ICON Community Interface „ICON ComIn“

Nils-Arne Dreier[1], **Kerstin Hartung**[*2], Patrick Jöckel[2], Astrid Kerkweg[3], Bastian Kern[2], Mariano Mertens[2], Florian Prill[4], Daniel Reinert[4], Daniel Rieger[4]

[1]Deutsches Klimarechenzentrum (DKRZ)
[2]Deutsches Zentrum für Luft- und Raumfahrt (DLR), Institut für Physik der Atmosphäre
[3]Forschungszentrum Jülich (FZJ), Institut für Energie- und Klimaforschung: Troposphäre/ Center for Advanced Simulation and Analytics (CASA)
[4]Deutscher Wetterdienst (DWD)
[*]kerstin.hartung@dlr.de

What are the applications of ComIn?

First ideas for use cases of the ComIn generalized interface to ICON:
- connect ICON to e.g. wave model, chemistry components, ocean or land model
- couple external models e.g. via YAC
- interface ICON variables with other frameworks, e.g. ECMWF's Atlas library, and with other programming languages like C/C++
- embed Python scripts, execute during the simulation
- (interpolated) model I/O
- additional diagnostics

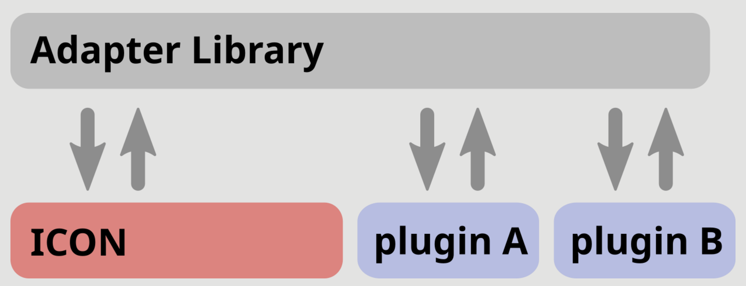
... and what is ComIn?

The community **interface** organizes the inclusion of **simulation events** from a plugin to ICON and **sharing of data** between the two. ComIn is not a coupler but the interface controls what, how and when foreign functions are called within ICON and data is exposed or exchanged. ComIn is actively developed by DWD, DLR, DKRZ and FZJ with consultative contributions from KIT.

Software prototype
- implemented in Fortran but interface design open for plugins in other languages through ISO-C bindings
- ComIn requires publishing a small part of ICON data structures in adapter library
- YAC adapter as one example ComIn plugin
- general aim: Open Source license

Design concept overview

- version and compatibility handling (backward compatibility)
- **adapter library**
  - read/write access to ICON variables
  - **descriptive data structures** shared from ICON (grid, parallelization info, simulation time stamp, ...)
  - plugins can create additional variables
- **callback register**, definition of entry points
- build process (independent from ICON's build process)



Schematic 1: The adapter library controls exchange of variables between ICON and plugins connected via ComIn.

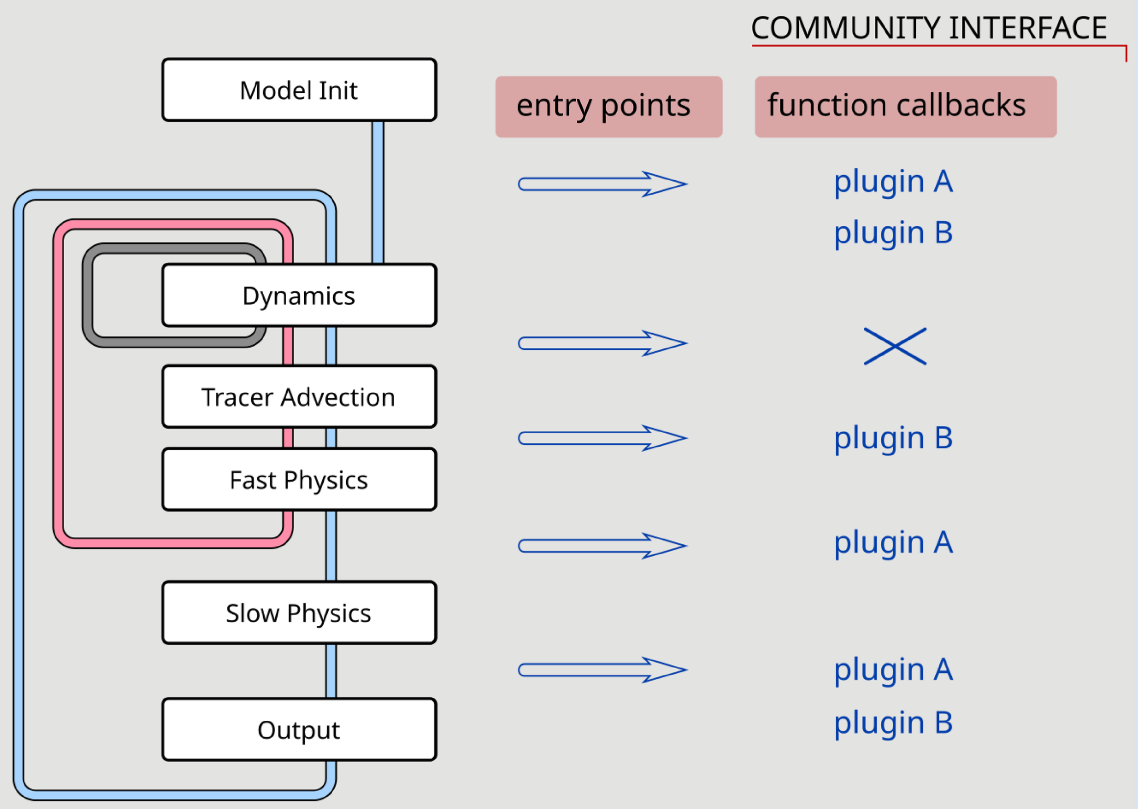
Specifics of the adapter library (at least for v1.0)

Request POINTERs to ICON variables for specific entry points.
- ComIn interfaces are restricted to cell-based variables
- blocked, process-local variable arrays are exposed as they are stored in ICON (information on location of vertical and blocking indices provided)
- multiple computational domains supported
- no variables with multiple time levels (instead POINTER swapping)
- plugins can READ, WRITE or READ/WRITE ICON variables, no safe-guarding against "malicious" write access
- plugins cannot change restart flag to ICON's standard fields

Plugins can also create additional variables (surface or 3D, tracers possible, can be requested exclusively, registration for inclusion in restart files).

