OASIS4_1 User Guide

S. Valcke, CERFACS
M. Hanke, DKRZ
L. Coquart, CNRS

June 2011 -
CERFACS Technical Report
TR-CMGC-11-50

How to get assistance?

Assistance can be obtained by sending an electronic mail to oasis4_help(at)lists.enes.org.

Contents

Chapter 1

Acknowledgments

We would like to thank the main past or present developers of OASIS4 (in alphabetical order, with the name of their institution at the time of their contribution to OASIS):
Philippe Bourcier (CNRS)
Joseph Charles (CERFACS)
Laure Coquart (CNRS)
Damien Declat (CERFACS)
Jean-Marie Epitalon (CERFACS)
Josefine Ghattas (CERFACS)
Moritz Hanke (DKRZ)
Jean Latour (CERFACS)
René Redler (NEC, now MPI-M)
Hubert Ritzdorf (NEC)
Thomas Schoenemeyer (NEC)
Sophie Valcke (CERFACS)
Reiner Vogelsang (SGI)

Chapter 2

Introduction

The development of the fully parallel OASIS4 coupler started during the EU FP5 PRISM project to answer the needs of the European climate modelling community that was, at the time, starting to target higher resolution climate simulations on massively parallel platforms. The concepts of parallelism and efficiency drove OASIS4 developments, at the same time keeping in its design the concepts of low-intrusiveness and portability that made the success of OASIS3. Chapter 3 provides a more detailed description of OASIS4 sources and how to obtain them.

An ESM coupled by OASIS4 consists of different applications (or executables), each one hosting only one or more than one climate components (e.g. model of the ocean, sea-ice, atmosphere, etc.). After compilation, OASIS4 sources form a separate Driver/Transformer executable and a coupling interface library, the `PSMILe` that needs to be linked to and used by the components.

Each component must be provided with an XML[1] file that describes its coupling interface established through `PSMILe` calls. The configuration of one particular ESM simulation, i.e. the coupling and I/O exchanges that will be performed at run-time between the components or between the components and disk files, is also done through XML files. A Graphical User Interface (GUI), described in detail in the separate OASIS4-GUI User Guide, facilitates the creation of those XML files.

During the run, the role of the Driver/Transformer is to extract the configuration information defined by the user in the XML files, to organize the process management of the coupled simulation, and to perform the regridding needed to express, on the grid of the target components, the coupling fields provided by the source components on their grid. The OASIS4 Driver/Transformer is described in chapter 4.

The `PSMILe` , linked to the component models, includes a data exchange library which performs the MPI-based (Message Passing Interface, Snir et al. (1998)) exchanges of coupling data, either directly or via additional Transformer processes, and the GFDL mpp_io library Balaji (2001), which reads/writes the I/O data from/to files following the NetCDF format. The `PSMILe` and its Application Programming Interface (API) are described in chapter 5.

The structure and content of the descriptive and configuring XML files are then detailed in chapter 6. In chapter 7, instructions on how to compile and run the example toy coupled model TOYOA4 using OASIS4 are given; a toy model is an empty model in the sense that it contains no physics or dynamics. It reproduces, however, a realistic coupling in terms of number of component models, number, size and interpolation of the coupling fields, coupling frequencies, etc.

The originality of OASIS4 relies in its low intrusiveness, its great flexibility, and in its parallel neighbourhood search based on the geographical description of the process local domains performed by the `PSMILe` library.

[1]http://www.w3.org/XML

Chapter 3

OASIS4 sources

3.1 Warning and Copyright Notice

The software is in constant evolution and known bugs under consideration are detailed on the developers' wiki at: https://oasistrac.cerfacs.fr/report/1

OASIS4 offers interpolations and regriddings based on the Los Alamos National Laboratory SCRIP 1.4 library[1]. The SCRIP 1.4 copyright statement reads as follows:

[1]http://climate.lanl.gov/Software/SCRIP/

3.2 Reference

If you feel that your research has benefited from the use of the OASIS4 software, we will greatly appreciate your reference to the following report (Redler et al. (2010)):

R. Redler, S. Valcke and H. Ritzdorf, 2010: OASIS4 - A Coupling Software for Next Generation Earth System Modelling, Geoscience Model Development, 3, 87 - 104, DOI:10.5194/gmd-3-87-2010.

http://www.geosci-model-dev.net/3/87/2010/gmd-3-87-2010.pdf

3.3 How to obtain OASIS4 sources

It is interesting for the developers to know who is using the software and for which purpose. Therefore, to obtain instructions on how to download OASIS4 sources , the user first has to fill a registration form available at https://verc.enes.org/models/software-tools/oasis/download/oasis4-registration-form asking for the user identity, whether he/she wants to use OASIS3 or OASIS4, the component models he/she would like to couple with OASIS, whether it is a new coupled model or an upgrade of an exiting one, the target compute platform, and the project. The user can also tick a box if he/she agrees to appear on the OASIS download page and another box if he/she wants to subscribe to OASIS mailing list.

After submitting the form, the user will get detailed instruction on how to download the OASIS4 sources, either from the OASIS SVN server at CERFACS, memphis, or from CERFACS anonymous ftp site. The sources distributed are always the latest ones registered on the SVN trunk. The sources in the tar balls available from the ftp site are automatically update each day.

3.4 OASIS4 directory structure

3.4.1 OASIS4 sources

OASIS4 sources are divided into three directories under `oasis4/lib/` and one directory `oasis4/src`. With this structure, only a relatively small library *common_oa4* is used by both the OASIS4 Driver/Transformer executable and by the OASIS4 `PSMILe` coupling library The different directories are:

- `oasis4/lib/common_oa4/`: contains sources that are used both by the Driver/Transformer and the PSMILe coupling library. After compilation, these sources becomes the *libcommon_oa4.a* library.
- `oasis4/lib/mpp_io/`: contains the sources of the GFDL I/O library Balaji (2001). After compilation, these sources form the library *libmpp_io.a*. Compiling and linking this library to a component model is not mandatory if the PSMIle I/O functionality is not used (see compilation details in section 7).
- `oasis4/lib/psmile_oa4/`: contains the sources that form the main part of PSMILe coupling library and become, after compilation the library *libpsmile_oa4.a*.
- `prism/src/mod/oasis4/`: contains the main part of OASIS4 Driver/Transformer sources. Linked with the library *libcommon_oa4.a*, these sources form, after compilation, the OASIS4 Driver/Transformer executable named `oasis4.MPI1.x` or `oasis4.MPI2.x` (according to the choice of MPI1 or MPI2 done at compilation, see section 7 for details).

3.4.2 Other OASIS4 directories

In the `oasis4` directory, three more directories `/doc`, `/examples` and `/util` are found:

- `/doc`, contains OASIS4 User Guide.

- `/examples`, contains three toy examples:
  - toyoa4 : see details in section 7.3
  - toyoa4_restart : to generate the restart file for field COSENHFL in the toyoa4 example, see the README therein
  - tutorial1 : reproduces ping-pong exchanges between model1 and model2 with either the OASIS3 or OASIS4 coupler, with or without lag, in parallel or not; see the readme_tutorial1.pdf therein. This is probably the simplest toy model available to start learning about OASIS4.
- `/util` contains the following directories
  - creation_restart_oa4 : sources to generate restart files for coupled runs with OASIS4, by reading data in restart files used by OASIS3.
  - gui : the sources of the Graphical User Interface that can be used to generate the component description and configuration XML files (see section 6); a GUI User Guide is available in `oasis4/util/gui/doc`
  - `/make_dir` : top makefile and platform dependent header files for compiling OASIS4 (see section 7.2.1)
  - `/xmlfiles` : the SCHEMAs of the different XML files used with OASIS4 (see section 6)
  - `mppnccombine` : `mppnccombine.nc`, which may be used to join together NetCDF data files representing a decomposed domain into a unified NetCDF file.
  - `runscripts` : scripts to run the examples
  - `license` and `perl_script` : can be ignored as they contain scripts of interest for developers only.

Chapter 4

OASIS4 Driver/Transformer

OASIS4 Driver/Transformer tasks are described in this chapter to give the user a complete understanding of OASIS4 functionality. The realisation of these tasks at run-time is however completely automatic and transparent for the user. OASIS4 Driver/Transformer is parallel; the Driver tasks are performed by the master process only but the interpolation tasks are performed by all.

4.1 The Driver part

The first task of the Driver is to get the process management information defined by the user in the SCC XML file (see section 6.4). The information is first extracted using the libxml C library [1], and then passed from C to Fortran to fill up the Driver structures.

Once the Driver has accessed the SCC XML file information, it will, if the user has chosen the `spawn` approach, launch the different executables (or applications) that compose the coupled model, following the information given in the SCC file. For the `spawn` approach, only the Driver should therefore be started and a full MPI2 implementation Gropp et al. (1998) is required as the Driver uses the MPI2 `MPI_Comm_Spawn_Multiple` functionality. If only MPI1 implementation is available Snir et al. (1998), the Driver and the applications must be all started at once in the run script; this is the so-called `not_spawn` approach. The advantage of the `spawn` approach is that each application keeps its own internal communication context (e.g. for internal parallelisation) unchanged as in the standalone mode, whereas in the `not_spawn` approach, OASIS4 has to recreate an application communicator that must be used by the application for its own internal parallelisation. Of course, the `not_spawn` is also possible if an MPI2 library is used[2].

The Driver then participates in the definition of the different MPI communicators (see section 5.1.3), and transfers the relevant SCC information to the different component `PSMILe`coupling library (corresponding to their `prism_init` call, see section 5.1.1).

When the simulation context is set, the Driver accesses the SMIOCs XML files information (see section 6.5), which mainly defines all coupling and I/O exchanges (e.g. source or target components or files, local transformations, etc.). The Driver sorts this component specific information, and defines global identifiers for the components, their grids, their coupling/IO fields, etc. to ensure global consistency between the different processes participating in the coupling. Finally, the Driver sends to each component `PSMILe` coupling library the information relevant for its coupling or I/O exchanges (e.g. source or components target or files and their global identifier) and information about the transformations required for the different coupling fields. This corresponds to the component `PSMILe` `prism_init_comp` call (see section 5.1.2)[3]. With such information, the applications and components are able to run and perform the cou-

[1] http://www.xmlsoft.org

[2] See section 7.2.2 for related use of appropriate CPP keys.

[3] If the component is running stand-alone but linked with the `PSMILe` library for I/O actions only, there is no need to start

pling exchanges as specified by the user. The Driver/Transformer processes are then used to execute the Transformer routines (see Section 4.2).

When a component reaches the end of its execution, its processes send a signal to the Transformer master process by calling the PRISM_Terminate routine (see Section 5.8.1). Once the Transformer master process has received as many signals as processes active in the coupled run, it sends a termination message to all Transformer processes and ends.

4.2 The Transformer part

The Transformer manages the regridding (also called the interpolation) of the coupling fields, i.e. the expression on the target component model grid of a coupling field given by a source component model on its grid. The Transformer performs only the weights calculation and the regridding *per se*. As explained in section 5.5.1, the neighbourhood search, i.e. the determination for each target point of the source points that will contribute to the calculation of its regridded value, is performed in parallel in the source `PSMILe`.

The Transformer can be assimilated to an automate that reacts following predefined sequences of actions considering what is demanded. The implementation of the Transformer is based on a loop over the receptions of predefined arrays of 11 Integers sent by the component `PSMILe` . These 11 integers give a clear description of what has to be done by the Transformer. The Transformer is thus able to react with a pre-defined sequence of actions matching the corresponding sequence activated on the sender side.

The first type of action that can be requested by the component `PSMILe` is to receive the grid information resulting of the different neighbouring searches. The Transformer receives, for each intersection of source and target process calculated by the `PSMILe` , the latitude, longitude, mask, or areas of all source and target grid points in the intersection involved in the regridding (EPIOS and EPIOT, see section 5.5.1). The Transformer then calculates the weight corresponding to each source neighbour depending on the regridding method chosen by the user. The end of this phase corresponds in the component models to the `PSMILe` routine `prism_enddef`call.

During the simulation timestepping, the Transformer receives orders from the `PSMILe` linked to the different component processes to receive data for transformation (source component process) or to send transformed data (target component process). After a reception, the Transformer applies the appropriate transformations or regridding following the information collected during the initialisation phase (here, the regridding corresponds to applying the pre-calculated weights to the source field). In case of request of fields, the Transformer is able to control if the requested field has already been received and transformed. If so, the data field is sent; if not, the data field will be sent as soon as it is received and treated.

At the end of the run, the participating processes inform the Transformer once they are ready to finish the coupled simulation so that they all terminate collectively.

4.3 Interpolations and regriddings

OASIS4 offers interpolations and regriddings based on the Los Alamos National Laboratory SCRIP 1.4 library[4]. For more details on these algorithms, see SCRIP 1.4 documentation in Jones (1999) or `oasis4/doc/SCRIPusers.pdf`. These interpolations and related options are described here in more detail. All related XML elements and attributes used in the SMIOC configuration files and mentionned here are precisely defined in section 6.5.6 and in their corresponding schema in `oasis4/util/xmlfiles`.

With OASIS4, all coupling fields must be provided on a 3D grid. If a coupling field is in fact given on a 2D surface (e.g. the SST at the ocean surface) the vertical dimension of the field and the grid must have an

the Driver/Transformer; the `PSMILe` component will automatically read its SMIOC information below the `prism_init_comp` call. In this case, the component SMIOC is used to configure the I/O of the component from/to files.

[4]http://climate.lanl.gov/Software/SCRIP/

extent of 1 (see more details in section 5.3). Therefore, the interpolation of a coupling field must always be expressed either as a full 3D interpolation (see element `interp3D`) or as a combination of same 2D interpolation for all vertical levels (see element `interp2D`) followed by a 1D interpolation in the vertical (see element `interp1D`).

Currently, the 3D interpolation algorithms available are 3D nearest neighbour (element `nneighbour3D`) or trilinear (element `trilinear`). A remapping using a set of weights and addresses pre-defined by the user and stored in a file can also be chosen with element `user3D` (see 4.3.3). The 2D interpolation available are 2D nearest neighbour (element `nneighbour2D`) or bilinear (element `bilinear`) or bicubic (element `bicubic`) or 2D conservative remapping (element `conservativ2D`). For the interpolation in the vertical, a linear (element `linear`) algorithm or no interpolation at all (element `none`), which should be chosen when the extent of the grid is 1 in the vertical, are possible choices.

When the interpolation is expressed as a 2D interpolation for all vertical levels followed by a 1D interpolation in the vertical, the combinations that can be specified are:

- `nneighbour2D` and `none`
- `bilinear` and `none`
- `bicubic` and `none`
- `conservativ2D` and `none`
- `nneighbour2D` and `linear`
- `bilinear` and `linear`.

4.3.1 2D interpolations and regriddings

More details on the 2D interpolations and regriddings available and related options are provided here.

- 2D nearest neighbour (element `nneighbour2D`): an inverse-distance weighted nearest-neighbour interpolation (the great circle distance on the sphere is used):
  - The number N of source neighbours can be specified (element `nbr_neighbours`).
  - The distance can be weighted by a Gaussian (element `gaussian_variance`)
  - If some or all of the N nearest neighbours are masked, different options are available (element `if_masked`)
  - This interpolation is available for all types of 2D grid supported by OASIS4 (see section 5.3.1).
- Bilinear (element `bilinear`): an interpolation based on a local bilinear approximation :
  - If some or all of the 4 bilinear neighbours are masked, different options are available (element `if_masked`).
  - This interpolation is available for all types of 2D grid supported by OASIS4 (see section 5.3.1)
- Bicubic (element `bicubic`): an interpolation based on a local bicubic approximation :
  - Two bicubic methods are available (element `bicubic_method`): either `gradient` i.e. the 4 enclosing source neighbour values and gradient values based on the 12 additional enclosing neighbours are used (only for `PRISM_reglonlatvrt` and `PRISM_irrlonlat_regvrt` grids, see section 5.3.1) or `sixteen` i.e. the 16 enclosing source neighbour values are used (this second method assumes that the source points are located 4 by 4 at the same latitude and is therefore valid only for `PRISM_reglonlatvrt` and `PRISM_gaussreduced_regvrt` grids, see section 5.3.1).
  - If some or all of the 16 bilinear neighbours are masked, different options are available (element `if_masked`).
- 2D conservative (element `conservativ2D`): the weight of a source cell is proportional to area of the source cell intersected by target cell.
  - Currently, only the first order conservative remapping is available.

