Electronic Communications of the EASST
Volume 85 Year 2025

deRSE25 - Selected Contributions of the 5th Conference for Research Software Engineering in Germany

Edited by: René Caspart, Florian Goth, Oliver Karras, Jan Linxweiler, Florian Thiery, Joachim Wuttke

Developing a Modern Build System for the Earth System Modelling Framework MESSy

Sven Goldberg, Melven Röhrig-Zöllner

DOI: 10.14279/eceasst.v85.2695

Electronic Communications of the EASST (https://eceasst.org).
Published by **Berlin Universities Publishing**
(https://www.berlin-universities-publishing.de/)

Developing a Modern Build System for the Earth System Modelling Framework MESSy

Sven Goldberg[a] **and Melven Röhrig-Zöllner**[b]

[a]sven.goldberg@dlr.de,
[b]melven.roehrig-zoellner@dlr.de
Institute of Software Technology, German Aerospace Center (DLR), Cologne (Germany)

Abstract: The earth system modelling framework MESSy [JSK+05] is a large Fortran-based software used on high-performance computing (HPC) clusters. On these systems, software is usually built from source with dedicated configuration for each cluster. This paper describes the process of replacing the old build system based on Autoconf [MED23] by a modern build system based on CMake [CMA]. CMake offers a higher abstraction level and better portability across HPC systems and architectures. We focus on recreating the existing configuration options and build targets (binaries, libraries) with identical compiler flags and dependencies while improving the maintainability, the usability, and the compilation time.

Keywords: Build System, CMake, Research Software, Legacy Software, Software Engineering, Compiled Languages

1 Introduction

The **M**odular **E**arth **S**ubmodel **Sy**stem (MESSy) [JSK+05] is a large software for earth system modelling with a focus on atmospheric chemistry and its role for air quality and climate. The development started in 2001 with an increasing number of yearly publications. In 2024, there were 41 publications [MESb] related to MESSy. MESSy has a modular structure and contains both older legacy parts and newly developed parts. The MESSy repository also includes code from third-party tools and libraries, often with patches or adjustments of the build instructions required to make those work on HPC clusters and to obtain binary identical results. It mainly consists of 3,500,000 lines of Fortran and 500,000 lines of C/C++ code, not counting comments or empty lines, from which more than 50 executables are built. Currently, the software is only available under closed-source license, but it is planned to be published as open-source after the licensing is fully clarified.

MESSy is supported and jointly developed by several research and computing centers, see [MESa]. The developers of MESSy come from different scientific domains such as atmospheric chemistry, geophysics, and climate modeling. In the last few years, about 50 scientists have contributed to the repository.

We, the authors of this paper, perform research on software and numerical methods but also act as part-time research software engineers (RSEs) and we did not previously know MESSy. This work was initiated through a survey aimed to improve the run-time performance and integration of software into the HPC systems of the German Aerospace Center (DLR). The build system

was on the list of open points for improvement of the MESSy developers, but they did not have appropriate funding for this. The complete re-implementation of the build system in CMake took about 1–1.5 years of part-time work (50%). Most of the work was performed by one person without deep prior knowledge about Autoconf and CMake with assistance from a person with 10+ years of experience in scientific computing.

This paper is structured as follows: In Section 2, we give a short introduction to build systems in general, and Autoconf and CMake in particular. After that, we describe the process of our implementation step by step in Section 3 starting with a minimal working configuration and then extending it to the feature set of the previous Autoconf build. In Section 4, we present obstacles and how we resolved them. In Section 5 we compare both build systems in MESSy and confirm the advantages of the new build. Finally, in Section 6, we summarize our main findings and suggest future improvements.

For the remainder of this paper, "old build (system)" will be used as a synonym for the Autoconf build and "new build (system)" will be used as a synonym for the CMake build.

As "users", we denote all people that make use of the build system. This includes developers and software maintainers as well as people that build MESSy to run a simulation on an HPC cluster (as this is the usual deployment method).

2 Build Systems

Build systems are an important part of a software with a strong influence on the development workflow. A short, high-level overview and some useful references on build systems are given in [MM18]. The main task of the build system is to set up rules on how to compile the code. Depending on the chosen configuration and environment, the build system defines all compiler commands with possibly tens to hundreds of flags and the ordering of the commands. For this, it needs to know the desired configuration, the structure of the code and their external dependencies. The internal high-level structure of the code is usually defined via libraries and their dependencies. Modern build systems automatically detect the low-level structure (file dependencies)—meaning that compiling one file outputs a file that is needed for compiling another file, or that changing one file requires recompiling another. External dependencies are the compilers and other software libraries but can also include further tools used for building, running or testing the software. In general, the build system needs to find required external dependencies and to adjust the compiler commands appropriately, as the location and specific configuration of libraries or tools vary. Additionally, modern build systems handle many details automatically, e.g., when facing code in different programming languages. Thus, developing and using scientific software written in a compiled language is inconceivable without a good build system.

2.1 Autoconf

The previous build system of MESSy is based on Autoconf [MED23] from the GNU Autotools suite. For a more detailed introduction and a modern minimal example, see [MTD+25, An introduction to the Autotools]. At first, the build is configured by executing the `configure` file. Here, configuration options are set. After that, the actual build is carried out using the

root `Makefile` and the ones in deeper layers by calling each other. Here, the compile commands are defined and executed and the source code's structure has to be represented. The `configure` file is generated by Autoconf from a template file (`configure.in` in old versions, `configure.ac` in newer versions). The `Makefile` files are then generated during the configure process from template files (`Makefile.in`). With Automake [MTD+25], one can also generate the `Makefile.in` files from another level of template files (`Makefile.am`). It is also possible to integrate targets and build commands from non-generated `Makefile` files. As a consequence, different types of files potentially have to be edited during the maintenance and extension of the build system: The sources of the `configure` file on the one hand and the template for the respective `Makefile` on the other hand.

In MESSy, most (but not all) of the `Makefile` files are generated. Automake is not used during the build process (some included libraries contain unused `Makefile.am` files). MESSy requires at least Autoconf 2.63 from 2008.

To summarize, the old build system uses a mixture of (old) Autoconf and plain (non-generated) Makefiles. We remark that rewriting the old build system using Automake and newer Autoconf features most-probably could also significantly improve the maintainability.

2.2 CMake

Compared to that, CMake consists of `CMakeLists.txt` files. These are written in the CMake language, see [CMA, CMake Tutorial] for a minimal example. A repository usually has one main `CMakeLists.txt` which can directly include other CMake files. One often structures CMake by defining one file in each directory containing source code. Then, one includes the build targets from the CMake file in the parent directory using the command `add_subdirectory(...)`. Running CMake generates the needed `Makefile` type of files from scratch. CMake also supports other build backends such as Ninja [NIN24], see [CMA, cmake-generators(7)]. Changing the build generator (the backend tool that actually calls the compiler) only requires changing one flag during the configuration. The generated files for the backend tool contain required compile instructions and dependencies. CMake and Autoconf are thus both meta build systems. A detailed general discussion about which meta build system is best is out of the scope of this paper. Some experiences from migrating from Autoconf to CMake are discussed in [SNH+12, MNA+14].

