

4. CMake for MESSy – Developing the build system

Add configuration options – From autoconf to CMake

- Solid basis → Add compile definitions and configuration options
- Configure file lists options and defines their behaviour (very verbose)

autoconf

- Example: `--enable-ASYNCF`

```
if test "$enable_ASYNCF" = "yes"; then :
            ASYNCF_DEF="HAVE_ASYNCF"
            LIBSRCS+=" libsrc/async-fortran/build"
            MESSY_LIB+=" -lasyncf"
```

- Define CMake `option` ASYNCF (by default `OFF`)
- Add path and define variables to link against library

CMake

```
option(ASYNCF "Build MESSy with asynchronous fortran library (default: OFF)." OFF)
if (ASYNCF)
  add_subdirectory(libsrc/async-fortran)
  set(ASYNCF_COMPILE_DEF "HAVE_ASYNCF")
  list(APPEND MESSYLIBS_OPTIONAL_Fortran async_threads_fortran)
endif()
```

5. General aspects and learnings I

- CI job in repo to save state and (pFUnit) test framework
- Use `README-cmake.md`, Modulefiles and flagfiles (→ clusters)
- Do not run autoconf and CMake build in same local directory
- CMake syntax (string behaviour, structure of function inputs, …)
- Often, there is a way to translate autoconf into CMake command
  - If not: `add_custom_target()`, `add_custom_command()`, `execute_process()`
  - The detailed CMake documentation is your friend
- ‘Trial and error‘ is legitimate approach
- MESSy: Work of almost 6 months of FTE

5. General aspects and learnings II
MESSy-specific

- Compiling error for MESSy‘s optional `guess` library
- Some libraries bring their own `config.h` file → Naming conflicts
  - Usual way to add a (relative) directory for include files to target

```
target_include_directories(guess PUBLIC framework)
```

  - BUT: CMake appends all paths → Path to MESSy `config.h` file found first
  - Results in compilation error since the file does not contain right information
  - Fix this, by prepending the path using CMake keyword `BEFORE`

```
target_include_directories(guess BEFORE PUBLIC framework)
```

5. General aspects and learnings III
MESSy-specific

- `find_package()` very powerful to find external packages/libraries
- Might still find wrong/unwanted packages sometimes
- Example: Loaded module on cluster includes its own NetCDF
- Function will find this (unwanted) NetCDF
  → Make use of CMake variable `CMAKE_IGNORE_PATH`
- Append variable by path to module before `find_package()`
  - Hint: Store the old value of the variable and reset it after that

6. Conclusion

- Quotes by one of the main developers:
  - Maintenance simplified
  - Builds are more flexible, yet faster
  - Developers and users get along very well with new build system
  - It was worth the effort

Metric		CMake build	autoconf build
LOC (total)		~ 10,000	~ 100,000
LOC (root files)		650	~ 11,000
Build time (averaged)		180 s	290 s
CMake/configure		30 s	13 s
make		150 s	277 s
Recompile time File with many dependecies		3.5 s	15 s

Thank you for your attention!

Questions?