- Different types of normalization can be applied (element `methodnorm2D`)
- This remapping is available for all types of 2D grid supported by OASIS4 (see section 5.3.1).
- The following considerations must be taken into account when choosing the 2D conservative remapping:
  - Using the divergence theorem, the SCRIP library evaluates the cell intersections with the line integral along the cell borders enclosing the area. As the real shape of the borders is not known (only the location of the 4 corners of each cell is defined with the prism_set_corners call, see 5.3.2), the library assumes that the borders are linear in latitude and longitude between two corners. In general, this assumption is not really valid close to the poles. For latitudes above the `north_thresh` or below the `south_thresh` values specified in `oasis4/lib/common_oa4/include/psmile.inc`, the library evaluates the intersection between two border segments using a Lambert equivalent azimuthal projection. Problems have been observed in some cases for the grid cell located around this `north_thresh` or `south_thresh` latitude.
  - Another limitation of the SCRIP conservative remapping algorithm is that is also supposes, for line integral calculation, that $sin(latitude)$ is linear with respect to the longitude on the cell borders which again is in general not valid close to the pole.
  - For a proper consevative remapping, the corners of a cell have to coincide with the corners of its neighbour cell.
  - Duplicated cells (e.g. when a periodic grid overlaps to itself) are not allowed. In general, duplicated cells should be excluded from the valid shape (see 5.3.1); if it is not possible, dupliczted cells should then be masked.
  - A target grid cell intersecting no source cell (either masked or non masked) at all i.e. falling in a "hole" of the source grid will not be treated and will not receive any value
  - If a target grid cell intersects only masked source cells, it will be given the `psmile_dundef` value (=-280177.0).

4.3.2 3D interpolations and remappings

3D interpolations and remappings in OASIS4 are just 3D extensions of the SCRIP 2D algorithms (see section 4.3.1). These interpolations are implemented but still need to be fully validated.

- 3D nearest neighbour (element `nneighbour3D`): an inverse-distance weighted nearest-neighbour interpolation (the distance is the square root of the sum of the square radial distance and the square of the great circle distance on the sphere at the highest vertical level):
  - The number N of source neighbours can be specified (element `nbr_neighbours`).
  - The distance can be weighted by a Gaussian (element `gaussian_variance`)
  - If some or all of the N nearest neighbours are masked, different options are available (element `if_masked`)
  - This interpolation is available for all types of 3D grid supported by OASIS4 (see section 5.3.1).
- trilinear (element `trilinear`): an interpolation based on a local trilinear approximation :
  - If some or all of the 8 bilinear neighbours are masked, different options are available (element `if_masked`).
  - This interpolation is available for all types of 2D grid supported by OASIS4 (see section 5.3.1).

4.3.3 User-defined remapping

The remapping algorithms described above are based on a geographical localization of the points or cells on the target and source grids. However, some of the fields exchanged in a coupled experiment, like the water runoff of rivers or the water added to the oceans by the melting icebergs, do not fit these interpolation schemes, since these events occur at some specific place and so we would like to model them as occurring at specific places. This locality implies that the remapping should associate some specific points of the source grid with some specific points of the target grid with a user-defined weight. There is no true ”interpolation” ; instead, the computation of a value of the target function is defined by a weighted sum of a few values of the source function, taken from specific points of the source grid. The user-defined remapping is illustrated at Figure 4.1.

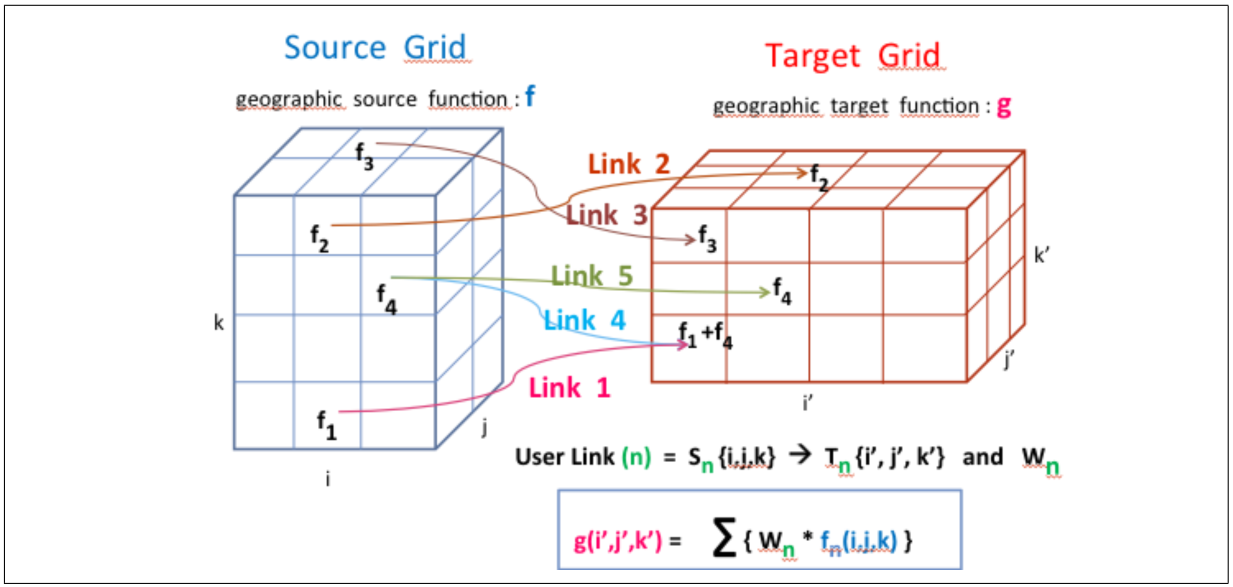
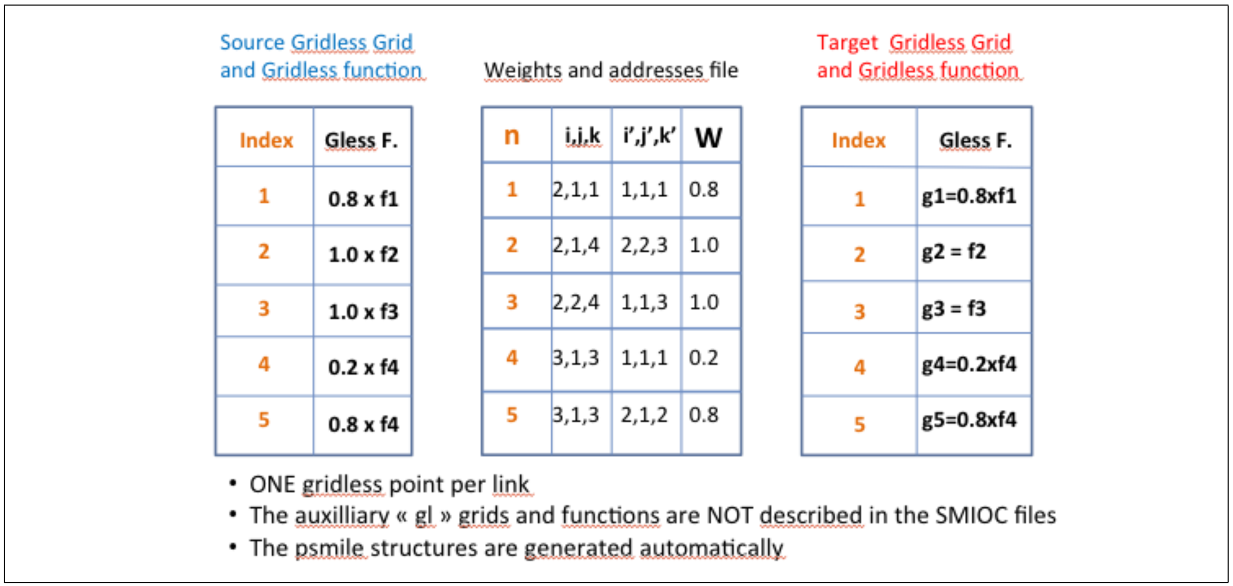

Source Grid
geographic source function : f
Target Grid
geographic target function : g
f3
f2
f4
f1
Link 2
Link 3
Link 5
Link 4
Link 1
f2
f3
f4
f1 +f4
k
i
j
k'
j'
i'
User Link (n) = Sn {i,j,k} → Tn {i', j', k'} and Wn
g(i',j',k') = ∑ { Wn * fn(i,j,k) }

Figure 4.1: User-defined remapping: association between specific points of the source grid with some specific points of the target grid

In order to achieve this, the user has to define, in a separate NetCDF file, the links associating specific points of the source grid with specific points of the target grid and the weights corresponding to each link. This is the ”user-defined weight-and-address file”. This file has to provide for each of the `nlinks` links, the index of the source point in each dimension of the source grid and the index of the target point in each dimension of the target grid. The links, source and target indices for the user-defined remapping illustrated at Figure 4.1 are detailed in Figure 4.2



Source Gridless Grid and Gridless function

Index	Gless F.
1	0.8 x f1
2	1.0 x f2
3	1.0 x f3
4	0.2 x f4
5	0.8 x f4

Weights and addresses file

n	i,j,k	i',j',k'	W
1	2,1,1	1,1,1	0.8
2	2,1,4	2,2,3	1.0
3	2,2,4	1,1,3	1.0
4	3,1,3	1,1,1	0.2
5	3,1,3	2,1,2	0.8

Target Gridless Grid and Gridless function

Index	Gless F.
1	g1=0.8xf1
2	g2 = f2
3	g3 = f3
4	g4=0.2xf4
5	g5=0.8xf4

- ONE gridless point per link
- The auxilliary « gl » grids and functions are NOT described in the SMIOC files
- The psmile structures are generated automatically

Figure 4.2: Links, source and target indices for the user-defined remapping illustrated at Figure 4.1

An example of a toy model using a user-defined remapping can be found at https://oasistrac.cerfacs.fr/browser/trunk/prism/dev_ex/user3d-auto . For this example, the content of the user-defined weight-and-address NetCDF file is:

```
netcdf weights_addresses {
dimensions:
nlinks = 10 ;
variables:
int src_ind1(nlinks) ;
src_ind1:title = "source grid  first index" ;
int src_ind2(nlinks) ;
src_ind2:title = "source grid   2nd index" ;
int src_ind3(nlinks) ;
src_ind3:title = "source grid third index" ;
int tgt_ind1(nlinks) ;
tgt_ind1:title = "target grid  first index" ;
int tgt_ind2(nlinks) ;
tgt_ind2:title = "target grid   2nd index" ;
int tgt_ind3(nlinks) ;
tgt_ind3:title = "target grid third index" ;
double weight(nlinks) ;
weight:title = "weight" ;
}
```

Chapter 5

OASIS4 Model Interface library, PSMILe

An system coupled by OASIS4 consists of different applications (each application forming one executable), each one hosting one or more than one components. To communicate with the rest of the coupled system, each component needs to perform appropriate calls to the OASIS4 Model Interface Library (`PSMILe`)[1] . The `PSMILe` is the software layer that manages the coupling data flow between any two (possibly parallel) components, directly or via additional Transformer processes, and handles data I/O from/to files.

The `PSMILe` is layered, and while it is not designed to handle the component internal communication, it completely manages the communication to other components and can also manage the details of the I/O file access. The detailed communication patterns among the possibly parallel components are established by the `PSMILe`. They are based on the source and target components identified for each coupling exchange by the user in the SMIOC XML files (see section 6.5) and on the local domain covered by each component process. This complexity is hidden from the component codes as well as the exchanges of coupling fields *per se* built on top of MPI. In order to minimise communication, the `PSMILe` also includes some local transformations on the coupling fields, like accumulation, averaging, gathering or scattering, and performs the required transformation locally before the exchange with other components of the coupled system.

The interface was designed to keep modifications of the model codes at a minimum when implementing the API. Some complexity arises however in the API from the need to transfer not only the coupling data but also the definition of the grid, mask, etc. as will be explained below. In order to match the data structures of the various component codes (in particular for the geographical information) as closely as possible, Fortran90 overloading is used. All grid description and field arrays provided by the component code through the `PSMILe` API (e.g. the grid point location through `prism_set_points` , see 5.3.6) can have one, two or three numerical dimensions and can be of type "Real" or "Double precision". There is no need to copy the data arrays prior to the `PSMILe` API call in order to match some predefined internal `PSMILe` shape. To interpret the received array correctly, a properly defined grid type is required (see section 5.3.1), since the grid type implicitly specifies the shape of the data arrays passed to the `PSMILe`. Few API routines, i.e. `prism_init`, `prism_enddef` and `prism_terminate,` are collective routines that need to be called by all processes of all applications whether or not they participate in the coupling exchanges, while the other API routine should be called only by the application processes involved in the coupling.

A major principle followed throughout the declaration phase and during the transmission of transient fields is that of using identifiers (ID) to data objects accessible in the user space once they have been declared. Like in MPI, the memory that is used for storing internal representations of various data objects is not directly accessible to the user, and the objects are accessed via their ID. Those IDs are of type INTEGER and represent an index in a table of the respective objects. The object and its associated ID are significant only on the process where it was created.

[1]The name `PSMILe` originally comes from the 'PRISM System Model Interface Library'.

The PSMILe API routines that are defined and implemented are not subject to modifications between the different versions of the OASIS4 coupler. However new routines may be added in the future to support new functionality. In addition to that the PSMILe is extendable to new types of coupling data and grids.

The next sections describe the functioning of the PSMILe, and explain its different routines in the logical order in which they should be called in a component.

5.1 Initialisation phase

The developer first has to use in his code the PRISM module ('use PRISM', see oasis4/lib/psmile_oa4/src/prism.F90), which declares all PRISM structures and PRISM integer named parameters from oasis4/lib/common_oa4/include/prism.inc (data types, grid types, error codes, etc.). The following routines then participate in the coupling initialisation phase:

5.1.1 prism_init

```
prism_init (appl_name, ierror)
```

Argument	Intent	Type	Definition
appl_name	In	character(len=*)	name of application in SCC XML file
ierror	Out	Integer	returned error code

Table 5.1: prism_init arguments

The initialisation of the OASIS4 coupling environment is performed with a call to prism_init. **This routine belongs to the class of so-called collective calls and therefore has to be called once initially by each process of each application.**

Since all communication is built on MPI routines, the initialisation of the MPI library is checked below prism_init, and a call to MPI_Init is performed if it has not been called already by the application. It is therefore not allowed to place a call to MPI_Init after the prism_init call in the application code, since this will lead to a runtime error with most MPI implementations. Conversely, a call to prism_terminate (see 5.8.1) will terminate the coupling. If MPI_Init has been called before prism_init, internal message passing within the application is still possible after the call to prism_terminate; in this case, MPI_Finalize must be called somewhere after prism_terminate in order to shut down the parallel application in a well defined way.

Within prism_init, it is detected if the coupled model has been started in the spawn or not_spawn mode (see 4.1 and 6.4). In spawn mode, all spawned processes remain in prism_init and participate in the launching of further processes until the spawning of all applications is completed.

Below prism_init call, the SCC XML information (see 6.4) is transfered from the Driver to the application process PSMILe (see 4.1).

5.1.2 prism_init_comp

```
prism_init_comp (comp_id, comp_name, ierror)
```

Argument	Intent	Type	Definition
comp_id	Out	Integer	returned component ID
comp_name	In	character(len=*)	name of component in SCC XML file
ierror	Out	Integer	returned error code

Table 5.2: prism_init_comp arguments

`prism_init_comp` needs to be called initially by each process once for each component executed by the process, no matter if different components are executed sequentially by the process or if the process is devoted to only one single component. [2]

Below the `prism_init_comp` call, the component SMIOC XML information (see 6.5) is transfered from the Driver to the component process `PSMILe` or is read directly by the `PSMILe` itself in the standalone case (see 4.1).

5.1.3 prism_get_localcomn

```
prism_get_localcomm (comp_id, local_comm, ierror)
```

Argument	Intent	Type	Definition
`comp_id`	`In`	`Integer`	component ID or `PRISM_Appl_id`
`local_comm`	`Out`	`Integer`	returned MPI communicator to be used by the component or the application for its internal communication
`ierror`	`Out`	`Integer`	returned error code

Table 5.3: prism_get_localcomm arguments

MPI communicators for the internal communication of the application and/or the components, separated from the MPI communicators used for coupling exchanges, are provided by the `PSMILe` and can be accessed via `prism_get_local_comm`.

If the argument `comp_id` is the component ID returned by routine `prism_init_comp`, then `local_comm` is a communicator gathering all component processes running the related `comp_name` component as prescribed by the user in the SCC XML file (see section 6.4) ; if instead, the predefined named integer `PRISM_appl_id` is provided, the returned `local_comm` is a communicator gathering all processes of the application.