2.3 Motivation for Using CMake

Maintaining the build system incurs overhead for the software development as already indicated by the small empirical studies in [KE02, MAN+11]. A more exhaustive study about build systems in open source projects is shown in [MNA+14]. Further, those studies underline that improvements in the build system help to enhance the workflow of the development team.

Less Maintenance Overhead Autoconf uses different file types, whereas CMake only needs one. In addition, from our experience, the same build logic can be expressed with a higher abstraction level and much less code in CMake: In Section 5, we obtain roughly a reduction in code size of the MESSy build system of a factor of ten.

Simplified Support for Different Systems MESSy is used on different HPC clusters with various software environments and potentially different computer architectures. CMake simplifies building software on different hardware or operating systems. In the new build system, only one small file is added per cluster and compiler which defines the required software stack.

Simplified out-of-Source Builds The new build system allows compiling the code in a dedicated build directory (out-of-source build). That makes it possible to use several build configurations for the same checked-out repository locally in parallel. Moreover, the source code in the repository is not spoiled by generated files.

Autoconf is in general also capable of this. However, CMake does this by default in a simplified manner. Moreover, the present Autoconf implementation in MESSy is not designed to manage this due to hard coded paths containing mandatory auxiliary files. To generate an out-of-source build within MESSy's old system, modifications are necessary either way.

Library Dependency Handling CMake handles dependencies to external and internal libraries similarly. In addition, CMake has the concept of `PUBLIC`, `INTERFACE` and `PRIVATE` dependencies, see `target_link_libraries` in [CMA]. This concept greatly simplifies defining the dependencies between larger hierarchies of libraries. With Autoconf, one needs to manually propagate variables with required combinations of include and linker flags.

Dependencies to Third-Party Software It is simple to use third-party software that also comes with a CMake build system. CMake also supports dependencies that use other build systems, for example if their installation provides a `pkg_config` file. Dependency handling is more tedious and less standardized with Autoconf from our experience and many software packages for HPC clusters support and use CMake. Additionally, CMake has a large community and is being actively developed and improved regularly.

User Interface CMake has a command-line user interface, `ccmake`, to get an overview of all possible variables and to manipulate configuration options.

3 Creating the CMake Build System

This section focuses on our process to implement the new MESSy build system. As a preliminary, we start with a short overview on the usual software stack and workflow for compiling software on an HPC cluster. Afterward, we introduce the initial situation and first steps to gain an overview over the software. Then, we briefly describe some basics of the Autoconf build system and how to "translate" them into CMake. After that, we discuss our process of setting up the new build system. We start with a general workflow and a minimal configuration before moving over to extending it step by step to reach the final production stage.

3.1 Software Stack on HPC Clusters

Most current HPC clusters use some Linux variant as operating system, see statistics in [top25]. The computing center then provides software including compilers and libraries for the cluster using a module system as introduced in [Fur91] and developed further by, e.g., [GHM14] and [GLC+15]. The idea behind this is that on the one hand, users need different combinations of third-party software, possibly in different variants and versions. While on the other hand, all software must be tailored to the cluster and configured correctly for using the parallelism of the cluster. The user then "loads" the modules, which will adjust environment variables and paths such that build systems can find the desired software. A minimal example from MESSy consists of at least a compiler (here: Intel compilers), an MPI library, and the netCDF library, see Figure 1.

```
1 module load intel-oneapi-compilers
2 module load openmpi
3 module load netcdf-c
4 module load netcdf-fortran
```

Figure 1: Minimal example for loading required environment modules for MESSy.

The module system ensures that all loaded modules are compatible, e.g., in the background several netCDF variants are available. The previously selected compiler and MPI version then determine which one is loaded.

3.2 Preparation

In the following, we describe our first steps to tackle a huge software such as MESSy. For all these, one needs to consider that the MESSy repository contains about 32,000 files, out of which 15,000 contain source code and about 5,000,000 lines of code as seen from Table 1. So for anyone without a very deep understanding of the whole software, it is not clear where to start.

3.2.1 Initial Situation

We can follow the structure of the existing build system and analyze its configuration options. It is not mandatory to have a fully productive build system for all cases at the start, but it is necessary to know how the software needs to be built. MESSy does not provide automated tests. To compare simulation results, one has to submit a jobscript on an HPC cluster. The provided jobscript is a shell script of about 7,000 lines of code that the user needs to adjust for different configurations and different systems. The job script handles the interaction between various executables of the software and copies required input data into the `workdir` directory. Therefore, we could not easily automate tests when improving the build system.

3.2.2 First Steps

Setting up a new build system, it is started with the easiest possible configuration that produces a valid working program. That means, all options, non-compulsory flags or similar are ignored

Table 1: Files in the MESSy repository analyzed with cloc [Dan21].

Language	files	blank %‡	comment %‡	code
Fortran 90/95	6,388	14	23	3,323,226
Shell Script	1,541	11	22	472,963
C	903	15	15	385,959
TeX	622	14	8	174,536
Fortran 77	708	4	41	155,596
Fortran Namelist	3,801	3	63	140,058
Fortran Header	395	10	15	135,134
m4/Autoconf	455	10	5	129,161
KPP tool⋆	86	2	21	126,361
XML	454	9	4	117,077
C++	97	15	10	84,265
C/C++ Header	718	15	26	75,882
Python	400	15	18	71,580
Perl	160	14	18	51,069
CUDA	130	15	31	38,475
Java	228	14	25	28,596
Other	1,433	15	12	95,387
SUM:	18,519	12	24	5,605,325

⋆ MESSy includes an older version of the KPP tool [SSL+25] which uses a domain specific language to describe systems of chemical reactions.
‡ Percentages are given in relation to the total number of lines.

at first. Also, it is only focused on one common compiler which will be the GNU compiler here.

To have a rough idea on how much code is needed for the minimal configuration, some numbers are listed here for the MESSy build system: We created 16 new files, of which three are needed as helper files for CMake to find third-party software packages. These files contain around 750 lines of code and less than 200 lines without the three helper files.

Ideally, software installations include helper files (`<PackageName>.config`) that allow CMake to find them. Alternatively, CMake provides helper files for detecting many common software dependencies (`Find<PackageName>.cmake`). For less common dependencies, one might need to write appropriate helper files. Here, we could reuse helper files from open-source projects (respecting their licenses) with similar dependencies.

To create the desired configuration, the mandatory source files to build the executable need to be identified. As mentioned in Section 1, MESSy has a modular structure, and therefore, many subdirectories exist. By default, the `ECHAM` "basemodel" and its executable `echam5` are built [JKP+10, RBE+06]. This is thus the only relevant part for the desired minimal configuration. The old build system helps here, as the (root) `Makefile` defines required steps.