How are the descriptive data organized?

Descriptive data structures contain information on the ICON setup, the computational grid(s), and the simulation status. They are part of the adapter library.
- all descriptive data structures are read-only
- **global data** (e.g. Fortran KIND values, MPI communicator) and **grid information** are available from the primary constructor
- **simulation status** available from the secondary constructor
- references (POINTERs) are preferred over values (copies), efficiency vs. safe-guarding against "malicious" write access



Schematic 2: Callbacks of plugins registered via ComIn are called at specified entry points within ICON.

How are callbacks organized?

Functions from plugins may be called at pre-defined events during a model simulation. One CALL in ICON handles execution of all registered plugins for one entry point. All function callbacks need to be registered in the primary constructor.
- two entry points before the time loop (see details on the right)
  - **primary constructor**
  - **secondary constructor** (optional, needs to be registered in primary constructor, called after allocation of ICON variable list and before time loop)
- several entry points during the **time loop**
  - all entry points above the "block loop level"

Roadmap

1. Finish the **prototype** (standalone version) ~ Q2/23
2. Initiate an internal **review phase**
   - interfaces, definition of entry points, documentation
   - fill mockup with "real information" in ICON feature branch
3. Test use of ICON via ComIn in first test application(s).

Some further decisions, e.g. testing procedure for external modules, are open.

How can ComIn be integrated into your application?

Primary constructor of plugin A

```
SUBROUTINE plugin_modA_setup(lrestart, comin_setup_version_info_out, ierr)
  LOGICAL,                            INTENT(IN)  :: lrestart
  TYPE(t_comin_setup_version_info),   INTENT(OUT) :: comin_setup_version_info_out
  INTEGER,                            INTENT(OUT) :: ierr

  INCLUDE "comin_version_info.f90"

  comin_setup_version_info_out = comin_setup_version_info
  ierr = 0
  IF (comin_setup_version_info%version_no_major > 1) THEN
    ierr = 1; RETURN
  END IF

  CALL get_comin_parallel_info(info)

  CALL comin_setup_activate_plugin(pluginname, plugin_setup, wp, ierr)
  IF (ierr /= 0)  RETURN

  CALL comin_request_add_var(t_comin_var_descriptor(jg = 1,              &
    &                  name = "myvariable"), t_comin_var_metadata(  &
    &                  ltracer = .FALSE., lmodexclusive = .FALSE.,  &
    &                  lrestart = .FALSE., itype_vlimit = 1,        &
    &                  itype_hlimit = 4, is_3d_field = .TRUE.), ierr)
  IF (ierr /= 0)  RETURN

  CALL comin_callback_register(EP_SECONDARY_CONSTRUCTOR, my_constructor, ierr)
  CALL comin_callback_register(EP_BEFORE_WRITE_OUTPUT,   my_diagfct, ierr)

  p_patch => comin_descrdata_get_patch_grid_info(1)
END SUBROUTINE plugin_modA_setup
```

- return ComIn version with which this module was built to ICON
- check own compatibility with ComIn version
- register plugin
- request additional ICON variable
- register function callbacks
- get descriptive data structures, e.g. grid information

Secondary constructor of plugin A

```
SUBROUTINE my_constructor()
  var_desc%name = 'press'
  var_desc%jg = 1
  CALL comin_var_get(EP_BEFORE_WRITE_OUTPUT, var_desc, FLAG_SYNCHRONIZED, press)
  var_desc%name = 'temp'
  CALL comin_var_get((/EP_SECONDARY_CONSTRUCTOR,EP_BEFORE_ADVECTION/), &
    &  var_desc, FLAG_SYNCHRONIZED, temp)
END SUBROUTINE my_constructor
```

- get pointer to ICON variable, updated at two entry points with READ/WRITE intent (synchronized)

Nth callback of plugin A

```
SUBROUTINE my_diagfct()
  WRITE (0,*) "data in callback:",press%ptr(1,1,1,1,1)
END SUBROUTINE my_diagfct
```

- data access through previously stored POINTERs

Entry point in ICON

```
#ifdef HAVE_COMIN
   !> example of a third-party entry point (callback)
   CALL comin_callback_context_call(EP_BEFORE_ADVECTION)
#endif /* ifdef HAVE_COMIN */
```

- preprocessor directives encapsulate ComIn calls
- one CALL for all plugins

Build process

Using **dynamic linking** of plugins (recommended approach):
1. Build ICON ComIn.
2. Build plugins (linking against ComIn library).
3. Specify plugins for primary constructor in *comin_nml*. Generates primary constructor calls automatically.
4. Build ICON with ComIn (--with-comin=${ICON_COMIN_DIR} during configure).

Alternative build process using **statically linked libraries:** In the third step the plugin, which should be statically linked, is not itself specified but "icon" is set as plugin. This allows (though with some effort) to link individual libraries statically to ICON.