This routine needs to be called only by MPI parallel code; it is the only MPI specific call in the `PSMILe` API.

5.1.4 prism_initialized

```
prism_initialized (flag, ierror)
```

Argument	Intent	Type	Definition
`flag`	`Out`	`Logical`	logical indicating whether `prism_init` was already called or not
`ierror`	`Out`	`Integer`	returned error code

Table 5.4: prism_initialized arguments

This routine checks if `prism_init` has been called before. If `flag` is true, `prism_init` was successfully called; if `flag` is false, `prism_init` was not called yet.

[2] If `prism_init` has not been called before by the process, `prism_init_comp` calls it and returns with a warning. Although recommended, it is therefore not necessary to implement a call to `prism_init`; in this case, as `prism_init` is a collective call, all processes of all applications need to call `prism_init_comp` .

5.2 Retrieval of SCC XML information

This section presents `PSMILe` routine that can be used in the application code to retrieve SCC XML information (see 6.4).

5.2.1 prism_get_nb_ranklists

```
prism_get_nb_ranklists (comp_name, nb_ranklists, ierror)
```

Argument	Intent	Type	Definition
`comp_name`	In	`character(len=*)`	name of the component in the SCC XML file
`nb_ranklists`	Out	`Integer`	number of rank lists for the component in the SCC file
`ierror`	Out	`Integer`	returned error code

Table 5.5: prism_get_nb_ranklists arguments

This routine needs to be called before `prism_get_ranklists` (see 5.2.2) to obtain the number of rank lists that are specified for the component in the SCC XML file (i.e. the number of elements `rank` specified for the element `component`, see 6.4).

5.2.2 prism_get_ranklists

```
prism_get_ranklists (comp_name, nb_ranklists,ranklists, ierror)
```

Argument	Intent	Type	Definition
`comp_name`	In	`character(len=*)`	name of the component in the SCC XML file
`nb_ranklists`	In	`Integer`	number of rank lists
`ranklists`	Out	`Integer`	Array `(nb_ranklists,3)` containing for the `nb_ranklists` lists of component ranks: a minimum value `(nb_ranklists,1)`, a maximum value `(nb_ranklists,2)`, an increment value `(nb_ranklists,3)`.
`ierror`	Out	`Integer`	returned error code

Table 5.6: prism_get_ranklists arguments

This routine returns the lists of ranks that are specified for the component in the SCC XML file. The ranks are the numbers of the application processes used to run the component; in the SCC XML file, the component ranks are given as lists of 3 numbers giving, in each list, a minimum value, a maximum value, and an increment value (see also section 6.4). For example, if processes numbered 0 to 7 are used to run a component, this can be describe with one rank list (0, 7, 1); if processes 0 to 2 and 5 to 7 are used, this can be described with two rank lists (0, 2, 1) and (5, 7, 1). If no maximum values is specified in the SCC file the maximum value is set to the minimum value. If no increment is specified the increment is set to 1.

Rationale: The application rank lists may be needed before the call to `prism_init_comp` in order to run the components according to the rank lists. Since a component ID is available only after the call to `prism_init_comp`, the component name is required as input argument to the `prism_get_ranklists` call instead of the component ID.

5.3 Grids and related quantities definition

In order to describe the grids onto which the coupling fields sent or received by the components are placed, the following approach was chosen.

All grids have to be described as covering a 3D domain. A 2D surface in a 3D space necessarily requires information about the location in the third dimension. For example, the grid used in an ocean model to calculate the field of sea surface temperature (SST) would be described vertically by a coordinate array of extent 1 in the vertical direction; the (only) level at which the SST field is calculated would be defined (`prism_set_points`) as well as its vertical bounds (`prism_set_corners`).

The first step is to declare a grid (see `prism_def_grid` in 5.3.1). The grid volume elements which discretise the sphere need to be defined by providing the corner points (vertices) of these volume elements (see `prism_set_corners` in 5.3.2). At this time, other properties of these volume elements can also be provided, such as the volume element mask (see `prism_set_mask` in 5.3.3).

In a second step, different sets of points on which the component calculates its variables can be placed in these volume elements. There is only one definition of volume elements per grid but there can be more than one set of points for the different variables on the same grid. The model developer describes where the points are located (see `prism_set_points` in 5.3.6). Points can represent means, extrema or other properties of the variables within the volume.

The description of the grid and related quantities is done locally for the coupling domain treated by the local process. The communication patterns used to exchange the coupling fields will usually be based on the geographical description of the local process domain. However, for fields located on a non-geographical grid, the coupling exchanges are also supported, based on the description of the local process partition in terms of indices in the global index space (see 5.3.1 and 5.3.4)[3].

5.3.1 prism_def_grid

```
prism_def_grid(grid_id, grid_name, comp_id, valid_shape, grid_type,ierror)
```

Argument	Intent	Type	Definition
`grid_id`	`Out`	`Integer`	returned grid ID
`grid_name`	`In`	`character(len=*)`	name of the grid (see below)
`comp_id`	`In`	`Integer`	component ID as provided by `prism_init_comp`
`valid_shape`	`In`	`Integer`	array(`2,ndim`) (see table 5.8) giving for each dimension the minimum and maximum index of the valid range (see below)
`grid_type`	`In`	`Integer`	PRISM integer named parameter describing the grid structure (see table 5.8)
`ierror`	`Out`	`Integer`	returned error code

Table 5.7: prism_def_grid arguments

This routine declares a grid and describes its structure.

- `grid_name`

  The argument `grid_name` must match the attribute 'local_name' of the corresponding element 'grid' in the SMIOC XML file and must be unique within the component.

- `valid_shape`

  The array `valid_shape` is dimensioned (`2,ndim`) and gives, for each of the `ndim` dimensions of the grid (see table 5.8), the minimum and maximum local index values corresponding to the "valid"

[3]Note that the IO of fields located on a non-geographical grid are not supported in the current OASIS4 version

part of the corner (see 5.3.2), point (see 5.3.6, mask (see 5.3.3) and field (see 5.4.1) arrays treated by the process, without the halo region (i.e. $iloc_{low}, iloc_{high}, jloc_{low}, jloc_{high}$ on figure 5.7). For example, if the actual extent of the first dimension is from 1 to 100, it may be that the "valid" part of the array goes from 2 to 98 (i.e. `valid_shape(1,1)=2` and `valid_shape(2,1)=98` . **Note that the "valid" part of the grid must be uniquely defined and cannot overlap to itself.**

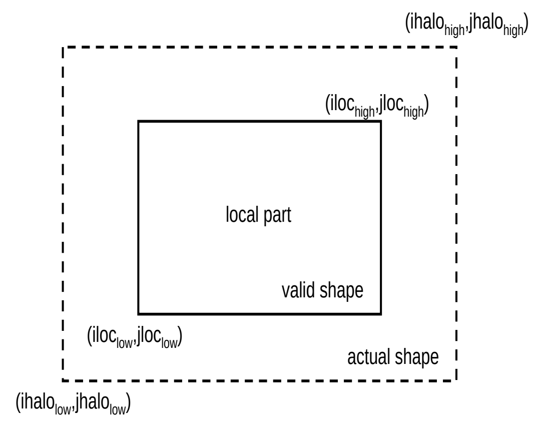


Figure 5.1: 2D example of the valid shape (expressed as `valid_shape` in `prism_def_grid`) of the corner, point, mask and field arrays transfered through the PSMILe API with shape `corner_actual_shape` or `point_actual_shape` or `mask_actual_shape` or `var_actual_shape` respectively (see below).

- `grid_type`

  The argument `grid_type` describes the grid type and implicitly specifies the shape of the corner, point, mask and field arrays passed to the `PSMILe`. Grids that are currently supported cover:

  - in the horizontal: regular, irregular, Gaussian reduced
  - in the vertical: regular
  - non-geographical grids ('gridless' grids) are also supported for repartitioning (but not for I/O in the current version).

  Table 5.8 lists the possible values of `grid_type` for the different grids supported by OASIS4 and the corresponding number of dimensions `ndim`.

  Other characteristics of the grid will be described by other routines and the link will be made by the grid identifier `grid_id`.

grid_type	ndim
`PRISM_gridless`	3, *noted (i,j,k) here*
`PRISM_reglonlatvrt`	3, *noted (i,j,k) here*
`PRISM_gaussreduced_regvrt`	2, *noted (npt_hor,k) here*
`PRISM_irrlonlat_regvrt`	3, *noted (i,j,k) here*

Table 5.8: Possible values of `grid_type` and `ndim` for the different grids supported by `PSMILe`.

Gaussian reduced grids. For Gaussian reduced grids, all processes defining the grid have to call `prism_def_grid` with `grid_type=PRISM_gaussreduced_regvrt`. Two numerical dimensions (`ndim`=2) are used to describe the 3D domain: the first dimension covers the horizontal plane and the second dimension covers the vertical. Furthermore, all these processes have to provide a description of the global reduced gaussian grid by a call to `prism_reducedgrid_map` (see 5.3.5), and have to describe the local partition of the grid with a call to `prism_def_partition` (see 5.3.4).

Non-geographical grids. For fields located on a non-geographical grid, `prism_def_grid` still has to be called with `grid_type = PRISM_gridless`. For coding reasons, `ndim` must be always equal to 3 and the call to `prism_def_grid` must be done with `valid_shape(1:2, 2:3) = 1`. The partitioning of non-geographical grids must also be described by a call to `prism_def_partition` (see 5.3.4); furthermore, a call to `prism_set_points_gridless` (see 5.3.7) is also required.

5.3.2 prism_set_corners

```
prism_set_corners (grid_id, nc, corner_actual_shape, corner_1st_array,
                   corner_2nd_array, corner_3rd_array, ierror)
```

Argument	Intent	Type	Definition
`grid_id`	`In`	`Integer`	grid ID returned by `prism_def_grid`
`nc`	`In`	`Integer`	total number of corners for each volume element
`corner_actual_shape`	`In`	`Integer`	array `(2,ndim)` giving for each `ndim` dimension of `corner_xxx_array` the minimum and maximum index of the actual range (see below)
`corner_1st_array`	`In`	`Real or Double`	corner longitude (see Table 5.10)
`corner_2nd_array`	`In`	`Real or Double`	corner latitude (see Table 5.10)
`corner_3rd_array`	`In`	`Real or Double`	corner vertical position (see Table 5.10)
`ierror`	`Out`	`Integer`	returned error code

Table 5.9: prism_set_corners arguments

For geographical grids, the volume elements which discretise the computing domain covered locally by the process are defined by giving the geographical position of the corner (vertices) of those volume elements. The exchange and repartitioning between two coupled components of a field provided on a geographical grid will be based on this geographical description of the local partition.

- `corner_actual_shape`

  The array `corner_actual_shape` is dimensioned (`2,ndim`) and gives, for each of the `ndim` dimensions (see table 5.8), the minimum and maximum local index values corresponding to the "actual" shapes of the `corner_xxx_array` arrays treated by the process including halo regions. `corner_actual_shape` is therefore greater or equal to the `valid_shape` (see section 5.3.1).

- `corner_xxx_array`

  Shape of `corner_xxx_shape`

  Table 5.10 gives the expected shape of the `corner_xxx_array` for the various `grid_type`. For `PRISM_irrlonlat_regvrt`, the corners must be given in an order such that when moving from one corner to the next one, the grid cell interior must always be to the left[4]. Furthermore, the first corner must be the lower left one in the (i,j) space, as illustrated in Figure 5.2 in the case nc$_{half} = 4$.

[4]This means that the corners must be given in a mathematical positive sense for a right-handed (i,j,k) coordinate system, but in a mathematical negative sense for a left-handed (i,j,k) coordinate system. See also http://en.wikipedia.org/wiki/Cartesian_coordinate_system.

grid_type	corner_1st_array	corner_2nd_array	corner_3rd_array
PRISM_reglonlatvrt	(i,2)	(j,2)	(k,2)
PRISM_gaussreduced_regvrt	(npt_hor,2)	(npt_hor,2)	(k,2)
PRISM_irrlonlat_regvrt	(i,j, nc_{half})	(i,j, nc_{half})	(k,2)

Table 5.10: Dimensions of corner_xxx_arrays for the various grid_type; nc is the total number of corners for each volume element; nc_{half} is nc divided by 2; i, j, k, npt_hor are the extent of the respective numerical dimensions (see table 5.8).

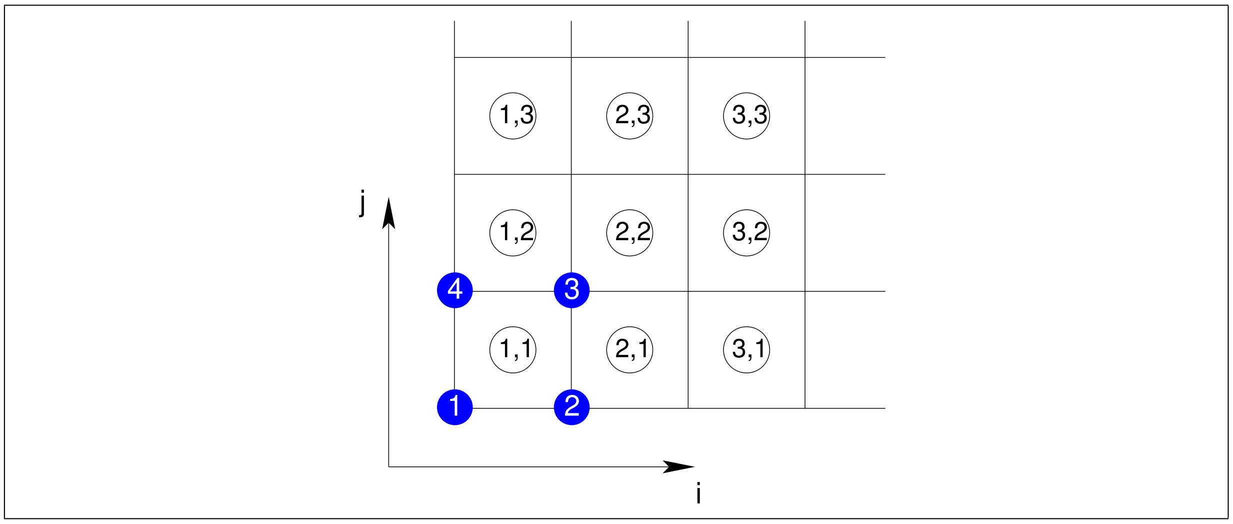


Figure 5.2: Corner ordering for PRISM_irrlonlat_regvrt grids (here with $nc_{half} = 4$)

Units of corner_xxx_array

Currently, the array corner_1st_array must be provided in degrees East in the interval -7*180 to 7*180; longitudes of the corners of one cell have to define the size of the cell (e.g. a cell with corners at 5 and 355 is a cell of 350 degrees, not a cell of 10 degrees). Currently, the array corner_2nd_array must be provided in degrees North (spherical coordinate system) in the interval -90 to 90. For corner_3rd_array, units must be the same on the source and target sides.

5.3.3 prism_set_mask

```
prism_set_mask(mask_id, grid_id, mask_actual_shape, mask_array,
               new_mask, ierror)
```

Argument	Intent	Type	Definition
mask_id	InOut	Integer	mask ID
grid_id	In	Integer	grid ID returned by prism_def_grid
mask_actual_shape	In	Integer	array (2,ndim) giving for each ndim dimension of mask_array the minimum and maximum index of actual range
mask_array	In	Logical	array of logicals; see table 5.12 for its profile; if an array element is .true. (.false.), the corresponding field point is (is not) valid.
new_mask	In	Logical	always .true. (in the current version)
ierror	Out	Integer	returned error code

Table 5.11: prism_set_mask arguments

This routine defines a mask array. Different masks can be defined for the same grid. One particular mask will be attached to a field by specifying the corresponding `mask_id` in the `prism_def_var` call used to declare the field (see section 5.4.1). The shape of `mask_array` is given in table 5.12 for the different grid types.

`grid_type`	mask_array
`PRISM_reglonlatvrt`	(i,j,k)
`PRISM_gaussreduced_regvrt`	(npt_hor,k)
`PRISM_irrlonlat_regvrt`	(i,j,k)

Table 5.12: Dimensions of `mask_array` for the various `grid_type`; `i`, `j`, `k`, `npt_hor` are the extent of the respective numerical dimensions (see table 5.8).