The executable links against several libraries. They are needed for the minimal new build as well. One needs to distinguish between internal libraries built by our own software and external

ones, which need to be installed on the used system or loaded via the cluster module system. The latter ones should be somehow named in the old build system since they have to be loaded before using it, as well. Some brute force might be helpful to find the needed external libraries in such a big project: We also tried to build the software using the old build system with missing external libraries to detect possible errors during configuring, compiling or linking. Internal libraries, on the other hand, require more work. They come with their own separated source code. To understand how to build them, the respective old `Makefile` is helpful. At this point, it is also referred to Subsection 3.4.

Of course, one comes up with the question of what exactly to build. As stated earlier, the main work here lies in finding the key source code files and subdirectories. To begin with that, it is started with investigating the old build system's default `all` target and what it does and invokes.

3.2.3 Basics about the Autoconf Build System

To understand what the new build system has to be capable of, it is mandatory to know how the old build system works. Here, we will focus on the `Makefile` and `configure` file, since these form the core of MESSy's old build. Hence, for the mentioned `all` target, the focus is now on the very basics of the Autoconf build system.

Makefile A `Makefile` mainly consists of rules which start with the target and are defined by dependencies and recipes. A target is usually the name of the generated executable or object file. On the other hand, so-called `PHONY` targets exist which mark a recipe name to be executed for an explicit request, see [GNU].

Examples can be found in Figure 2, Figure 3 and Figure 4. The dependencies of a target are listed right after the target name and colon. Here, e.g., files or other targets can be listed on which the origin target depends. That means, if those are updated, the target is invoked again. The recipe, starting in the next code line, denotes the actual (shell) commands that are performed if the target is (re-)built.

3.2.4 Translating the Structure of the Old to the New Build System

After understanding how to read single `Makefile` targets, the focus is now on how to obtain the MESSy-specific structure that should be recreated in the new build.

As mentioned, the `all` target is the starting point. It is defined in the root `Makefile` and is shown in Figure 2. It has no recipe but two dependencies, namely `modlog` and `basemodels`, which are also targets. The former only produces some log files listing the loaded modules at execution time. For a minimal configuration, we only need the target `basemodels`.

```
1 .PHONY: all
2 all: modlog basemodels
```

Figure 2: The `all` target as defined in the MESSy Autoconf build system. It depends on the targets `modlog` and `basemodels`.

An excerpt can be seen in Figure 3. It depends on `libs` and furthermore executes several commands. Later on in the recipe, other files are invoked to build the actual executable `echam5`. The called `Makefile` in turn, includes, e.g., needed libraries, flags, compile options and paths to the source code. The latter determine the core part here and need to be implemented later in the new build. Depending on the user input at configuration time, other executables from different basemodels are built.

```
1 .PHONY: basemodels
2 basemodels: libs
3    @if [ "$$target" != "" ] ; then \
```

Figure 3: The (clipped) rule of the `basemodels` target as defined in the MESSy Autoconf build system. It depends on `libs`. Later in the recipe, commands are invoked to build the actual executable.

The `libs` target (cf. Figure 4) does not depend on any other files or targets. Its recipe iterates through the internal libraries' subdirectories and performs their respective `Makefile`, meaning that it builds the necessary libraries.

```
1 .PHONY: libs
2 libs:
3    @for DIR in $(LIBSRCS) ;\
4    do \
```

Figure 4: The (clipped) rule of the `libs` target as defined in the MESSy Autoconf build system. It has no further dependencies.

Now, the basic high-level structure of the old build system is found. Of course, there are much more (recursive) dependencies. An overview of the just discussed dependencies within the old build system can be found in Figure 5.

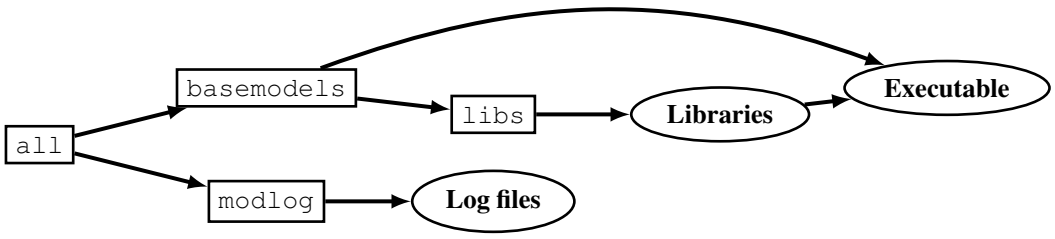

Figure 5: An excerpt of the resulting targets' dependency tree. It starts with the `all` root target. The dependent `modlog` target only produces log files. The relevant outcomes for a minimal configurations are the necessary **libraries** and the built **executable** at the end.

3.3 General Workflow

Before starting to set up a minimal configuration, visualizing a meaningful workflow is very helpful.

Comparing the Old and New Build It is important to keep the old build system while developing the new one. For that time, both systems should be maintained (and developed) in parallel, e.g., via using an extra repository branch. The results of similar actions—like building a specific meta target, executable, or library—have to result in the same outcome with both systems at the end. It is helpful to check out the repository twice in two different local directories. One for the old and one for the new build. That way, one can compare the outcome easily. This comparison should also include some production runs from time to time to, e.g., ensure that different configurations are correctly processed and that the executables can be started.

Continuous Integration Once we implemented a working minimal configuration, we wanted to ensure that it keeps working while we extend it. For this, we set up continuous integration (CI) tests [Fow24] using GitLab-CI [Git25]. Especially at the beginning stage, the new build might be sensitive regarding changes of any kind. To detect possible bugs early and make the system more robust, each code change should be checked on whether the CMake build still works. To simplify the setup of the CI system and to avoid interfering with the regular MESSy development, we used a mirrored repository and our (mini) test cluster for the development of the CMake build system.

Documentation It is very helpful to compose a documentation in parallel to record the coarse structure and usage examples for the build process. Hence, we created a markdown file named `README-cmake.md`. This file should for example also list important configuration options, buildable targets and supported compilers. In addition to that, it allows communicating further software-specific information, respectively introducing the developers to the CMake build system. Doing so right from the start is the easiest way to record the information for other users. Additionally, the hurdle is lowered for MESSy's users without experience in CMake to switch and maybe even expand it on their own.

Later as the developing proceeds, more and more features from the old build system are translated to the new one, cf. Subsection 3.6.

3.4 A Minimal Configuration

Up to now, the rough structure of the old build system is known to build a working executable. In this subsection, a minimal configuration to build MESSy using CMake will be discussed by implementing the mentioned structure. For this, we will show some files from the new build system, the root `CMakeLists.txt` file in Figure 6 and the one to configure the executable in Figure 7. There is also a file listed to configure an exemplary internal library in Figure 8.

Root CMake File In the root path of a project, there is one special `CMakeLists.txt` that serves as entry point for CMake. In addition, it defines the minimally required CMake version, sets the project name and usually enables required programming languages, see Figure 6. To

ensure compatibility across the desired version range, we set up a matrix CI job that tests building with all (major) CMake versions since our minimal requirement. For this, we installed different CMake versions using the Spack [GLC+15] package manager.

Auxiliary files for the CMake build are commonly stored in an extra `cmake` directory on the root level. Those files are, e.g., needed when searching for external packages during the configuration stage. For CMake to detect these files, the variable `CMAKE_MODULE_PATH` has to be appended by that path, see line 5 in Figure 6. The variable `CMAKE_CURRENT_SOURCE_DIR` in CMake points to the directory of the currently processed `CMakeLists.txt`. In addition, we add some compiler flags and definitions in lines 7–10. These are needed for compiling some Fortran files with the current Fortran compiler from the GNU compiler collection (GCC) [GFo].

After that, in lines 12–14 of Figure 6, we include a few mandatory third-party libraries using the `find_package` command. Variables for using them are now known globally in the project. Then, the `config.h` file is generated from a template within the repository's `config` directory and placed into the build directory. The variable `CMAKE_CURRENT_BINARY_DIR` points to the corresponding subdirectory of the current `CMakeLists.txt` in the build directory.

Now, we recreate the structure of the `Makefile` targets depicted in Figure 5: In lines 22–25, we include the corresponding `CMakeLists.txt` files from subdirectories. The directory `libsrc` contains further subdirectories with multiple required internal libraries. This is equivalent to the `libs` target in Figure 4 from the old build system. The directories `messy` and `mpiom` contain source code and configure corresponding libraries. The directory `echam5` contains the source code for the desired executable, which is linked to all the previous targets.

The Executable's CMake File The corresponding file is shown in Figure 7. In lines 1–4, the required source files are defined and stored in a CMake variable. There is one file from the respective directory that needs to be excluded from the build (line 4), see discussion in Subsection 4.2. Following that in line 6, the executable `echam5` is defined. CMake automatically adds the language-specific compiler flags (see line 8 in Figure 6) and it automates the whole compilation process in the generated `Makefile`. In lines 8–9, required include directories and a preprocessor definition are added to the target. The executable also has to be linked against the aforementioned libraries, see lines 10–11. All of them except one (`LAPACK`) are internal and are defined in the subdirectories included previously in the root `CMakeLists.txt`. Finally, some source files require an additional compiler flag which is added in lines 13–15.

Example for an Internal Library To complete the explanation of a working minimal configuration, we present one of the libraries' `CMakeLists.txt` in Figure 8 and compare it to the old corresponding Autoconf counterpart (see Figure 9). This also illustrates that CMake allows a much shorter representation of the same build logic: The required CMake code for the `isorropia_lite` library (code based on [NPP98]) only encompasses two lines of code. The corresponding `Makefile` from the old build system (not generated!) needs almost 20 lines. The situation is similar for the other libraries included in the `libsrc` directory in MESSy.

```
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
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)
```

Figure 6: The root `CMakeLists.txt` for a minimal configuration.

```
file(GLOB ECHAM5_SRC RELATIVE ${CMAKE_CURRENT_SOURCE_DIR}
   src/*.f90 modules/*.f90 ../messy/bmil/*.f90 ../messy/echam5/bmil/*.f90
   ../messy/echam5/smil/*.f90 ../messy/smil/*.f90)
list(REMOVE_ITEM ECHAM5_SRC ../messy/bmil/mo_mpi.f90)

add_executable(echam5 ${ECHAM5_SRC})
target_include_directories(echam5 PRIVATE include ../messy/bmil)
target_compile_definitions(echam5 PRIVATE ECHAM5)

target_link_libraries(echam5
   mpiom messy support isorropia_lite isorropia_v23 qhull crm LAPACK::LAPACK)

set_source_files_properties(modules/mo_netcdf.f90
   modules/mo_netcdfstream.f90 modules/mo_grib.f90
   PROPERTIES COMPILE_OPTIONS "-fallow-argument-mismatch")
```

Figure 7: The main executable's `CMakeLists.txt` for a minimal configuration.

```
file(GLOB ISORROPIA_LITE_SRC *.f)
add_library(isorropia_lite ${ISORROPIA_LITE_SRC})
```

Figure 8: An exemplary `CMakeLists.txt` for the library `isorropia_lite` included in the MESSy repository.

```
srcdir = .
LIB = ../../../lib/libisorropia_lite.a

SRCS = isocom.f isofwd.f isorev.f

OBJS := $(SRCS:.f=.o)

all: $(LIB)

install: all

$(LIB): $(OBJS)
   $(AR) $(ARFLAGS) $(LIB) $(OBJS)
.SUFFIXES: $(SUFFIXES) .f

%.o: %.f
   $(FC) $(FFLAGS) -c $<

clean:
   rm -f ./*.o ./*~ ./*.mod

distclean: clean
   rm -f ./$(LIB)
```

Figure 9: The handwritten `Makefile` for the necessary library `isorropia_lite` from the old build system. In the MESSy repository, there does not exist any template for generating it.

3.5 Adding Flexibility

The previous subsections discussed a minimal configuration which only works with one compiler and only builds one specific base configuration. The following steps add support for different compilers and environments.

3.5.1 Module Load Files

A "module load file" as used here is a `shell` script sourced by the user for loading the required external libraries into the current shell environment and looks mostly like Figure 1. It includes required `module load <...>` commands to prepare the user environment on an HPC cluster. They also allow setting additional environment variables, such as paths to data files that vary from system to system. This ensures that everyone uses identical environments on the same system and makes the development workflow simpler.

The old `configure` script includes different files based on the current host name. These

files define environment variables for paths as well as for compiler flags. Using the host name prevents calling configure on a system not supported by the MESSy developers.

With our concept for the new build system, the user first sources the "module load file" and then runs CMake. Compiler flags are handled independently of the target HPC cluster in CMake, see Subsubsection 3.5.2. This makes the "module load files" much shorter and simpler to understand than the previously needed files with hundreds of compiler flags and more than 20 different environment variables. To give some hints on relevant options, the "module load file" prints some helpful message as shown in Figure 10.

```
# user help
echo "to compile, use:"
echo " cd build"
echo " cmake .. -DCMAKE_BUILD_TYPE=Release"
echo " make -j 128"
```

Figure 10: Example text at the end of the `Modulefile` to present the building steps to the user.

Using the new approach, it is easy to build the software on other HPC clusters. One can directly call CMake and then obtains errors for missing dependencies. And one can prepare a similar "module load file" which should be short and straight forward to do using the existing ones and the documentation of the HPC cluster.

3.5.2 Compiler Flag Files

Depending on the detected compiler (CMake variable `CMAKE_<LANG>_COMPILER_ID`), the root `CMakeLists.txt` includes a different CMake file that manipulates CMake variables to adjust compiler flags, see Figure 11. This is mostly necessary as the Fortran code in MESSy

```
# Fortran flags depend on used compiler (imported from file)
if(CMAKE_Fortran_COMPILER_ID STREQUAL GNU)
   message(STATUS "GNU compiler (gfortran) detected,
                  using GNU flags (cmake/GNU_Fortran_flags.cmake).")
   include(cmake/GNU_Fortran_flags.cmake)
elseif(CMAKE_Fortran_COMPILER_ID STREQUAL IntelLLVM)
```

Figure 11: Excerpt from the root `CMakeLists.txt` to include the proper "compiler flag file".

needs specific and possibly uncommon compiler flags. For many common settings such as choosing the C++ standard, CMake directly provides an abstraction layer that chooses suitable compiler flags. Moving these compiler-specific settings into corresponding helper files keeps the main `CMakeLists.txt` readable and allows for easily adding new compilers. There is one "compiler flag file" for each supported compiler vendor (GNU [GFo, GCC], Cray [CRAb, CRAa], Intel (classic [IFO, ICC] and LLVM-variant [IFX, ICX]), NAG [NAG], PGI [PGI], NVHPC [NVH]) and programming language (Fortran and C/C++).