5.3.4 prism_def_partition

`prism_def_partition (grid_id, nbr_blocks, offset_array, extent_array,`

Argument	Intent	Type	Definition
`grid_id`	`In`	`Integer`	grid ID returned by `prism_def_grid`
`nbr_blocks`	`In`	`Integer`	number of blocks, in the global index space, covered by the `valid_shape` domain
`offset_array`	`In`	`Integer`	array(`nbr_blocks, ndim`) containing for each block the offset in each `ndim` dimension in a global index space sweeping all "valid" parts of local domains
`extent_array`	`In`	`Integer`	array(`nbr_blocks, ndim`) containing for each block the extent in each `ndim` dimension in the global index space.
`ierror`	`Out`	`Integer`	returned error code

Table 5.13: prism_def_partition arguments

The local partition treated by the process must also be described with a call to `prism_def_partition` in terms of indices in a global index space sweeping all "valid" parts of local domains, therefore based on the valid_shape defined in the routine prism_def_grid, .

The global index space is a unique and common indexing for all grid points of the component. For example, if a component covers a global domain of 200 grid points that is distributed over two processes covering 100 points each, the first and second partition **local** indices can both be (1:100); however, their **global** indices will be respectively (1:100) and (101:200).

A partition may also cover different sets of points disconnected in the global index space; each one of those sets of point constitutes one block and has to be described by its offset and extent in the global index space. Let's suppose, for example, that the 200 points in the first i direction of a component are distributed over two processes such that points with i= 1 to 50 and i = 76 to 100 are treated by the first process and such that points with i = 51 to 75 and i = 101 to 200 are treated by the second process. In this case, the number of blocks for each process is 2, and the first process blocks can be described with global offsets of 0 and 75 (`offset_array(1,1)=0`, `offset_array(2,1)=75`) and extents of 50 and 25 (`extent_array(1,1)=50`, `extent_array(2,1)=25`), while the second process blocks can be described by global offsets of 50 and 100 (`offset_array(1,1)=50`, `offset_array(2,1)=100`) and extent of 25 and 100 (`extent_array(1,1)=25`,
`extent_array(2,1)=100`). An example of `offset_array` and `extent_array` is available in the tutorial toy model is `oasis4/examples/tutorial1` (see the `readme_tutorial1.pdf` therein).

Gaussian reduced grids .

For Gaussian reduced grids, `prism_def_partition` must be called by each process to describe its local partition. The horizontal partitioning, described by `offset_array(:,1)` and `extent_array(:,1)`, must describe each latitudinal band of the reduced grid local partition as a block on its own. The `offset_array(:,1)` refer to the offset of each block in a horizontal global index space defined as the sequence of points starting at the most northern (or southern) latitude band and is going down in circular manner to the most southern (or northern) latitude band.

In this OASIS4 version, the horizontal partitioning, if any, must be the same for all vertical levels; therefore, `offset_array(:,2)` must always be equal 0 and `extent_array(:,2)` must always be equal to the number of vertical levels.

Note that in addition all processes have to call `prism_reducedgrid_map` for a description of the global reduced Gaussian grid (see 5.3.5).

Non-geographical grids ('gridless' grids).

Coupling exchanges (but not I/O in the current version) of fields not located on a geographical grid are supported, based on the description of the process local partition in terms of indices in the global index space. For these 'gridless' grids, as `ndim=3` but only the first dimension is meaningful, `extent_array(:,2:3) = 1 offset_array(:,2:3) = 0`.

2D partitions supported

Different types of 2D partitions with one or more than one block per partition are supported for the different grids.

- 2D partitions supported for `PRISM_reglonlatvrt` and `PRISM_irrlonlat_regvrt` grids

  For these type of grids, rectangular partitions with one or more blocks per partition, as illustrated in Figure 5.3 are supported. Coupling and I/O are supported for this type of partitions.

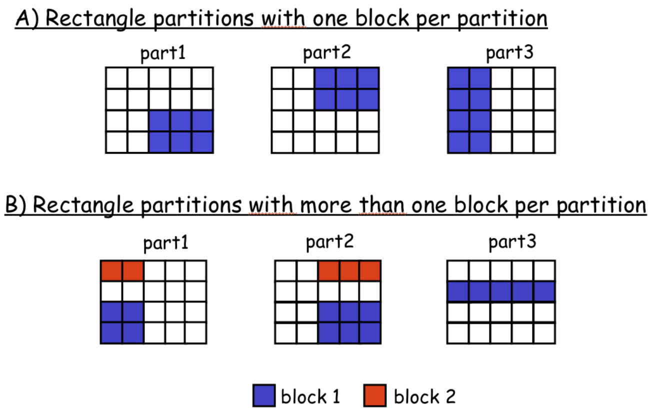

A) Rectangle partitions with one block per partition
part1
part2
part3
B) Rectangle partitions with more than one block per partition
part1
part2
part3
block 1
block 2

Figure 5.3: Partitions supported for `PRISM_reglonlatvrt` and `PRISM_irrlonlat_regvrt` grids

Some more complex partitions are also supported for these grids if one declares them as `PRISM_gaussreduced_regvrt` grids (see Figure 5.6).

- 2D partitions supported for `PRISM_gaussreduced_regvrt`

  For these grids, partitions with complete or partial latitudinal bands are supported. As explained above, each latitudinal band must be expressed as a separate block. Figure 5.4 illustrate cases where the latitudinal bands are consecutive, where as figure 5.5 illustrate a case where the latitudinal bands are not consecutive. Coupling and I/O are supported when the latitudinal bands are consecutive **but I/O are not supported -in the current OASIS4 version- when the latitudinal bands are not consecutive.**

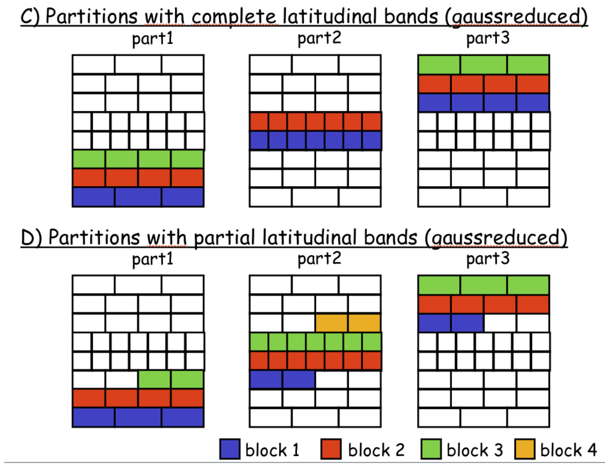

C) Partitions with complete latitudinal bands (gaussreduced)
part1
part2
part3
D) Partitions with partial latitudinal bands (gaussreduced)
part1
part2
part3
block 1
block 2
block 3
block 4

Figure 5.4: 2D partitions supported for `PRISM_gaussreduced_regvrt` (with consecutive complete or partial latitudinal bands

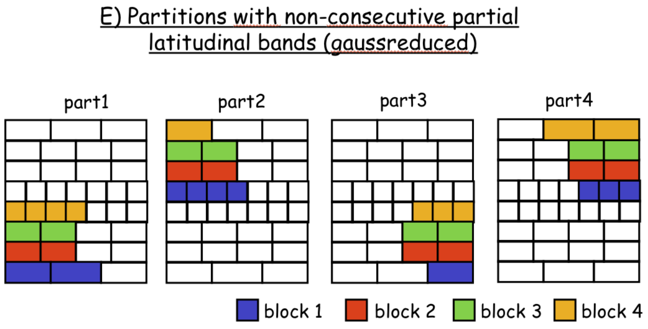

E) Partitions with non-consecutive partial latitudinal bands (gaussreduced)
part1
part2
part3
part4
block 1
block 2
block 3
block 4

Figure 5.5: 2D partitions supported for `PRISM_gaussreduced_regvrt` with non consecutive partial latitudinal bands

- Partitions supported for `PRISM_reglonlatvrt` and grids declared as `PRISM_gaussreduced_regvrt`

  The search algorithms developped for `PRISM_gaussreduced_regvrt` allows to support more general partitions, even for `PRISM_reglonlatvrt` if they are declared as `PRISM_gaussreduced_regvrt`. Figure 5.6 illustrates the case of partitions with partial latitudinal bands and with non-consecutive blocks. These cases should work but have not been thoroughly tested though. As above, I/O are not supported -in the OASIS4 current version- when the latitudinal bands are not consecutive (i.e. case G on Figure 5.6).

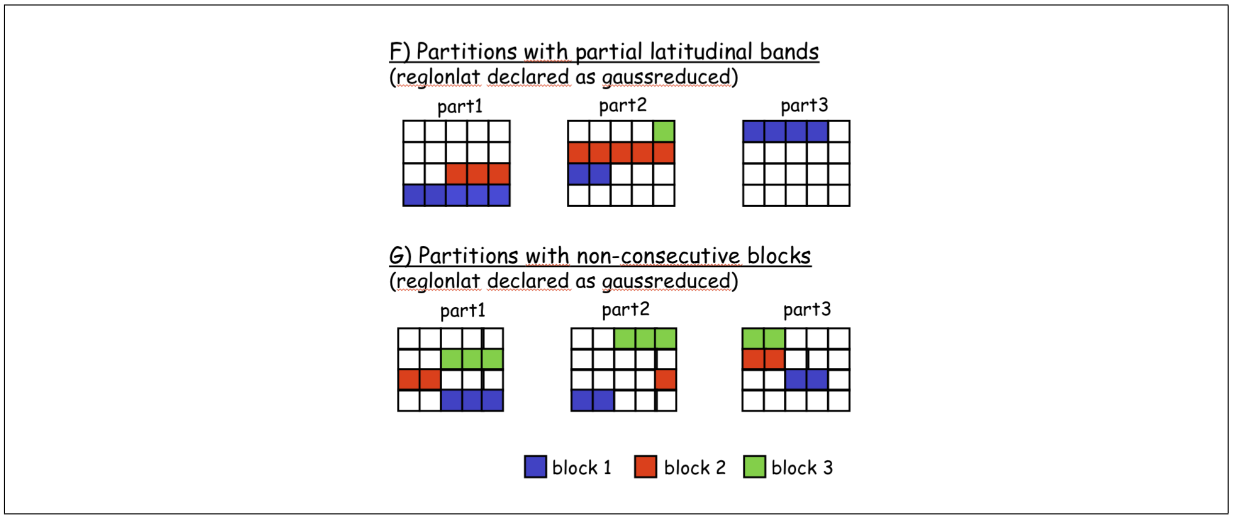

F) Partitions with partial latitudinal bands
(reglonlat declared as gaussreduced)
part1
part2
part3
G) Partitions with non-consecutive blocks
(reglonlat declared as gaussreduced)
part1
part2
part3
block 1
block 2
block 3

Figure 5.6: Partitions supported for `PRISM_reglonlatvrt` and grids declared as `PRISM_gaussreduced_regvrt`

5.3.5 prism_reducedgrid_map

`prism_reducedgrid_map (grid_id, nbr_latitudes, nbr_points_per_lat, ierror)`

Argument	Intent	Type	Definition
`grid_id`	`In`	`Integer`	grid ID returned by `prism_def_grid`
`nbr_latitudes`	`In`	`Integer`	number of latitudes of the global grid
`nbr_points_per_lat`	`In`	`Integer`	array(`nbr_latitudes`) containing for each latitude the number of grid points in longitude.
`ierror`	`Out`	`Integer`	returned error code

Table 5.14: prism_reducedgrid_map arguments

For Gaussian reduced grids only - mandatory. All processes that announce a Gaussian reduced grid have to call `prism_reducedgrid_map` for a description of the global reduced Gaussian grid, providing the same information about the global grid. As the coordinates of a Gaussian reduced grid are expressed in a 1d arrays, this additional information is needed to speed up the reconstruction of a 2d view of the 1d arrays.

Example:

```
integer, parameter          :: gnbr_lats    =     96
integer                     :: gnbr_lons(gnbr_lats)
integer, parameter          :: ndim = 48
integer                     :: array(ndim)
integer, intent(out)        :: nbr_lons(2*ndim)
integer                     :: ierror
```

```
data array / &
  20,   25,   36,   40,   45,   50,   60,   60,   72,   75,   &
  80,   90,   96,  100,  108,  120,  120,  120,  128,  135,   &
 144,  144,  160,  160,  160,  160,  160,  180,  180,  180,   &
 180,  180,  192,  192,  192,  192,  192,  192,  192,  192,   &
 192,  192,  192,  192,  192,  192,  192,  192/

do i = 1, ndim
  gnbr_lons(i) = array(i)
  gnbr_lons(2*ndim+1-i) = array(i)
enddo

call prism_reducedgrid_map ( grid_id(1), gnbr_lats, gnbr_lons, ierror )
if (ierror /= 0) n_errors = n_errors + 1
```

5.3.6 prism_set_points

```
prism_set_points ( point_id, point_name, grid_id, point_actual_shape,
                   point_1st_array, point_2nd_array, point_3rd_array,
                   new_points, ierror)
```

Argument	Intent	Type	Definition
`point_id`	`InOut`	`Integer`	ID for the set of points
`point_name`	`In`	`character(len=*)`	name of the set of points: can be anything.
`grid_id`	`In`	`Integer`	grid ID returned by `prism_def_grid`
`point_actual_shape`	`In`	`Integer`	array `(2,ndim)` giving for each `ndim` dimension of `point_xxx_array` the min and max index of actual range
`point_1st_array`	`In`	`Real or Double`	array giving the longitudes for this set of grid points
`point_2nd_array`	`In`	`Real or Double`	array giving the latitudes for this set of grid points
`point_3rd_array`	`In`	`Real or Double`	array giving the vertical positions for this set the grid points
`new_points`	`In`	`Logical`	always .true. (in the current version)
`ierror`	`Out`	`Integer`	returned error code

Table 5.15: prism_set_points arguments

With `prism_set_points` the model developer describes the geographical localization of the variables on the grid. Variables can represent means, extrema or other properties of the variables within the grid cell volume. Different sets of points can be defined for the same grid (staggered grids); each set will have a different `point_id`. A full 3D description has to be provided; for example, a set of points discretising a 2D surface must be given a vertical position. The profile of `point_xxx_array` is described in table 5.16.

Units for `point_1st_array`, `point_2nd_array` and `point_3rd_array` must be respectively the same than the ones for `corner_1st_array`, `corner_2nd_array` and `corner_3rd_array` (see

grid_type	point_1st_array	point_2nd_array	point_3rd_array
`PRISM_reglonlatvrt`	(i)	(j)	(k)
`PRISM_gaussreduced_regvrt`	(npt_hor)	(npt_hor)	(k)
`PRISM_irrlonlat_regvrt`	(i,j)	(i,j)	(k)

Table 5.16: Dimensions of `point_xxx_array` for the various `grid_type`; `i`, `j`, `k`, `npt_hor` are the extent of the respective numerical dimensions (see table 5.8).

section 5.3.2).

Non-geographical grids. For non-geographical grids ('gridless' grids), `prism_set_points_gridless` should be called instead of `prism_set_points` (see 5.3.7).

5.3.7 prism_set_points_gridless

```
prism_set_points_gridless( point_id, point_name, grid_id, new_points, ierror)
```

Argument	Intent	Type	Definition
`point_id`	`InOut`	`Integer`	set of points ID
`point_name`	`In`	`character(len=*)`	name of the set of points: can be anything.
`grid_id`	`In`	`Integer`	grid ID returned by `prism_def_grid`
`new_points`	`In`	`Logical`	always .true. (in the current version)
`ierror`	`Out`	`Integer`	returned error code

Table 5.17: prism_set_points_gridless arguments

The routine `prism_set_points_gridless` has to be called for non-geographical grids to retrieve a grid point ID.

5.4 Declaration of Coupling/IO fields

5.4.1 prism_def_var

```
prism_def_var(var_id, var_name, grid_id, point_id, mask_id, var_nodims,
              var_actual_shape, var_type, ierror)
```

Argument	Intent	Type	Definition
`var_id`	`Out`	`Integer`	returned field ID
`var_name`	`In`	`character(len=*)`	field name: must correspond to attribute `local_name` of element `transient` in the SMIOC XML file and must be unique within the component
`grid_id`	`In`	`Integer`	ID of the field grid (as returned by `prism_def_grid`)
`point_id`	`In`	`Integer`	ID of the field set of points as returned by `prism_set_points`
`mask_id`	`In`	`Integer`	ID of the field mask as returned by `prism_set_mask` or `PRISM_UNDEFINED`.
`var_nodims`	`In`	`Integer`	var_nodims(1): nb of `var_array` dimensions (see 5.6.1 and 5.6.2) that will contain the coupling/IO field (see 5.6), i.e. `ndim` except for bundles, for which it is `ndim+1` (see table 5.8); var_nodims(2): number of bundles or 0.
`var_actual_shape`	`In`	`Integer`	array`(2,ndim)` (array`(2,ndim+1)` for bundle fields) giving for each `ndim` dimension of `var_array` (see 5.6.1 and 5.6.2) the minimum and maximum index of actual range
`var_type`	`In`	`Integer`	data type of the field: PRISM integer named parameter `PRISM_Integer`, `PRISM_Real` or `PRISM_Double_Precision`
`ierror`	`Out`	`Integer`	returned error code

Table 5.18: prism_def_var arguments

After the initialisation and grid definition phases, each field that will be send/received to/from another component (coupling field) or that will be written/read to/from a disk file (IO field) through `PSMILe`'put' / 'send' actions needs to be declared and associated with a previously defined grid and mask.