In Figure 12, two CMake variables are manipulated: The first `set` appends two flags to the existing variable `CMAKE_Fortran_FLAGS` which defines the default compiler flags for all

```
1  # allow longer code lines
2  set(CMAKE_Fortran_FLAGS
3    "${CMAKE_Fortran_FLAGS} -ffree-line-length-none -fno-second-underscore")
4
5  # flag to switch real to double-precision as default
6  set(Fortran_FLAGS_DEFAULT_REAL8 "-fdefault-real-8")
```

Figure 12: Excerpt of the "compiler flag file" `GNU_Fortran_flags.cmake`.

Fortran files. The second `set` creates a new variable and sets it to one flag. This variable is used later to set specific flags for specific targets or files and replaces hardcoded flags in the minimal example such as in line 15 of Figure 7.

As previously done for different CMake versions in Subsection 3.3, we set up CI jobs for different compilers and compiler versions to ensure that the software at least builds in all variants (such as GCC 5–13, two Intel compiler versions, one NVHPC version, all installed via Spack [GLC+15]). Due to missing licenses, we could not automatically check some of the other compilers on our system.

3.6 Extending the Build System

After generalizing the minimal configuration for different HPC clusters and compilers, the next step is to extend the build system to include all options and targets from the old build system. The goal is to obtain a new build which can be used by the MESSy developers as a full replacement of the old build system.

3.6.1 Compiler Flags Revised

For the minimal variant, we only checked that all files are compiled into a working binary. However, to ensure correctness and trustworthiness of climate simulations, the MESSy software is carefully designed to produce binary identical simulation results. So in the development process of MESSy, modifications that numerically change results of reference simulation cases are usually not allowed. Exceptions to that require an exhaustive manual comparison of simulation results. And changing the build system should obviously not lead to a loss of binary identity under otherwise equal conditions. However, the first minimal build did. In addition, identical compiler flags ensure that the code optimization and run-time performance is not affected.

For that reason, we carefully compared the compiler flags of the old and the new build: This is non-trivial, as the ordering of commands and the ordering of compiler flags differs for both build systems. Unfortunately, we did not find a good automatic way to perform this comparison which made this a very time-consuming process.

Our final solution was as follows: We first ran the old build and the new build (serially) and captured the corresponding commands and outputs in two text files. For the CMake build, running `make VERBOSE=1` or enabling the option `CMAKE_VERBOSE_MAKEFILE` allows printing all executed compiler commands. We then sorted and compared the two text files blockwise for matching source code compile commands. Here, we manually replaced equivalent combina-

tions of compiler flags. Through this approach, we could detect and remove the differences in compiler flags that led to numerical deviations of the simulation results.

3.6.2 Adding Configuration Options

Up to this point, only the default configuration of the old build system is recreated. As next step, configuration options are considered that allow customizing the build. In most cases, these options set different compiler flags or add source code and introduce new libraries.

The command `./configure -h` displays all possible options for the old MESSy build. This list was maintained by the developers and includes a description of each option. The corresponding code in the `configure` and `Makefile` files provides necessary information. In the following, we discuss the option `ASYNCF` as an example, see Figure 13.

```
1  if test "$enable_ASYNCF" = "yes"; then :
2    { $as_echo "$as_me:${as_lineno-$LINENO}: result: enabled" >&5
3  $as_echo "enabled" >&6; }
4            ASYNCF_DEF="HAVE_ASYNCF"
5            LIBSRCS+=" libsrc/async-fortran/build"
6            MESSY_LIB+=" -lasyncf"
7            MEXT_LIB+=" $LIBCXX"
8  else
9    { $as_echo "$as_me:${as_lineno-$LINENO}: result: disabled" >&5
10 $as_echo "disabled" >&6; }
11           ASYNCF_DEF=""
12 fi
```

Figure 13: Excerpt from the configure file to define the configure option `enable-ASYNCF`.

If the option is set at configuration time, one variable (`HAVE_ASYNCF`) is set and some are extended: To build the additional library `asyncf`, the `LIBSRCS` variable consisting of all internal library source code directories is expanded by the path to the corresponding `Makefile.m`. This in turn, then calls and generates the library. After that, the library is added to the variable `MESSY_LIB` that stores required linker flags.

One can translate the above option step-by-step to CMake as shown in Figure 14. It starts with defining the configuration option for CMake using the `option` command and requires a description and a default value (`OFF`). The option can be set by the user at configuration by calling `cmake` with `-DASYNCF=<ON/OFF>` or via `ccmake`.

If the option is activated, CMake behaves roughly the same as the old build: The command `add_subdirectory` calls the CMake file in the given path to configure and later build the library. To reach the same flexibility as the Autoconf build provided in this case, the two variables `ASYNC_COMPILE_DEF` and `MESSYLIBS_OPTIONAL_Fortran` are edited afterward. This approach also evades additional `if` / `else` logic when the library could potentially be used. If deactivated, the variable `ASYNC_COMPILE_DEF` is cleared.

To use the optional library in another build target, we show the corresponding CMake commands in Figure 15. The first one sets the preprocessor definition, the second one links the target against the new library.

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

Figure 14: Excerpt from root `CMakeLists.txt` to define the CMake option `HAVE_ASYNCF`. Translates the Autoconf option from Figure 13 into CMake syntax.

```
target_compile_definitions(targetName ${ASYNCF_COMPILE_DEF})
target_link_libraries(targetName ${MESSYLIBS_OPTIONAL_Fortran})
```

Figure 15: Include the optional `asyncf` library from Figure 14 into other libraries/executables.

This procedure has to be repeated for all other configuration options that should be implemented in the new build. For each added option in CMake, we recommend adjusting the corresponding documentation (`README-cmake.md`).

3.6.3 Adding Further Build Targets

Until now, we focused on the default build target, the `all` target, in the `Makefile` of the old build. There are several other targets such as `docu`, `mbm` and `tools`. The targets `mbm` and `tools` build new executables, namely 41 and 13, respectively. One has to follow all targets recursively at this point and translate them step by step to CMake to build all the additional binaries. Yet, they turned out to be less complex to build.

The `docu` target builds the LaTeX-documentation of the software and its theoretical background. As the final step, we again extended the `README-cmake.md` to describe all supported new targets in the documentation.

3.6.4 MESSy "basemodels"

In the discussion on configuration options in Subsubsection 3.6.2, we skipped the option that selects which MESSy "basemodels" to build. This option enables (or disables) building large parts of the codebase and up to this point, we only considered the default "basemodel" (`ECHAM5`). Another comparably extensive "basemodel" is `COSMO` [KJ12, RWH08]. So to complete the functionality of the new build system, the steps previously performed for `ECHAM5` are repeated for all other "basemodels". This was a time-consuming process but followed the general workflow described above.

As sole exception, for the basemodel `ICON` [ZRRB14, KJ16] only rudimentary CMake sup-

port is provided in MESSy. The software `ICON` is actively developed outside MESSy and the MESSy developers plan to outsource the software from the main repository. Therefore, spending too much work on translating its build system in MESSy is not advisable at this point. A future approach might be to use the CMake `ExternalProject` feature [CMA].

3.6.5 Unit Tests

Before, MESSy did not use automated unit tests. As an additional step, the pFUnit test framework [pFU] was brought into the new build system. Currently, only examples for tests exist as it is out of the scope of our current project to write unit tests for all the existing code in MESSy.

4 Learnings

In this section, we discuss obstacles we encountered and aspects we learned.

4.1 General Aspects

Working with both Build Systems in Parallel It is essential to compare the outcome of the old and new build during the development. For that, both systems are needed in parallel. However, we do not recommend executing both builds in the same local directory, in particular as the Autoconf build generates new files in the source tree.

To give an example: At some point when building MESSy, a source file generator has to be executed to create new Fortran files from template Fortran files with the same suffix. When the new build is performed after the old one, it also processes the previously generated files in the source tree (due to our `file(GLOB ...)` approach). This results in strange compilation errors that are difficult to track down.

Reusing Shell Commands There is often a simple and direct way to translate Autoconf commands into CMake. If this is not the case, one can use shell scripts or snippets, e.g., for code generation, in the CMake build via one of the following functions: `add_custom_target(...)`, `add_custom_command(...)`, or `execute_process(...)`, see the corresponding parts in the CMake documentation [CMA].

CMake Syntax The syntax of the CMake language is somewhat cumbersome in some places. In particular, string variables behave differently compared to configure scripts, for example concerning escaping of characters and concatenation. Similarly, CMake has an uncommon way of passing function arguments and returning their results.

Trial and Error At some places, a trial-and-error approach is helpful, for example to figure out required dependencies and to get an understanding of parts of the old build without reading through all its code.

4.2 MESSy-specific Learnings

Creating a new build system for MESSy was at the end more time-consuming than initially expected, partly due to several MESSy-specific hurdles:

Filenames Appearing Several Times In the example Subsection 3.4 in Figure 7, a source file was removed from the list of globbed files for compiling the `echam5` executable. The reason for that is as follows: In MESSy, many files appear more than once in the source directory tree using symbolic links. If the same file is compiled and linked twice, this can result in linker errors. For Fortran code containing a `module`, this also creates compiler errors due to ambiguous module names. To evade this problem, the respective `Makefile` has to be studied precisely to know at which place the particular file should be processed. After the migration to CMake, the MESSy developers started to remove duplicate files as CMake makes it easier to reuse code from different directories in the repository.

CMake Dependency Checker CMake automatically checks dependencies to set up a meaningful build-chain for each resulting `Makefile`. This was not supported directly by the old Autoconf build for Fortran modules but needed a perl script maintained in MESSy. So moving to CMake has the benefit that CMake directly supports Fortran module dependencies. However, in some places the CMake dependency checker for Fortran did not recognize dependencies correctly. This yields non-deterministic errors when building in parallel and might or might not trigger an error when building serially. The exact reason was that CMake does not parse Fortran `USE` statements correctly if they are interrupted by preprocessor directives as illustrated in Figure 16a. We resolved this problem by adjusting the respective places in the source code (which were non-trivial to find) to the format shown in Figure 16b.

```
1    USE messy_main_grid_def_bi, ONLY: &
2 #ifdef DIUMOD_DEBUG
3                            philat_2d, &
4 #endif
5                            philon_2d, sinlat_2d, coslat_2d
```

(a) In the original source code, a preprocessor macro interrupts the `USE`. CMake does not seem to recognize dependencies correctly in this case.

```
1    USE messy_main_grid_def_bi, ONLY: philon_2d, sinlat_2d, coslat_2d
2 #ifdef DIUMOD_DEBUG
3    USE messy_main_grid_def_bi, ONLY: philat_2d
4 #endif
```

(b) In the adapted code, the split of the `USE` command is resolved.

Figure 16: Example for an issue of the dependency checker regarding `USE` statements.

Naming Conflicts for Include Files The `configure` script generated by Autoconf commonly creates a header file named `config.h`. This header file contains preprocessor definitions, e.g., from detected available dependencies and supported programming language feature. For the new build, we reconstructed this file using CMake. Unfortunately, there are several places in MESSy where different `config.h` files are generated in different subdirectories. In addition, the source directory tree itself also contains non-generated files named `config.h`. An example for this is the library `guess` in MESSy (originating from LPJ-GUESS [NAG+21]). This means that the ordering of compiler flags for include directories matters. And the resulting compilation errors for incorrect orderings are misleading and do not hint at file naming conflicts at all. Luckily, this can be simply fixed by changing the ordering of include directories using the keyword `BEFORE` in `target_include_directories` as illustrated in Figure 17—the problem is just tricky to track down.

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

(a) Add the relative path `framework` to the target `guess` as an include directory. By default, the path as appended.

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

(b) Prepend the relative path `framework` to the target `guess` as an include directory using the `BEFORE` keyword.

Figure 17: Two different ways to define an include directory depending on whether to append or prepend the path. The variant (b) ensures that the file `config.h` from the directory `framework` is used and not an identically named file in any of the dependencies.

External Python Package Naming Conflicts The function `find_package(...)`, as discussed above, is used in CMake to search for external dependencies. However, it might find the "wrong" location of a package in some cases. This often occurs when using Python environments on HPC clusters as both Python and the HPC cluster environment bring their own software stacks that are not compatible with each other.

As an example, when loading another module for adding a configuration option, CMake found a netCDF installation in this newly loaded module. However, this internal version was not built with the same dependencies (MPI or compiler) and configuration as the desired netCDF package. This again leads to errors that are difficult to analyze. As a workaround, the CMake variable `CMAKE_IGNORE_PATH` explicitly excludes directories when searching for packages. To avoid that the user has to specify this variable by hand, we added the environment variable `PATH_CONDA_IGNORE` to the "module load files" that is used to initialize this variable in CMake as shown in Figure 18. Unfortunately, for some `find_package` calls this path should be considered again. So we need to selectively set and unset this variable for different `find_package` calls. One could argue it is the task of the HPC cluster maintainers to help to avoid such cases but due to the complexity of the complete software stack, this currently seems difficult.

```
if(DEFINED ENV{PATH_CONDA_IGNORE})
  set(CMAKE_IGNORE_PATH
    "$ENV{PATH_CONDA_IGNORE};$ENV{PATH_CONDA_IGNORE}/bin")
  message(STATUS
    "Ignore the following path in find_package(): ${CMAKE_IGNORE_PATH}")
endif()
```

Figure 18: Excerpt from the root `CMakeLists.txt` to prevent finding a dependency in a wrong location.