The units of a coupling/IO field should be indicated in the PMIOD XML file. By consulting the appropriate PMIOD, the user is able to check if the units of a coupling field match on the source and target side and if not, he has to choose appropriate transformations in the SMIOC XML files.

For the case where a set of fields ordered along an extra dimension, sharing the same units, and located on the same set of points (e.g. chemical species), need to be treated together, the 'bundle' notion has been introduced. Such bundle field should be declared as one coupling/IO field and the number of bundles should be indicate in `var_nodims(2)`; only one `var_id` will be returned. This implies that the complete bundle will have to be transfered (send) to the remote component at once, and that the remote component must be able to treat these bundles; both components have to agree on the precise sequence of the physical fields contained in this fields.

5.5 Neighbourhood search and determination of communication patterns

5.5.1 prism_enddef

`prism_enddef (ierror)`

Argument	Intent	Type	Definition
`ierror`	`Out`	`Integer`	returned error code

Table 5.19: prism_enddef arguments

Following `prism_init`, **`prism_enddef` is the second collective call and has to be called once by each application process** when all components within the application have completed their definition phase. (The rest of this section can be skipped by users not interested in the PSMILe internal functioning.)

To perform the exchange of coupling fields during the run, it is required to establish communication only between those pairs of processes that actually have to exchange data based on the user defined coupling configuration in the SMIOCs XML files (see section 6.5).

For each coupling exchange involving a regridding between the source and the target grids, the neighbourhood search is performed. It identifies, for each grid point of each target process, the source grid points and corresponding source process that will be used to calculate the target grid point value. For a coupling exchange involving only repartitioning, each target grid point corresponds exactly to only one source grid point; in this case the 'neighbourhood search' process identifies, for each grid point of each target process, on which source process the matching source grid point is located.

In order to save memory and CPU time in the neighbourhood search and the establishment of the communication patterns, `prism_enddef` works in a parallel way on the local grid domain covered by each application process as much as possible. The search algorithm is split into three parts.

In an initial step, each process calculates a bounding box covering its local geographical volume domain previously defined by `prism_set_corners` (see section 5.3.2). The bouding boxes of all processes are sent to and collected by all processes. Each source process calculates the intersection of its bounding box with each other process bounding box, thereby identifying the potential interpolation partners and corresponding bounding box intersection. (For fields located on non-geographical fields, see 5.3.1, the intersection calculation is based on the local domain description in the global index space, see 5.3.4.) For each bounding box intersection, the source process creates a sequence of simplified grids and corresponding bounding boxes, each one coarsened by a factor of 2 with respect to the previous one, until falling back onto the bounding box covering the whole intersection (similar to a Multigrid Algorithm). Starting on the coarsest level the search algorithm determines, at each multigrid level, the source bounding box for each target grid point in the intersection. When the bounding box at the finer level is identified, the neighbours of the target grid point, i.e. the source points participating in its calculation (regridding case) or the matching source grid point (repartitioning only case), are identified. The source locations that are identified at this stage can be considered as a frist guess. The source locations are located close to the final source location.

In a second step, the exact source locations are determined and provide the initial location to identify further required neighbour points in a third step which is depending on the interpolation scheme chosen by the user.

For each intersection of source and target grid processes, the 'Ensemble of grid Points participating in the Interpolation Operation (EPIO)' (or in the repartitioning) on the source side (EPIOS) and on the target side (EPIOT) are identified. The results of this search are transfered to the target process. For the coupling exchange involving regridding, the EPIOS and EPIOT definition and all related grid information are also transferred to the Transformer (see section 4.2).

As the results of the neighbourhood search are known in the source `PSMILe`, only the usefull grid points will be effectively sent later on during the coupling exchanges, minimising the amount of data to be transferred.

5.6 Exchange of coupling and I/O fields

The `PSMILe` exchanges are based on the principle of "end-point" data exchange. When producing data, no assumptions are made in the source component code concerning which other component will consume these data or whether they will be written to a file, and at which frequency. Likewise, when asking for data, a target component does not know which other component produces them or whether they are read in from a file. The target or the source (another component or a file) for each field is defined by the user in the SMIOC XML file (see section 6.5) and the coupling exchanges and/or the I/O actions take place according to the user external specifications. The switch between the coupled mode and the forced mode is therefore totally transparent for the component. Furthermore, source data can be directed to more than one target (other components and/or disk files).

The sending and receiving `PSMILe` calls `prism_put` and `prism_get` can be placed anywhere in the source and target code and possibly at different locations for the different coupling fields. These routines can be called by the model at each timestep. The actual date at which the call is performed and the date bounds for which it is valid are given as arguments; the sending/receiving is actually performed only if the date bounds include a time at which it should be activated, given the field coupling (or I/O) period indicated by the user in the SMIOC; a change in the coupling or I/O period is therefore also totally transparent for the component itself. The `PSMILe` can also take into account a timelag between the sending `prism_put` and the corresponding `prism_get` defined by the user in the SMIOC (see item 6. of section 6.5.4).

Local transformations can be performed in the source component `PSMILe` below the `prism_put` and/or in the target component `PSMILe` below the `prism_get` like time accumulation, time averaging, algebraic operations, statistics, scattering, gathering (see item 7. of section 6.5.4 and item 5. of section 6.5.5).

When the action is activated, each process sends or receives only its local partition of the data, corresponding to its local grid defined previously. The coupling exchange, including data repartitioning if needed, occurs either directly between the components, or via additional Transformer processes if regridding needed (see section 4.2).

If the user specifies that the source of a `prism_get` or the target of a `prism_put` is a disk file, the `PSMILe` exploits the GFDL mpp_io package [Balaji (2001)] for its file I/O. The supported file format is NetCDF according to the CF convention [Eaton et al. (2003)]. The mpp_io package is driven by a `PSMILe` internal layer which interfaces with various sources of information. For instance, the definition of grids and masks as well as the form of the data of a field is provided through the `PSMILe` API. On the other hand the information with regard to the CF standard name and unit are provided by the SMIOC XML file through the Driver.

The mpp_io package can operate in three general I/O modes:

- **Distributed I/O**

  Each process works on a individual file containing the I/O field on the domain onto which that process works. Domain partitioning information is written into the resulting files such they can be merged into one file during a post processing step.

- **Pseudo parallel I/O**

  The whole field is read from or written to one file. The domain partitioning information is exploited such that the data are collected - stitched together - during the write operation or distributed to the parallel processes of a component during the read operation. This domain stitching or distribution is automatically done by the `PSMILe` on the component model master process and happens transparently for the parallel component itself.

- **Parallel I/O**

  A fully parallel I/O using the parallel NetCDF [Li et al. (2003)] library and MPI-IO is available. This allows parallel IO of distributed data into a single NetCDF file which is controlled by MPI-IO instead of collecting the data on the master process first. To have this feature available the PSMILe has to be linked against the parallel NetCDF library. The PSMILe library has to be generated with

-D__PARNETCDF. Note that this type of IO is not yet supported for applications having more than 1 component.

The PSMILe I/O layer also copes with the fact that the input data may be spread accross a number of different files[5], and that NetCDF file format has certain restrictions with respect to size of a file. Therefore, on output chunking of a series of time stamps across multiple files will be provided depending on a threshold value of the file size.

5.6.1 prism_put

`prism_put (var_id, date, date_bounds, var_array, info, ierror)`

Argument	Intent	Type	Definition
var_id	In	Integer	field ID returned from prism_def_var
date	In	Type(PRISM_Time_Struct)	date at which the prism_put is performed
date_bounds	In	Type(PRISM_Time_Struct)	array(2) giving the date bounds between which this call is valid
var_array	In	Integer, Real or Double	field array to be sent (see table 5.21 for its profile, adding one dimension for bundle fields)
info	Out	Integer	returned info about action performed (see below)
ierror	Out	Integer	returned error code

Table 5.20: prism_put arguments

This routine is called to send var_array content to a target component or file. The target is defined by the user in the SMIOC XML files (see section 6.5). This routine will return even if the corresponding prism_get has not been performed on the target side, both for an exchange through the Transformer and for a direct exchange (for direct exchange the content of the var_array is buffered in the PSMILe and for an exchange through the Transformer the data are buffered in the Transformer). The shape of var_array is given in table 5.21 for the different grid types.

grid_type	var_array
PRISM_reglonlatvrt	(i,j,k)
PRISM_gaussreduced_regvrt	(npt_hor,k)
PRISM_irrlonlat_regvrt	(i,j,k)

Table 5.21: Dimensions of var_array for the various grid_type; i, j, k, npt_hor are the extent of the respective numerical dimensions (see table 5.8).

This routine can be called in the component code at each timestep but the sending is actually performed only if the time *period* covered by the date bounds (with $period =]date\_bounds(1), date\_bounds(2)]$ i.e. with the lower and upper date bounds being respectively excluded and included) covers a valid coupling or I/O date, given the field coupling or I/O period indicated by the user in the SMIOC XML files. The date and date_bounds arguments must be given as PRISM_Time_Struct structures (see

[5]The system calls 'scandir' and 'alphasort' are used to implement this feature (see routine /oasis4/lib/psmile_oa4/src/psmile_io_scandir.c). In case of problems with these system calls, one may try to compile with the -D__MYALPHASORT. If there are still problems, one has to comment the calls to psmile_io_scandir_no_of_files and psmile_io_scandir in psmile_open_file_byid.F90, but then that PSMILe functionality will not be provided anymore.

/oasis4/lib /common_oa4/src/prism_constants.F90). The sum of the time periods covered by the date_bounds of the different prism_put calls of a run must cover exactly the whole run duration without any gap and any overlap. The meaning of the different info returned are as follows:

- PRISM_NoAction = 0: no action is performed for this call
- PRISM_Cpl = 1000: the array is only sent to another component
- PRISM_CplIO = 1100: the array is sent to another component and written to a file
- PRISM_CplRst = 1010: the array is sent to another component and written to a coupling restart file (see 5.7).
- PRISM_CplTimeop = 1001: the array is sent to another component and used in a time operation (accumulation, averaging)
- PRISM_CplIORst = 1110: the array is sent to another component, written to a file, and written to a coupling restart file (see 5.7)
- PRISM_CplIOTimeop = 1101: the array is sent to another component, written to a file, and used in a time operation
- PRISM_CplRstTimeop = 1011: the array is sent to another component, written to a coupling restart file (see 5.7), and used in a time operation
- PRISM_CplIORstTimeop = 1111: the array is sent to another component, written to a file, written to a coupling restart file (see 5.7), and used in a time operation
- PRISM_IO = 100: the array is only written to a file
- PRISM_IORst = 110: the array is written to a file and to a coupling restart file (see 5.7)
- PRISM_IOTimeop = 101: the array is written to a file and used in a time operation
- PRISM_IORstTimeop = 111: the array is written to a file and to a coupling restart file (see 5.7) and is used in a time operation
- PRISM_Rst = 10: the array is only written to a coupling restart file
- PRISM_RstTimeop = 11: the array is written to a coupling restart file (see 5.7) and used in a time operation
- PRISM_Timeop = 1: the array is used in a time operation

The meaning of the different ierror returned can be accessed using the routine prism_error (see section 5.9.3).

5.6.2 prism_get

```
prism_get(var_id, date, date_bounds, var_array, info, ierror)
```

Argument	Intent	Type	Definition
`var_id`	`In`	`Integer`	field ID returned by `prism_def_var`
`date`	`In`	`Type(PRISM_Time_Struct)`	date at which the `prism_get` is performed
`date_bounds`	`In`	`Type(PRISM_Time_Struct)`	array(2) giving the date bounds between which this call is valid
`var_array`	`InOut`	`Integer, Real or Double`	field array to be received (see Table 5.8 for its profile, adding one dimension for bundle fields)
`info`	`Out`	`Integer`	returned info about action performed (see below)
`ierror`	`Out`	`Integer`	returned error code

Table 5.22: prism_get arguments

This routine is called to receive a field `var_array` from a source component or file. The source is defined by the user in the SMIOC XML files (see section 6.5). This routine returns only when the corresponding `prism_put` is performed on the source side and when data is available in `var_array`, after regridding if needed. The shape of `var_array` is given in table 5.21 for the different grid types.

As for `prism_put` , this routine can be called in the component code at each timestep but the receiving is actually performed only if the time *period* covered by the date bounds (with $period =]date\_bounds(1), date\_bounds(2)]$ i.e. the lower and upper date bounds being respectively excluded and included) covers a valid coupling or I/O date, given the field coupling or I/O period indicated by the user in the SMIOC XML files. The sum of the time periods covered by the `date_bounds` of the different `prism_get` calls of a run must cover exactly the whole run duration without any gap and any overlap.

Note that `var_array` is of intent `InOut` and is updated only for the part for which data have been effectively received. We therefore recommend to initialise `var_array` with a very recognizable positive value (i.e. 999999.) before the `prism_get` to be able to clearly identify the data received; this value should be positive so to clearly identify the target grid points which take the `PSMILe_dundef` value (=-280177.) after interpolation (see details in section 6.5.6.

The meaning of the different `info` returned is as follows:

- PRISM_NoAction = 0: no action is performed for this call
- PRISM_Cpl = 1000: the array is only received from another component
- PRISM_IO = 100: the array is read from a file
- PRISM_IOTimeop = 101: the array is read from a file and used in a time operation

The meaning of the different `ierror` returned can be accessed using the routine `prism_error` (see section 5.9.3).

5.6.3 prism_put_inquire

```
prism_put_inquire (var_id, date, date_bounds, info, ierror)
```

Argument	Intent	Type	Definition
`var_id`	`In`	`Integer`	field ID returned from `prism_def_var`
`date`	`In`	`Type(PRISM_Time_Struct)`	date at which the `prism_put` would be performed
`date_bounds`	`In`	`Type(PRISM_Time_Struct)`	array(2) giving the date bounds between which the field would be valid
`info`	`Out`	`Integer`	returned info about action that would be performed (see below)
`ierror`	`Out`	`Integer`	returned error code

Table 5.23: prism_put_inquire arguments

This function is called to inquire if the corresponding `prism_put` (i.e. for same `var_id`, `date`, and `date_bounds`) would effectively be activated. This can be useful if the calculation of the related `var_array` is CPU consuming.

The meaning of the different `info` returned is the same as for the `prism_put` routine (see 5.6.1).

The meaning of the different `ierror` returned can be accessed using the routine `prism_error` (see section 5.9.3).

5.6.4 prism_put_restart

`prism_put_restart (var_id, date, date_bounds, data_array, info, ierror)`

Argument	Intent	Type	Definition
`var_id`	`In`	`Integer`	field ID from `prism_def_var`
`date`	`In`	`Type(PRISM_Time_Struct)`	date at which the `prism_put_restart` is performed
`date_bounds`	`In`	`Type(PRISM_Time_Struct)`	array dimensioned (2) giving the date bounds between which this data is valid
`data_array`	`In`	`Integer, Real or Double`	data array to be transferred (see table 5.8 for its profile, adding one dimension for bundle fields),
`info`	`Out`	`Integer`	returned info about action performed
`ierror`	`Out`	`Integer`	returned error code

Table 5.24: prism_put_restart arguments

This function forces the writing of a field into a coupling restart file and can be used in the same fashion as `prism_put` , except that the `prism_put_restart` does not consider any coupling period from the SMIOC file and writes the data array into the restart file each time it is called. Note that a `prism_put_restart` can follow a `prism_put` .

This function should be used when a coupling restart file of a coupling field is needed (see section 5.7 for details on the restart mechanism) but not available. In that case, the user should run the source component beforehand to create the first coupling restart file of an experiment explicitly with a call to `prism_put_restart` . The returned `info` should be PRISM_Rst = 10 (the array is only written to a coupling restart file); in case the returned `info` is PRISM_Noaction = 0, an internal error occured. The meaning of the different `ierror` returned can be accessed using the routine `prism_error` (see section 5.9.3).

To use `prism_put_restart` , one should pay attention to the following details:

- There must be a lag equal to 0 defined for the corresponding field in the component SMIOC XML file (see 6.5.4) .
- Since the `prism_enddef` performs some IO related initialisation, a `prism_put_restart` cannot be invoked before the `prism_enddef` is completed.
- The name of the restart file will be <field_local_name>_<component_local_name> _<application_local_name>_rst.<date>.nc, where <date> is the current run `end_date` indicated in the SCC XML file.
- The time information written into the restart file (variable time(time)) corresponds to the calling argument `date_bounds(2)`. At restart, the time information in the restart file must be the run `start_date` indicated in the SCC XML file $\pm$ 2 seconds.
- Currently, fields written to a restart file via `prism_put_restart` are currently taken as is and are not processed with respect to local operations like gathering/scattering averaging, summation or any reduction operations.

A concrete example on how to use the PSMILe `prism_put_restart` routine to create an OASIS4 coupling restart file can be found in directory `oasis4/examples/toyoa4_restart` (see the README therein). In practice, it is recommended, as done in `toyoa4_restart` to follow these rules:

- In the SCC XML file, put the `experiment start_date` = `experiment end_date` = `run start_date` = `run end_date` = the start date of the run for which the restart file will be created.

- As date_bounds arguments for the prism_put_restart put date_bounds(2) = the start date of the run for which the restart file will be created and date_bounds(1) = some time (e.g. one hour) before that same start date.

5.7 The Restart Mechanism

With OASIS4, a coupling restart file is required for the prism_getperformed at the beginning the run when the user wants the time stamp of the prism_put to lag behing the time stamp of the corresponding prism_get. In OASIS4, this is realised by specifying a lag for the corresponding output field in the source component SMIOC XML file (see 6.5.4). This lag specifies the number of prism_put periods to be added to the prism_put date and date_bounds to match the corresponding prism_get date and date_bounds in the target component. A prism_put period is defined as the time between the upper and lower date_bounds of a prism_put; i.e. for a lag of 1, the time added to the prism_putdate and date_bounds arguments will be once the time difference between the associated date_bounds.

If a lag is specified in the SMIOC file, two restart files are opened by the source PSMILe library under the prism_enddef , one for reading at the beginning of the run and one for writing at the end of the run. For the first run of an experiment, a restart file has to be created externally by the user and provided at run time (see prism_put_restart 5.6.4) . In the same fashion, the new restart that is written at run time at the end of the run has to be provided as the input to the subsequent run.

The name of the netCDF restart file must be
`<field name>_<component name>_<application name>_rst.<date>.nc`, where `<date>` is the run start_date.

Inside the prism_enddef (and therefore hidden from the user), the respective source PSMILe processes automatically read the local partitions of their output coupling fields defined with a lag from the coupling restart files. The field in the restart file must have a shape equal to the valid shape of the global grid (i.e. the sum of the local valid_shape over all source processes - see 5.3.1). Each process then automatically sends its local restart information to the Transformer which performs the interpolation and sends the interpolated fields to the target component (if no interpolation is required, the data is sent directly to the target component).

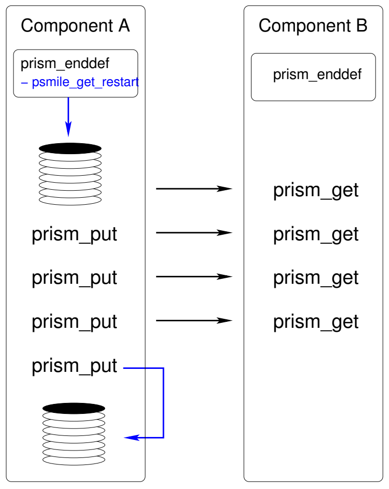


Figure 5.7: Schematic calling sequence for handling of restart files.

On the receiving side, the prism_get of a coupling field defined with a lag in the source SMIOC XML file, which date_bounds include the run start_date, will receive the data coming from the restart file. The time information in the restart file (variable time(time)) must be the run start_date indicated in the SCC XML file with a tolerance of ± 2 seconds in the current implementation. For the subsequent echanges, thanks to the `lag`, the `prism_put` called by the source component with arguments `(source) date_bounds` will match the `prism_get` called in the target component at at date = `(source) date_bounds + lag`. At the end of the run, a restart file will automatically be written below a `prism_put` call which `date_bounds(2) + lag` exceeds the `run end_date`[6]. The time information written to the restart file will correspond to the `end_date` of the run. This restart file has to be made available for the next run.

5.7.1 An example of a restart generation

An example of how to generate a restart file for the OASIS4 toyoa4 toy coupled model is available in directory `oasis4/examples/toyoa4_restart`. In the toyoa4 toy coupled model, a field with name COSENHFL is send from the atmosphere component to the land component; if a lag is specified for this field in the atmosphere SMIOC file, a restart file must be provided for this field at the beginning of the run. The sources from the `toyoa4_restart` example can serve as a template for the user to create her own restart files by extending the Fortran source (atmoa4_rsst.F90) and the XML file (atmoa4_atmos_smioc.xml).

One can note that although grid information is written into the restart file, this information is not interpreted by the OASIS4 PSMILe at runtime when the restart is read in. By having a closer look at atmoa4_rst.F90, one will recognise that only a dummy grid is written out, but that the size of the field written to the file exactly cover the valid shape of the global grid. The `date` in the restart file name and the time information written into the restart file must match with the `start_date` of toyoa4 first run. To achieve this, the run `start_date` and `end_date` and the upper `date_bounds` (`date_bounds(2)`) of the `prism_put_restart` are set to the toyoa4 initial run `start_date` which is 1 Jan 2000, 0:0:00.0.

If the same toy example is run with lag set to zero no restart file is required, and the first `prism_get` in the land model will be served by the first `prism_put` of the atmosphere.

5.8 Termination Phase

5.8.1 prism_terminate

`prism_terminate (ierror)`

Argument	Intent	Type	Definition
`ierror`	`Out`	`Integer`	returned error code

Table 5.25: prism_terminate arguments

In analogy to the initialisation phase, a call to **`prism_terminate`, which again is a collective call and therefore needs to be called by all processes of all applications**, will make the calling process to wait for other processes participating in the coupling to reach the `prism_terminate` as well. At this point, the following actions are performed:

- All open units under control of the `PSMILe` are closed.
- The output to standard out is flushed.
- The Driver is notified about the termination of the respective process.
- All memory under control of `PSMILe` is deallocated.

After calling `prism_terminate` , no coupling exchanges are possible for this process and no further I/O actions under control of the `PSMILe` can be performed; however, it is still possible for the application to perform local operations and to write additional output which shall not be under control of the `PSMILe`.

[6]When the duration of the run does not equal a finite number of coupling period, the field is written to the restart file if `date_bounds(2)+lag > last_date` where `last_date` is the first coupling time $\geq$ `run end_date`.

If `MPI_Init` has been called in the code before the call to `prism_init`, component internal MPI communication is still possible after the call to `prism_terminate`, until the `MPI_Finalize` is called by the component (see also section 5.1.1). Otherwise `prism_terminate` will call `MPI_Finalize`itself.

5.8.2 prism_terminated

`prism_terminated (flag, ierror)`

Argument	Intent	Type	Definition
`flag`	`Out`	`Logical`	if .true., prism_terminate was already called
`ierror`	`Out`	`Integer`	returned error code

Table 5.26: prism_terminated arguments

This routine can be used to check whether `prism_terminate` has already been called by this process. This may help to detect ambiguous implementations of multi-component applications.

5.8.3 prism_abort

`prism_abort (comp_id, routine, message )`

Argument	Intent	Type	Definition
`comp_id`	`In`	`Integer`	component ID as provided by `prism_init_comp`
`routine`	`In`	`Character`	calling routine name
`message`	`In`	`Character`	user defined message

Table 5.27: prism_abort arguments

It is common practice in non parallel Fortran codes to terminate the program by calling a Fortran `STOP` in case a runtime error is detected. In MPI-parallelised codes it is strongly recommended to call `MPI_Abort` instead to ensure that all parallel processes are stopped and thus to avoid non-defined termination of the parallel program. For coupled application, the `PSMILe` provides a `prism_abort` call which guarantees a clean and well-defined shut down of the coupled model. We recommend to use `prism_abort` instead of a Fortran `STOP` or a `MPI_Abort`.

5.9 Query and Info Routines

5.9.1 prism_get_calendar_type

`prism_get_calendar_type (calendar_name, calendar_type_id, ierror)`

Argument	Intent	Type	Definition
`calendar_name`	`Out`	`Character(len=132)`	name of calendar used
`calendar_type_id`	`Out`	`Integer`	ID of calendar used
`ierror`	`Out`	`Integer`	returned error code

Table 5.28: prism_get_calendar_type arguments

This routine returns the name and the ID of the calendar used in the `PSMILe`. Currently, the only calendar supported is the 'Proleptic Gregorian Calendar' (i.e. a Gregorian calendar[7] extended to dates before 15 Oct 1582) and its ID is 1 (i.e. the PRISM integer name parameter `PRISM_Cal_Gregorian = 1`, see

[7]The Gregorian calendar considers a leap year every year which is multiple of 4 but not multiple of 100, and every year which is a multiple of 400.

oasis4/lib/common_oa4/include/prism.inc). Simple calendars with 360 and 365 days are implemented but not directly available to the user. In a future version, the calendar type should be chosen and specified by the user in an XML configuration file, read in from this XML file by the Driver, and transfered to the PSMILe.

5.9.2 prism_calc_newdate

prism_calc_newdate (date, date_incr, ierror)

Argument	Intent	Type	Definition
date	InOut	Type(PRISM_Time_Struct)	In and Out date
date_incr	In	Integer, Real or Double	Increment in seconds to add to the date
ierror	Out	Integer	returned error code

Table 5.29: prism_calc_newdate arguments

This routine adds a time increment of date_incr seconds to the date given as In argument and returns the result in the date as Out argument. The time increment may be negative. The time structure PRISM_Time_Struct object is defined as a Fortran type of the form

```
   Type PRISM_Time_Struct
      Double Precision   :: second
      integer            :: minute
      integer            :: hour
      integer            :: day
      integer            :: month
      integer            :: year
   End Type PRISM_Time_Struct
```

5.9.3 prism_error

prism_error (ierror, error_message)

Argument	Intent	Type	Definition
ierror	In	Integer	an error code returned by a PSMILe routine
error_message	Out	character(len=*)	corresponding error string

Table 5.30: prism_error arguments

This routine returns the string of the error message error_message corresponding to the error code ierror returned by other PSMILe routines. In general, 0 is returned as error code if the routine completed without error; a positive error code means a severe problem was encountered.

5.9.4 prism_version

prism_version()

This routine prints a message giving the SVN revision of the PSMILe library currently used.

5.9.5 prism_get_real_kind_type

prism_get_real_kind_type (kindr, type, ierror)

Argument	Intent	Type	Definition
`kindr`	`In`	`Integer`	kind type parameter of REAL variables
`type`	`Out`	`Integer`	PRISM datatype corresponding to `kindr`
`ierror`	`Out`	`Integer`	returned error code

Table 5.31: prism_get_real_kind_type arguments

This routine returns in `type` the PRISM datatype which corresponds to the kind type parameter `kindr`. `type` can be either `PRISM_Real = 4`, or `PRISM_Double_Precision = 5` (see `oasis4/lib/common_oa4/include/prism.inc`).

5.9.6 prism_remove_mask

```
prism_remove_mask ( mask_id, ierror )
```

Argument	Intent	Type	Definition
`mask_id`	`In`	`Integer`	mask ID as returned by `prism_set_mask`
`ierror`	`Out`	`Integer`	returned error code

Table 5.32: prism_remove_mask arguments

The routine removes the mask information linked the mask ID `mask_id` given as argument.

Chapter 6

OASIS4 description and configuration XML files

This chapter details the content of the XML description and configuration files used with OASIS4.

- The XML description files are used to:
  - describe each application: the "Application Description" (AD)
  - describe the relations a component model of an application is able to establish with the external environment through inputs and outputs: the "Potential Model Input and Output Description" (PMIOD)

  The description XML files, i.e. the ADs and PMIODs, should be created by the component model developer, either by hand or with the graphical user interface `wizard.tcl` available in `oasis4/util/gui`, to provide information about the general characteristics and the potential coupling interface of its code, but they are not used by the OASIS4 coupler.

- The XML configuration files are used to:
  - configure the general characteristics of a coupled model run: the "Specific Coupling Configuration" (SCC)
  - configure the relations the component model will establish with the external environment through inputs and outputs for a specific run: the Specific Model Input and Output Configuration (SMIOC).

  The configuration XML files, i.e. the SCC and the SMIOCs, must be created by the coupled model user, i.e. the person that builds the coupled model, either by hand or with the graphical user interface `oasis-gui.tcl` available in `oasis4/util/gui`. They provide specifications about the process management and the coupling and I/O exchanges of one particular coupled model and are used by the OASIS4 coupler.

6.1 Introduction to XML concepts

Extensible Markup Language (XML) is a simple, very flexible text format. Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere. An XML document is simply a file which follows the XML format.

The purpose of a DTD or a Schema is to define the legal building blocks of an XML document. The AD, SCC, PMIOD and SMIOC XML documents must follows the Schemas files `ad.xsd`, `scc.xsd`, `pmiod.xsd` and `smioc.xsd` respectively, available in the directory `oasis4/util/ xmlfiles`.

The `xmllint` command with the following options can be used to validate an XML file `file.xml` against a Schema file `file.xsd`:

```
xmllint --noout --valid --postvalid --schema file.xsd file.xml
```

The building blocks of XML documents are Elements, Tags, and Attributes.

- Elements

  Elements are the main building blocks of XML documents.

  Examples of XML elements in `pmiod.xsd` are `component` or `code`. Elements can contain text, other elements, or be empty.

  The values of `minOccurs` and `maxOccurs` for an element in the Schema file indicate how many times this element must occur in the corresponding XML file; if `minOccurs` and `maxOccurs` are not specified, the element must appear once.

- Tags

  Tags are used to markup elements.

  In the XML file, a starting tag like <element_name> marks up the beginning of an element, and an ending tag like </element_name> marks up the end of an element.

  Example: <laboratory>Meteo-France</laboratory>

  An empty element will appear as <element_name />.

- Attributes

  Attributes provide extra information about elements and are placed inside the start tag of an element. As indicated in the Schema file, an attribute may be "required" (use='required') or "optional" (use='optional').

  Example: <grid local_name="AT31_2D">

  The name of the element is "grid". The name of the attribute is "local_name". The value of the attribute is "AT31_2D".

6.2 The Application Description (AD)

The Application Description (AD) describes the general characteristics of one application. There is one AD per application, i.e. per code which when compiled forms one executable. An application may contain one or more component model. This description XML file should be created by the application developer, either by hand or with the graphical user interface `wizard.tcl` available in `oasis4/util/gui`, to provide information about the application general characteristics but it is not used by the OASIS4 coupler.

The AD Schema is given in `oasis4/util/xmlfiles/ad.xsd`. The AD file name must be <application_local_name>_ad.xml where <application_local_name> is the application name.

The AD contains the element 'application' which is composed of (see the `ad.xsd`):

- the application name: attribute 'local_name', which should match argument `appl_name` of `PSMILe` call `prism_init` (see section 5.1.1);
- a description of the application: attribute 'long_name';
- the mode into which the application may be started: attribute 'start_mode': 'spawn', 'notspawn' or 'notspawn_or_spawn' (see section 4.1);
- the mode into which the application may run: attribute 'coupling_mode': 'coupled', 'standalone', or 'coupled_or_standalone';
- the arguments with which the application may be launched: element 'argument';
- the total number of processes the application can run on: element 'nbr_procs';
- the platforms on which the application has run: element 'platform';
- the list of components included in the application: element 'component'; for each component:

- the component name: attribute 'local_name', which should match the argument `comp_name` of `PSMILe` call `prism_init_comp` (see section 5.1.2);
- a description of the component: attribute 'long_name';
- whether or not this component is always active in the application: attribute 'default', either `true` or `false`);
- the number of processes on which the component can run (element 'nbr_procs').

6.3 The Potential Model Input and Output Description (PMIOD)

The Potential Model Input and Output Description (PMIOD) describes the relations a component model is potentially able to establish with the external environment through inputs and outputs. There should be one PMIOD per component model, written by the component developer, either by hand or with the graphical user interface `wizard.tcl` available in `oasis4/util/gui`, to describe its component potential coupling interface, but the PMIOD files are not used by the OASIS4 coupler.