5 Results

In the following, we describe the benefits of the new build system based on a few metrics and feedback from the developers.

Improved Maintainability As seen from Table 2, the amount of code for the build system roughly shrinks by a factor of ten. This is mainly due to the higher abstraction level offered by CMake. Additionally, CMake only needs one type of maintained files, the `CMakeLists.txt`, and not 2–4 different kinds of templates files like Autoconf. The root CMake file is also significantly smaller than the `configure.in` file and the root `Makefile.in`. This makes it easier to get an overview of the performed steps and, e.g., to add new configuration options.

Table 2: Comparison of some important metrics of the new CMake and the old Autoconf build system. The compile time measurements were performed on the DKRZ (German Climate Computing Center) Levante cluster with 2 AMD EPYC 7763 processors with 64 cores each and the command `make -j 64`, so effectively only 64 cores were used. The timings are averaged over three runs.

Metric	CMake build system	Autoconf build system
LOC (total)	$\sim 10{,}000$	$\sim 100{,}000^\star$
LOC (root files)	650	$\sim 2{,}200 + 1{,}200$
Build time (averaged)	180 s	290 s
$=$ configure	30 s	13 s
$+$ make	150 s	277 s
Recompile time (total)	3.5 s	15 s
File with many dependencies		

* Source code lines for Autoconf are just a rough estimate. It is not clear whether all files of the respective file types are actually used, in particular for included libraries; there are 95,000 lines of m4-type according to cloc [Dan21] and 53,000 lines in `Makefile.in` files.

Faster Build Process The total build time decreases significantly, which improves the development workflow, see Table 2. The configure time with CMake includes time-consuming checks for recreating the `config.h` file from Autoconf which might get optimized away in the future.

In the old build system, by default a timestamp argument was passed as a preprocessor definition to later see when the code was built. This caused a complete recompilation of all source files, even for small changes. So without adjusting a specific configuration option, recompile time was identical to the time for building from scratch. In discussions with the main developer, we argued that a faster development workflow (per default) outweighs the benefit of the timestamp. So we improved the default recompilation time by a factor of ~ 18 by removing the timestamp and by a factor of ~ 4 through using CMake.

Condensed "Module Load Files" We separated the setup of the user environment and the compiler flags into "module load files" and "compiler flag files". This reduces code duplication in the build system and avoids unintended differences in compiler flags on different systems.

Integration into the Development Workflow After supporting all desired features from the Autoconf build, the MESSy developers took over the maintenance of the new CMake build system. Currently, both build systems are supported in parallel but new features are only integrated into the new one. It is planned to remove the old build system after some time.

Feedback by the Developers One of the MESSy core developers, Patrick Jöckel, could substantiate our findings: After introducing the new build and using it more and more in production, maintenance, and expansion were simplified. Now, the builds are more flexible, yet faster and the developers (and more and more MESSy's users) get along very well with the new build system.

6 Conclusion

We implemented a new build system using CMake for the large mono repository of the earth system simulation software MESSy which includes about 4,000,000 lines of code mostly in Fortran. This build system replaces the existing Autoconf/Makefile based build system. The new build system needs about a factor of 10 fewer lines of code compared to the old one. It is easier to maintain and to extend and leads to faster compilation times. In addition, one can now easily build the software on other HPC clusters than the ones used by the main development team without deeper knowledge of the MESSy build system. So overall, we improved the development process of a large research software intended for HPC clusters.

Unfortunately, there are few funding opportunities for such software engineering tasks even though the involved scientists see their benefits. As next steps, we want to integrate unit tests into the workflow for newly written code in MESSy.

Final Thought MESSy contains code that is more than 30 years old and still works as programming languages are standardized (e.g., [ISO04, ISO99, ISO03]). However, there is no ISO standard for the build system and CMake has a fast release cycle. Hence, without modifying it, this code will most likely be unusable in another 30 years. So we see the strong need to standardize build systems to ensure long-term comparability of scientific results.

Acknowledgements: We thank the main developer of the MESSy software, Patrick Jöckel, and the whole MESSy team for their helpful feedback and fruitful cooperation. We also thank the anonymous reviewers for their helpful feedback and for the critical remarks that allowed us to clarify some aspects related to the previous Autotools build system in MESSy.

Bibliography

[CMA] CMake Documentation. https://cmake.org/cmake/help/latest/. Accessed: 2025-05-13.

[CRAa] Documentation of the Cray C/C++ Compiler. https://cpe.ext.hpe.com/docs/24.07/guides/CCE/index_cpp.html. Accessed: 2025-05-14.

[CRAb] Documentation of the Cray Fortran Compiler. https://cpe.ext.hpe.com/docs/24.07/guides/CCE/index_fortran.html. Accessed: 2025-05-14.

[Dan21] A. Danial. cloc: v1.92. Dec. 2021. doi:10.5281/zenodo.5760077

[Fow24] M. Fowler. Continuous Integration. Jan. 2024. accessed: 2025-05-10. https://martinfowler.com/articles/continuousIntegration.html

[Fur91] J. L. Furlani. Modules: Providing a flexible user environment. In *Proceedings of the fifth large installation systems administration conference (LISA V)*. Pp. 141–152. 1991.

[GCC] Documentation of the GNU C/C++ Compiler. https://gcc.gnu.org/onlinedocs/gcc/. Accessed: 2025-05-14.

[GFo] Documentation of the GNU Fortran Compiler. https://gcc.gnu.org/onlinedocs/gfortran/. Accessed: 2025-05-14.

[GHM14] M. Geimer, K. Hoste, R. McLay. Modern scientific software management using EasyBuild and Lmod. In *Proceedings of the First International Workshop on HPC User Support Tools*. HUST '14, p. 41–51. IEEE Press, 2014. doi:10.1109/HUST.2014.8

[Git25] Get started with GitLab CI/CD. 2025. accessed: 2025-05-10. https://docs.gitlab.com/ci/

[GLC+15] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, S. Futral. The Spack package manager: bringing order to HPC software chaos. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '15. Association for Computing Machinery, New York, NY, USA, 2015. doi:10.1145/2807591.2807623

[GNU] GNU make. https://www.gnu.org/software/make/manual/make.html. Accessed: 2025-05-12.

[ICC] Documentation of the icc Intel C/C++ Compiler (deprecated). https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-10/overview.html. Accessed: 2025-05-14.

[ICX] Documentation of the icx Intel C/C++ Compiler. https://www.intel.com/content/www/us/en/docs/dpcpp-cpp-compiler/developer-guide-reference/2025-1/overview.html. Accessed: 2025-05-14.

[IFO] Documentation of the ifort Intel Fortran Compiler (deprecated). https://www.intel.com/content/www/us/en/docs/fortran-compiler/developer-guide-reference/2024-1/overview.html. Accessed: 2025-05-14.