The PMIOD Schema is given `oasis4/util/xmlfiles/pmiod.xsd`. The PMIOD file name should be <application_local_name>_<component_local_name>_pmiod.xml where <application_local_name> is the application name and <component_local_name> is the component name. Examples of PMIOD xml files for the toy coupled model TOYOA4 can be found in
`oasis4/examples/toyoa4/input`.

The PMIOD contains 3 types of information:

- general characteristics of the component
- information on the grids
- information on the coupling/IO fields, also called 'transient variables'

6.3.1 Component model general characteristics

This type of information gives an overview of the component model:

- the component name: attribute 'local_name' of element 'component', which should match the 2nd argument of `PSMILe` call `prism_init_comp`(see section 5.1.2);
- a short general description of the component model: attribute 'long_name';
- the name of the laboratory developing the component: element 'laboratory' in element 'code';
- the contact for additional information: element 'contact' in element 'code';
- the reference in the literature: element 'documentation' in element 'code';

6.3.2 Grids

This part contains information on the grids used by the component model. There might one or more grid per component. All grids should be described by the component developer in the PMIOD.

Each grid (element 'grid') is described by:

- the grid name: attribute 'local_name', which must match the 2nd argument `grid_name` of `PSMILe` call `prism_def_grid` (see section 5.3.1)
- for each global grid dimension: elements 'indexing_dimension':
  - the rank of the dimension: attribute 'index' which is of type integer
  - whether or not the global grid is periodic is this dimension (e.g. a global grid is periodic in `i` if index `imax` of the `grid_valid_shape`, see 5.3.1, is the neighbour of index `i=1`); attribute 'periodic' either `true` or `false`

6.3.3 Coupling/IO fields (transient variables)

Each coupling/IO field possibly received or provided by the component model from/to its external environment (another model or a disk file) through `prism_get` or `prism_put` call has to be described in the component PMIOD by one element 'transient' with the following attributes and sub-elements:

- attribute 'local_name': the field name (which must match 2^{nd} argument in the corresponding `PSMILe` call `prism_def_var` , see section 5.4.1);
- attribute 'long_name': gives a general description of the variable;
- element 'transient_standard_name': the standard variable names following the CF convention (if it exist). This uniquely identifies the nature of the coupling/IO field.

  In case of bundles, one element giving a generic name (e.g. `temperature`) plus one element per bundle species giving a specific name for the species (e.g `sea_water_temperature`, `air_temperature`, `snow_temperature`) need to be specified.
- element 'physics': a description of the coupling/IO field physical constraints:
  - attribute 'transient_type': the coupling/IO field physical type (either 'single' or 'bundle')
  - element 'physical_units': the coupling/IO field units
  - element 'valid_min': its physically acceptable minimum value
  - element 'valid_max': its physically acceptable maximum value
  - element 'nbr_bundles': for bundle variables, the number of bundles.
- element 'numeric', whose attribute 'datatype' provides the coupling/IO field numeric type: either `xs:real`, `xs:double`, or `xs:integer`
- element 'intent', which describes if the coupling/IO field may be exported or imported, or both. The sub-elements of 'intent' are:
  - element 'output': if the coupling field can be exported through `PSMILe` `prism_put` call (see section 5.6.1), this element shall contain:
    - element 'minimal_period', which is the period at which the `prism_put` is called in the code (to define this period the developer may specify a number of seconds, minutes, hours, days, months, and/or years, with respectively the sub-elements 'nbr_secs', 'nbr_mins', 'nbr_hours', 'nbr_days', 'nbr_months', 'nbr_years').
    - element 'source_transformation': the transformation that needs to be performed on the output coupling/IO field in the source component `PSMILe` ; if needed, this element contains only the element 'source_local_transformation' which in turn contains only the element 'scattering': the 'scattering' should be specified as `true` by the developer in the PMIOD and should not be changed by the user in the SMIOC. It is performed on an output coupling/IO field below the `prism_put` by the source `PSMILe` . It is required when grid information transfered to the `PSMILe` includes the masked points and when the array transfered to the `prism_put` API is a vector gathering only the non-masked points. Note that this complex structure is used here to specify scattering to be coherent with the 'source_transformation' structure of the SMIOC file (see 6.5)
  - element 'input': if the coupling/IO field can be imported through a `prism_get` call (see section 5.6.2), this element shall contain:
    - element 'minimal_period', which is the period at which the `prism_get` is called in the code (to define this period the developer may specify a number of seconds, minutes, hours, days, months, and/or years, with respectively the sub-elements 'nbr_secs', 'nbr_mins', 'nbr_hours', 'nbr_days', 'nbr_months', 'nbr_years').
    - element 'target_transformation': the transformation that needs to be performed on the input coupling/IO field in the target component `PSMILe` ; if needed, this element contains

only the element 'target_local_transformation' which in turn contains only the element 'gathering': the 'gathering' should be specified as `true` by the developer in the PMIOD and should not be changed by the user in the SMIOC. It is performed on an input coupling/IO field below the `prism_get` by the target `PSMILe` . It is required when the grid information transfered to the `PSMILe` covers the whole grid (masked points included), and when the array transfered through `prism_get` API is a vector gathering only the non-masked points. Note that this complex structure is used here to specify gathering to be coherent with the 'target_transformation' structure of the SMIOC file (see 6.5)

6.4 The Specific Coupling Configuration (SCC)

The Specific Coupling Configuration (SCC) contains the general characteristics and process management information of one coupled model simulation. There must be one SCC per coupled model (or per standalone application), named `scc.xml`, and written by the coupled model user either by hand or with the graphical user interface `oasis-gui.tcl` available in `oasis4/util/gui`.

The SCC Schema is given in `oasis4/util/xmlfiles/scc.xsd`.

After the call to `prism_init` in the application code, some of the SCC information is accessible directly by the model, with specific `PSMILe` calls (see section 5.2). In many cases, coherence with the compiling and running environment and scripts has to be ensured.

The SCC contains:

- some general information on the experiment defined by the user (element 'experiment'):
  - the experiment name (attribute 'local_name');
  - a description of the experiment (attribute 'long_name');
  - the mode into which all applications of the coupled model will be started (attribute 'start_mode': either `spawn` or `not_spawn`, see section 4.1); this user's choice, restricted by the possibilities given in the ADs, determines the way the applications should be started in the run script.
  - the number of processes used for the OASIS4 Driver/Transformer (element 'nbr_procs' of element 'driver') (this number must be equal to zero for a stand alone application)
  - the start date of the experiment (element 'start_date')
  - the end date of the experiment (element 'end_date')
- some general information on the current run, which therefore must be updated for each run of the experiment (element 'run'):
  - the start date of the run (element 'start_date'); the start date has to correspond to the lower bound of the time interval which is represented by the first time step of the run.
  - the end date of the run (element 'end_date'); the end date has to correspond to the upper bound of the time interval which is represented by the last time step of the run. Note that the end date of the current run has to be used as start date for the subsequent run.
- the list of applications chosen by the user (elements 'application'). For each chosen application:
  - the application name (as given in the corresponding AD) (attribute 'local_name') which must match argument `appl_name` of the `PSMILe` call `prism_init` ;
  - the application executable name, defined by the compiling environment (attribute 'executable_name') (used only in `spawn` mode as argument of the `MPI_Comm_Spawn_Multiple`).
  - whether the application stdout shall be redirected or not (user's choice) (attribute 'redirect', either `true` or `false`)

- a list of launching arguments (that should be chosen in the list given in the corresponding AD; used only in `spawn` mode as argument of the `MPI_Comm_Spawn_Multiple`) (elements 'argument');
- a list of hosts (elements 'host'); for each host:
  - the host name (attribute 'local_name') (used only in `spawn` mode as argument of the `MPI_Comm_Spawn_Multiple`).
  - the number of processes to run this host (element 'nbr_procs') (used in the `not_spawn` method to split the global communicator; for the `spawn` method, used as argument in `MPI_Comm_Spawn_Multiple`).
- the list of components activated (elements 'component', that should be chosen in the list given in the corresponding AD); for each component:
  - the component name (as given in the corresponding AD) (attribute 'local_name'), which must match the argument `comp_name` of `PSMILe` call `prism_init_comp` (see 5.1.2);
  - the lists of ranks in the total number of processes for the application (elements 'ranks'): The ranks are the numbers of the application processes (starting with zero) used to run the component model. They are given as lists of 3 numbers giving, in each list, a minimum value, a maximum value, and an increment value. For example, if processes numbered 0 to 31 are used to run a component model, this can be describe with one rank list (0, 31, 1); if processes 0 to 2 and 5 to 7 are used, this can be described with two rank lists (0, 2, 1) and (5, 7, 1).

6.5 The Specific Model Input and Output Configuration (SMIOC)

The Specific Model Input and Output Configuration (SMIOC) specifies the relations the component model will establish at run time with the external environment through inputs and outputs for a specific run. It must be generated by the user for each component model based on the corresponding PMIOD information, either by hand or with the graphical user interface `oasis-gui.tcl` available in `oasis4/util/gui`.

The SMIOC Schema is given in `oasis4/util/xmlfiles/smioc.xsd`. The SMIOC file name must be <application_local_name>_<component_local_name>_smioc.xml where <application_local_name> is the application 'local_name attribute and <component_local_name> is the component 'local_name' attribute in the scc.xml file. Examples of SMIOC xml files for the toy coupled model TOYOA4 can be found in `oasis4/examples/toyoa4/input`.

The SMIOC may contain 3 types of information detailed in the next paragraphs:

- general characteristics of the component, as described in the corresponding PMIOD
- information on the grids
- information on the coupling/IO fields, also called 'transient variables'

Part of this information is used to define attributes of the I/O NetCDF files but is not mandatory for the proper execution of the coupled model *per se*; if it is not specified in the SMIOC, it will just be missing in the I/O files. In the paragraphs below, it is detailed which information is mandatory and which is not.

6.5.1 Component model general characteristics

The SMIOC may repeat the description information provided about the component model general characteristics in the corresponding PMIOD (see section 6.3.1); however, the only mandatory information is the component name (see below).

The component model general characteristics are:

- the component name: attribute 'local_name' of element 'component', which should match the 2^{nd} argument of `PSMILe` call `prism_init_comp`(see section 5.1.2);

- a short general description of the component model: attribute ‘long_name’ (optional) ;
- the name of the laboratory developing the component: element ‘laboratory’ in element ‘code’ (optional) ;
- the contact for additional information: element ‘contact’ in element ‘code’ (optional) ;
- the reference in the literature: element ‘documentation’ in element ‘code’ (optional) ;

6.5.2 Grids

This part contains information on the grids effectively used during the run by the component model, based on the description done in the corresponding PMIOD file. There might one or more grids per component as described in the corresponding PMIOD.

Each grid (element’grid’) is described by:

- the grid name: attribute ‘local_name’, which must match the 2^{nd} argument `grid_name` of `PSMILe` call `prism_def_grid` (see section 5.3.1) (mandatory)
- for each global grid dimension: elements ‘indexing_dimension’ (mandatory for the dimensions that are effectively periodic):
  - the rank of the dimension: attribute ‘index’ which is of type integer
  - whether or not the grid is periodic is this dimension: attribute ‘periodic’ either `true` or `false`

6.5.3 Coupling/IO fields (transient variables)

Each coupling/IO field effectively received or provided by the component model from/to its external environment (another model or a disk file) through `prism_get` or `prism_put` call in the component code (see sections 5.6.1 and 5.6.2) must be specified by one element ‘transient’ which has the following attributes and sub-elements:

- attribute ‘local_name’: the field name, which must match 2^{nd} argument in the corresponding `PSMILe` call `prism_def_var` (see sections 5.4.1); (mandatory);
- attribute ‘long_name’: gives a general description of the variable; (optional)
- element ‘transient_standard_name’: one or more PRISM standard names following the CF convention (if they exist); see section 6.3.3 for details; (mandatory)
- element ‘physics’: a description of the coupling/IO field physical constraints; see section 6.3.3 for details; (optional)
- element ‘numeric’, whose attribute ‘datatype’ gives the coupling/IO field numeric type (either `xs:real`, `xs:double`, or `xs:integer`); (mandatory)
- element ‘intent’, which describes if the coupling/IO field will be exported or imported, or both (mandatory). This element contains in its sub-elements all coupling and I/O information (source and/or target, frequency, transformations, interpolation, etc.). The sub-elements of ‘intent’ are:
  - element ‘output’: When the coupling/IO field is exported through a `prism_put` , this export must be described in one or more elements ‘output’. The element ‘output’ is described in more details in section 6.5.4.
  - element ‘input’: When the coupling/IO field is imported through a `prism_get` , this import must be described in one element ‘input’. The element ‘input’ is described in more details in section 6.5.5.

6.5.4 The ‘output’ element

If the coupling/IO field is exported through a `prism_put` in the component code, it can be effectively be sent to none, one, or many targets, each target being described in one element ‘ouput’. A more detailed description of element 'output', its attributes and sub-elements is given here.

1. attribute 'transi_out_name' (mandatory) : a symbolic name defined by the user for that specific 'output' element.
2. element 'exchange_date' (mandatory) : The dates at which the coupling or I/O will effectively be performed. To express these dates, the user has to specify the following sub-element:
   - element 'period' (mandatory): The coupling or I/O is performed with a fixed period. To define this period, the user must specify a number of seconds, minutes, hours, days, months, and/or years, with respectively the sub-elements 'second', 'minute', 'hours', 'day', 'month', 'year' (all optional but at least one must be specified).
3. element 'corresp_transi_in_name' (mandatory) : The symbolic name of the corresponding input coupling/IO field origin (attribute 'transi_in_name' of element 'origin' of element 'input') in the target component or target file . This defines an exchange between a source and a target component or file. Coherence has to be ensured, i.e. the value of the current output 'transi_out_name' attribute (see above) has to be specified in the 'corresp_transi_out_name' element of the corresponding input coupling field origin (see also section 6.5.5). Note that this coherence is automatically ensured when using the graphical user interface `oasis-gui.tcl` (available in `oasis4/util/gui`) to create the SMIOC files.
4. element 'file' or element 'component_name' (one or the other mandatory): The target file description (I/O) or the target component 'local_name' attribute (coupling). The 'file' element is described in more detail in section 6.5.7.
5. element 'lag' (optional): The number of `prism_put` periods[1] to add to the output coupling field `prism_put` date and date_bounds to match the corresponding input coupling field `prism_get` date in the target component (see also 5.6.4).
6. element 'source_transformation' (optional) : The transformations performed on the output coupling/IO field in the source component `PSMILe` .
   - element 'source_time_operation' (optional) : for each grid point, the output coupling/IO field can be averaged (`taverage`) or accumulated (`accumul`) over the last coupling period by the source `PSMILe` and the result is transfered. Note that the average or the accumulation is simply done over the arrays provided as argument to the `prism_put` calls, not weighted by the time interval between these calls.
   - element 'statistics' (optional) : different statistics (minimum, maximum, integral) are calculated for the field on the masked points, and/or on the not masked points, and/or on all points of the output coupling/IO field, if respectively the sub-elements 'masked_points', and/or 'notmasked_points', and/or 'all_points' are specified with value 'on' or 'off'. This is done below the `prism_put` by the source `PSMILe` (after the time operations described in element 'source_time_operation' if any). These statistics are printed to the `PSMILe` log file for information only; they do not transform the output coupling/IO field.
   - element 'source_local_transformation' (optional) : the following local transformations may be performed on the output coupling/IO field by the source `PSMILe` :
     - element 'scattering': the 'scattering' should be specified by the developer in the PMIOD and should not be changed by the user in the SMIOC. If 'scattering' is `true`, scattering is performed on an output coupling/IO field below the `prism_put` by the source `PSMILe` . It is required when grid information transfered to the `PSMILe` includes the masked points and when the array transfered to the `prism_put` API is a vector gathering only the non-masked points.

[1]A `prism_put` period is the time between the `prism_put` date_bounds; e.g. for a lag of 1, the time added to the `prism_put` date and date_bounds arguments would be once the time difference between the associated date_bounds.

  - element ‘mult_scalar’ (optional) : Each grid point coupling/IO field value is multiplied by the scalar specified in this element.
  - element ‘add_scalar’ (optional) : The scalar specified in this element is added to each grid point coupling/IO field value.

    When these two elements are specified, the multiplication is performed before the addition.