[IFX] Documentation of the ifx Intel Fortran Compiler. https://www.intel.com/content/www/us/en/docs/fortran-compiler/developer-guide-reference/2025-1/overview.html. Accessed: 2025-05-14.

[ISO99] ISO. *ISO/IEC 9899:1999: Programming Languages — C*. International Organization for Standardization, Geneva, Switzerland, Dec. 1999. http://www.iso.ch/cate/d29237.html

[ISO03] ISO. *ISO/IEC 14882:2003: Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, 2003. http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=38110

[ISO04] ISO. *ISO/IEC 1539-1:2004 Information technology — Programming languages – Fortran – Part 1: Base language*. International Organization for Standardization, Geneva, Switzerland, 2004. https://www.iso.org/standard/39691.html

[JKP+10] P. Jöckel, A. Kerkweg, A. Pozzer, R. Sander, H. Tost, H. Riede, A. Baumgaertner, S. Gromov, B. Kern. Development cycle 2 of the Modular Earth Submodel System (MESSy2). *Geoscientific Model Development* 3:717–752, Dec. 2010. doi:10.5194/gmd-3-717-2010

[JSK+05] P. Jöckel, R. Sander, A. Kerkweg, H. Tost, J. Lelieveld. Technical Note: The Modular Earth Submodel System (MESSy) - a new approach towards Earth System Modeling. *Atmos. Chem. Phys.* 5:433–444, Feb. 2005. doi:10.5194/acp-5-433-2005

[KE02] G. Kumfert, T. Epperly. Software in the DOE: The Hidden Overhead of ”The Build”. Technical report, Lawrence Livermore National Lab. (LLNL), Livermore, CA (United States), 2002. doi:10.2172/15005938

[KJ12] A. Kerkweg, P. Jöckel. The 1-way on-line coupled atmospheric chemistry model system MECO(n) – Part 1: Description of the limited-area atmospheric chemistry model COSMO/MESSy. *Geoscientific Model Development* 5:87–110, Jan. 2012. doi:10.5194/gmd-5-87-2012

[KJ16] B. Kern, P. Jöckel. A diagnostic interface for the ICOsahedral Non-hydrostatic (ICON) modelling framework based on the Modular Earth Submodel System (MESSy v2.50). *Geoscientific Model Development* 9:3639–3654, Oct. 2016. doi:10.5194/gmd-9-3639-2016

[MAN+11] S. McIntosh, B. Adams, T. H. Nguyen, Y. Kamei, A. E. Hassan. An empirical study of build maintenance effort. In *2011 33rd International Conference on Software Engineering (ICSE)*. Pp. 141–150. 2011. doi:10.1145/1985793.1985813

[MED23] D. MacKenzie, B. Elliston, A. Demaille. Autoconf: Creating Automatic Configuration Scripts. Free Software Foundation, Inc., Dec. 2023. Version 2.72. https://www.gnu.org/savannah-checkouts/gnu/autoconf/manual/

[MESa] MESSy. https://messy-interface.org/. Accessed: 2025-05-16.

[MESb] MESSy: Publications in 2024. https://messy-interface.org/publications/. Accessed: 2025-05-09.

[MM18] G. Maudoux, K. Mens. Correct, Efficient, and Tailored: The Future of Build Systems. *IEEE Software* 35(2):32–37, 2018. doi:10.1109/MS.2018.111095025

[MNA+14] S. McIntosh, M. Nagappan, B. Adams, A. Mockus, A. E. Hassan. A Large-Scale Empirical Study of the Relationship between Build Technology and Build Maintenance. *Empirical Software Engineering* 20(6):1587–1633, Aug. 2014. doi:10.1007/s10664-014-9324-x

[MTD+25] D. MacKenzie, T. Tromey, A. Duret-Lutz, R. Wildenhues, S. Lattarini. GNU Automake. Free Software Foundation, Inc., June 2025. Version 1.18.1. https://www.gnu.org/software/automake/manual/

[NAG] Documentation of the NAG Fortran Compiler. https://support.nag.com/nagware/np/r72_doc/nagfor.html. Accessed: 2025-05-14.

[NAG+21] J. Nord, P. Anthoni, K. Gregor, A. Gustafson, S. Hantson, M. Lindeskog, B. Meyer, P. Miller, L. Nieradzik, S. Olin, P. Papastefanou, B. Smith, J. Tang, D. Wårlind, , past LPJ-GUESS contributors. LPJ-GUESS Release v4.1.1 model code. Oct. 2021. doi:10.5281/zenodo.8065737

[NIN24] The Ninja build system. https://ninja-build.org/manual.html, May 2024. Accessed: 2025-05-15.

[NPP98] A. Nenes, S. N. Pandis, C. Pilinis. ISORROPIA: A new thermodynamic equilibrium model for multiphase multicomponent inorganic aerosols. 1998. doi:10.1023/A:1009604003981

[NVH] Documentation of the NVIDIA HPC Fortran, C and C++ Compiler. https://docs.nvidia.com/hpc-sdk/. Accessed: 2025-05-14.

[pFU] Parallel Fortran Unit Testing Framework. https://github.com/Goddard-Fortran-Ecosystem/pFUnit. Accessed: 2025-05-14.

[PGI] Documentation of the PGI HPC Fortran, C and C++ Compiler. https://docs.nvidia.com/hpc-sdk/pgi-compilers/20.4/x86/fortran-ref-guide/index.htm. Accessed: 2025-05-14.

[RBE+06] E. Roeckner, R. Brokopf, M. Esch, M. Giorgetta, S. Hagemann, L. Kornblueh, E. Manzini, U. Schlese, U. Schulzweida. Sensitivity of Simulated Climate to Horizontal and Vertical Resolution in the ECHAM5 Atmosphere Model. *American Meteorological Society* 19:3771–3791, Aug. 2006. doi:10.1175/JCLI3824.1

[RWH08] B. Rockel, A. Will, A. Hense. The Regional Climate Model COSMO-CLM (CCLM). *Meteorologische Zeitschrift* 17(4):347–348, 08 2008. doi:10.1127/0941-2948/2008/0309

[SNH+12] R. Suvorov, M. Nagappan, A. E. Hassan, Y. Zou, B. Adams. An empirical study of build system migrations in practice: Case studies on KDE and the Linux kernel. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. Pp. 160–169. 2012. doi:10.1109/ICSM.2012.6405267

[SSL+25] A. Sandu, R. Sander, M. S. Long, R. M. Yantosca, H. Lin, L. Shen, D. J. Jacob. KineticPreProcessor/KPP: The Kinetic PreProcessor (KPP) 3.2.1. May 2025. doi:10.5281/zenodo.15350924

[top25] Top500 supercomputer site. June 2025. accessed: 2025-05-10. http://www.top500.org/

[ZRRB14] G. Zängl, D. Reinert, P. Rípodas, M. Baldauf. The ICON (ICOsahedral Non-hydrostatic) modelling framework of DWD and MPI-M: Description of the non-hydrostatic dynamical core. *Quarterly Journal of the Royal Meteorological Society* 141:563–579, Apr. 2014. doi:10.1002/qj.2378