7. element ‘debug_mode’ (optional) : either `true` or `false`; if it is `true`, the output coupling/IO field is automatically also written to a file below the `prism_put` . If there is a lag for this field (see above), the time in the debug file is the `date` argument of the `prism_put` call + lag. The field written to the restart file is also written to the `prism_put` debug file.

6.5.5 The ‘input’ element

If the coupling/IO field is imported through a `prism_get` in the component code, the user will have to describe one source for that field in the SMIOC. A more detailed description of element ‘input’, its attributes and sub-elements is given here.

1. element ‘exchange_date’ (mandatory) : The dates at which the coupling or I/O will effectively be performed (see ‘exchange_date’ in ‘output’ in section 6.5.4).
2. element ‘origin’ (mandatory): In the current OASIS4 version, an input coupling/IO field may come only from one origin being described by an element ‘origin’ which contains the following attributes and sub-elements:
   - attribute ‘transi_in_name’ (mandatory) : a symbolic name defined for that specific ‘origin’ element.
   - element ‘corresp_transi_out_name’ (mandatory) : The symbolic name of the corresponding output coupling/IO field (attribute ‘transi_out_name’ of element ‘output’) in the source component or source file. This defines an exchange between a source and a target component or file. Coherence has to be ensured, i.e. the value of the current input ‘transi_in_name’ attribute has to be specified in the ‘corres_transi_in_name’ element of the corresponding output coupling field (see also section 6.5.4). Note that this coherence is automatically ensured when using the graphical user interface `oasis-gui.tcl` (available in `oasis4/util/gui`) for creating the SMIOC files.
   - element ‘file’ or ‘component_name’ (one or the other mandatory) : The source file description (I/O) or the source component ‘local_name’ attribute (coupling). The ‘file’ element is described in more detail in section 6.5.7.
   - element ‘middle_transformation’ (optional): The transformations which link the source and the target.
     - element ‘interpolation’ (mandatory): The interpolation to be performed on the output coupling field to express it on the target model grid. This element is described in more detail in section 6.5.6.
3. element ‘target_transformation’ (optional) : The transformations performed on the input coupling/IO field in the target component `PSMILe` .
   - element ‘target_local_transformation’ (optional) : The local transformations performed on the input coupling/IO field.
     - element ‘gathering’ (optional) : The ‘gathering’ should specified by the developer in the PMIOD and should be kept as is in the SMIOC. If ‘gathering’ is `true`, it is performed on an input coupling/IO field below the `prism_get` by the target `PSMILe` . It is required when the grid information transfered to the `PSMILe` covers the whole grid (masked points

included), and when the array transfered through `prism_get` API is a vector gathering only the non-masked points.

  - element 'mult_scalar' (optional) : Each grid point coupling/IO field value is multiplied by the scalar specified in this element.
  - element 'add_scalar' (optional) : The scalar specified in this element is added to each grid point coupling/IO field value.

    When both operations are chosen, the multiplication is performed before the addition.

- element 'target_time_operation' (optional) : Target time interpolation is supported below the `prism_get` only for IO data[2]. The types of time interpolation are the nearest neighbour 'time_nneighbour' and linear time interpolation between the two closest timestamps 'time_linear' in the input file.
- element 'statistics' (optional) : see section 6.5.4.

4. element 'debug_mode' (optional) : either `true` or `false`; when it is `true`, the input coupling/IO field is automatically written to a file below the `prism_get` . Note that if there is a lag, the field read from the restart file is written to the `prism_get` debug file (but not to the `prism_put` debug file).

6.5.6 The element 'interpolation'

The element 'interpolation' is a sub-element of 'middle_transformation', which is a sub-element of 'origin', which is a sub-element of 'input'. The interpolation is needed to express the coupling field on the target model grid[3].

As all coupling arrays are given on a 3D grid, the user has to choose among the following:

- 'interp3D': A full 3D interpolation.
- '(interp2D, interp1D)': The same 2D interpolation for all vertical levels followed by a 1D interpolation in the vertical. This type of interpolation can be used for all grids which vertical dimension can be expressed as z(k), i.e. for all grid types currently supported besides `PRISM_gridless` (see table 5.8). The mask may vary with depth. Currently, the combinations implemented are `nneighbour2D` and `none`, `bilinear` and `none`, `bicubic` and `none`, `conservativ2D` and `none`, `nneighbour2D` and `linear`, `bilinear` and `linear`.

Note that the interpolation will provide values interpolated from the source field for all target grid cells except for the following ones:

- the target cell does not intersect any part of the source grid domain; for those cells, the target field keeps the same value as before the call to `prism_get` ;
- the target cell is masked; for those cells, the target field keeps the same value as before the call to `prism_get` ;
- the target cell is not masked, but the interpolation as requested in the SMIOC file cannot be performed (see for example the element 'novalue' here below); for those cells, the target field will take the `psmile_dundef` value (=-280177.).

The elements 'interp3D', 'interp2D', 'interp1D', are separately described here after:

[2]This feature is not essential for coupling data as each `prism_put` has a date and date_bounds as arguments. Therefore, a `prism_put` and a `prism_get` will be matched if the `prism_get` date falls into the date_bounds of the `prism_put` . Allowing for time interpolation, e.g. allowing a `prism_get` to match with an averaged value of the two `prism_put` nearest neighbour in time, could lead to deadlocks as the model performing the `prism_get` would be blocked until the two `prism_put` nearest neighbour in time are performed. We rely only the date_bounds to match `prism_put` and `prism_get` having non matching dates.

[3]In the current OASIS4 version, interpolation is available only for coupling fields and not for I/O fields read/written from/to a file.

1. element ‘interp3D’: For 3D interpolation, the user has to choose among the following methods:
   - element ‘nneighbour3D’: A 3D nearest neighbour algorithm; the parameters are:
     - element ‘nbr_neighbours’ (mandatory): the number of source points used to calculate each target point
     - element ‘gaussian_variance’ (optional) : the variance of the Gaussian function used to weight the neighbours, if any.
     - element ‘para_search’ (optional) :
       - `global`: global search which identifies neighbours across source domain boundaries on neighbouring processing elements if necessary (default).
       - `local`: a local less expensive neighbourhood search; the results will be affected by the source grid partitioning.
     - element ‘if_masked’ (optional) : only `novalue` is currently available for ‘nneighbour3D’
       - `novalue`: if some of the nbr_neighbours neighbours are masked, `psmile_undef` value is given to that target point.
   - element ‘trilinear’: A trilinear algorithm; the parameters are:
     - element ‘para_search’ (optional): see element ‘nneighbour3D’ above.
     - element ‘if_masked’ (mandatory): either `novalue`, `tneighbour`, or `nneighbour`.
       - `novalue`: if some of the 8 trilinear neighbours are masked, `psmile_undef` value is given to that target point;
       - `tneighbour`: if some of the 8 trilinear neighbours are masked, the non-masked points among those points are used for calculating a weighted average; if the nbr_neighbours neighbours are masked, `psmile_undef` value is given to that target point;
       - `nneighbour`: if some of the 8 trilinear neighbours are masked, the non-masked points among those points are used for calculating a weighted average; if the nbr_neighbours neighbours are masked, the non-masked nearest neighbour is used.
   - element ‘user3D’: the set of weights and addresses used for the remapping are pre-defined by the user and are stored in a file that will be read by the `PSMILe` library (see 4.3.3 for more details). For this remapping, only the file containing the weights and addresses defined by the user is needed:
     - element ‘file’ (mandatory) : the file containing the user-defined weights and addresses (see section 6.5.7 for the content of this element)

2. element ‘interp2D’: For 2D interpolation, the following methods can be chosen:
   - element ‘nneighbour2D’: A 2D nearest neighbour algorithm; the parameters are:
     - elements ‘nbr_neighbours’, ’gaussian_variance’, ‘para_search’, ‘if_masked’: see element ‘nneighbour3D’ above
   - element ‘bilinear’: A bilinear algorithm; for the parameters are:
     - element ‘para_search’ (optional) : see element ‘nneighbour3D’ above.
     - element ‘if_masked’: either `novalue`, `tneighbour`, or `nneighbour`.
       - `novalue`: if some of the 4 bilinear neighbours are masked, `psmile_undef` value is given to that target point;
       - `tneighbour`: if some of the 4 bilinear neighbours are masked, the non-masked points among those points are used for calculating a weighted average; if the nbr_neighbours neighbours are masked, `psmile_undef` value is given to that target point;
       - `nneighbour`: if some of the 4 bilinear neighbours are masked, the non-masked points among those points are used for calculating a weighted average; if the nbr_neighbours neighbours are masked, the non-masked nearest neighbour is used.

- element 'bicubic': A bicubic algorithm, the parameters are:
  - element 'bicubic_method' (mandatory) : The bicubic method: either `gradient` (the 4 enclosing source neighbour values and gradient values based on the 12 additional enclosing neighbours are used), or `sixteen` (the sixteen enclosing source neighbour values are used -this method assumes that the source points are located 4 by 4 at the same latitude).
  - element 'para_search' (optional) : see element 'nneighbour3D' above.
  - element 'if_masked': : either `novalue`, `tneighbour`, or `nneighbour`.
    - `novalue`: if some of the 16 bicubic neighbours are masked, `psmile_undef` value is given to that target point;
    - `tneighbour`: if some of the 16 bicubic neighbours are masked, the non-masked points among those points are used for calculating a weighted average; if the nbr_neighbours neighbours are masked, `psmile_undef` value is given to that target point;
    - `nneighbour`: if some of the 16 bicubic neighbours are masked, the non-masked points among those points are used for calculating a weighted average; if the nbr_neighbours neighbours are masked, the non-masked nearest neighbour is used.
- element 'conservativ2D': A 2D conservative remapping is applied: the weight of a source cell is proportional to the target cell area intersected by the source cell. See section 4.3.1 for details.

  The 2D conservative remapping parameters are:
  - element 'order' (mandatory) : Currently, the only possible value is 'first' as only the first order conservative remapping is available.
  - element 'normalisation2D' (optional): 2D normalisation options:
    - element 'methodnorm2D' (mandatory): the value of the normalisation method can be:
      - `fracarea`: The sum of the non-masked source cell intersected areas is used to normalise each target cell field value: the flux is not locally conserved, but the flux value itself is reasonable.
      - `destarea`: The total target cell area is used to normalise each target cell field value even if it only partly intersects non-masked source grid cells: local flux conservation is ensured, but unreasonable flux values may result.
      - `none`: No normalisation is applied.
  - element 'para_search' (optional) : see element 'nneighbour3D' above.

3. element 'interp1D' For 1D interpolations, the following methods can be chosen:
   - element 'linear':

     A linear algorithm is applied.
   - element 'none':

     Interpolation method that can be chosen for dimension with extent of 1. For example, to interpolate a field of Sea Surface Temperature dimensioned (i,j,k) with extent of k being 1, the interpolation type can be '(interp2D, interp1D)' and 'none' should be chosen for the 'interp1D'.

6.5.7 The 'file' element

The 'file element is composed of the following sub-elements:

- element 'name': a character string used to build the file name.
- element 'suffix': either `true` or `false`. When 'suffix' is false (by default), the file name is composed only of element 'name'; when it is true, the file name is composed of element 'name' to

which the PRISM suffix for dates is added. When the file is opened for writing, the suffix will be "_out.<job_startdate>.nc", where <job_startdate> is the start date of the job. When the file is opened for reading, the suffix should be "_in.<start_date>.nc", where <start_date> is the date of the first time stamp in that file. When reading an input from a file, the `PSMILe` will automatically match the requested date of the input with the appropiate file if it falls into the time interval covered by that file. The <job_startdate> and <start_date> must be written according to the ISO format yyyy-mm-ddTHH:MM:SS. The date/time string in the file name must have to format yyyy-mm-ddTHH.MM.SS since the colon is already used in other context for file systems. An example of an input file with 'suffix' = false is `SONSHLDO.nc` available in `oasis4/example/toyoa4/data`).

- element 'format': the format of the file; only NetCDF (`mpp_netcdf`) supported for now.
- element 'io_mode': either `iosingle` (by default) or `distributed`. The mode `iosingle` means that the whole file is written or read only by the master process; `distributed` means that each process writes or reads its part of the field to a different partial file. Note that if the PSMILe is linked against the parallel NetCDF library Li et al. (2003), the `parallel` mode will automatically be used; in this case each process writes its part of the field to one parallel file (see also our remarks about parallel NetCDF on page 28).
- element 'packing': packing mode , either `1`, `2`, `4` or `8` (for NetCDF format only)
- element 'scaling': if present, the field read from the file are multiplied in the `PSMILe` by the 'scaling' value (1.0 by default) (for NetCDF format only)
- element 'adding': if present, the 'adding' value (0.0 by default) is added to the field read from the file (for NetCDF format only)
- element 'fill_value': on output, specifies the value given to grid points for which no meaningfull value was calculated; on input, specifies the value given in the file to undefined or missing data.

Chapter 7

Compiling and running OASIS4 and TOYOA4

This chapter describe how to compile and run the OASIS4 coupler and its toy coupled model “TOYOA4”. It also describes how to get internal CPU and elapse time statistics for the`PSMILe` library and the Driver/Transformer.

7.1 Introduction

The list of platforms onto which OASIS4 was successfully compiled and run is avaliable on OASIS web site (https://verc.enes.org/models/software-tools/oasis/) under the ‘Technical’ tab on the ‘Tested and validated platforms for OASIS4’ page.

7.2 Compiling OASIS4 and its associated PSMIle library

Compiling is done using the top makefile `TopMakefileOasis4`, platform dependent header files (see section 7.2.1) and low-level makefiles in each source directory. During compilation, the `ARCHDIR` directory specified in the header file is created. After successful compilation, resulting executables are found in `$ARCHDIR/bin`, libraries in `$ARCHDIR/lib` and object and module files in `$ARCHDIR/build`.

7.2.1 Compilation with TopMakefileOasis4

Compiling OASIS4 using the top makefile `TopMakefileOasis4` is done in directory `oasis4/util/make_dir`. `TopMakefileOasis4` must be completed with a header file `make.`*yours* specific to the compiling platform used and specified in `oasis4/util/make_dir/make.inc`. One of the files `make.pgi_cerfacs`, `make.sx_frontend` or `make.aix` can by used as a template. The root of the OASIS4 tree can be anywere and must be set in the variable `COUPLE` in the `make.`*yours* file. The choice of MPI1 or MPI2 is also done in the `make.`*yours* file (see `CHAN` therein).

The following commands are available:

- `make -f TopMakefileOasis4`

  compiles OASIS4 libraries *common_oa4*, *psmile_oa4* and *mpp_io* and creates OASIS4 Driver/Transformer executable oasis4.MPI[1/2].x ;
- `make help -f TopMakefileOasis4`

  displays help information ;

- `make realclean -f TopMakefileOasis4`:

  cleans OASIS4 Driver/Transformer executable and libraries.

Log and error messages from compilation are saved in the files COMP.log and COMP.err in `make_dir`.

For not compiling the mpp_io library, the variable `PSMILE_WITH_IO` must be left undefined in the file make.*yours* .

7.2.2 Some details on the compilation

- Other librairies needed

  The following librairies (not provided with the OASIS4 sources) are required:

  - Message Passing Interface, MPI1 Snir et al. (1998) or MPI2 Gropp et al. (1998) (MPICH, openMPI, LAM-MPI, SGI native MPI, NEC SX native MPI, and SCAMPI were successfully tested)
  - NetCDF Version 3.4 or higher Eaton et al. (2003) or parallel NetCDF Li et al. (2003) (see page 5.6)
  - libxml Version 2.6.5 or higher [1]

- CPP keys

  The following CPP keys can be activated:
  (see `CPPDEF` in `oasis4/util/make_dir/make.xxx` files)

  - PSMILE_WITH_IO: to make use of the IO capability of `PSMILe`
  - PRISM_WITH_MPI1: This option has to be chosen if the available MPI library supports only MPI1 standard, like mpich1.2.*. Correct behaviour is ensured only on 32 bit architectures. This key is mutually exclusive with the PRISM_WITH_MPI2 key.
  - PRISM_WITH_MPI2: When the available MPI2 library supports the MPI2 standard, this option should be chosen instead (in particular on 64-bit architectures). This key is mutually exclusive with the PRISM_WITH_MPI1 key.
  - DONT_HAVE_STDMPI2: This key has to used in conjunction with PRISM_WITH_MPI2 for partial MPI2 implementation (e.g. on IBM Power 6 and with SCALI MPI). If activated, the MPI2 spawn functionality MPI_Comm_spawn_multiple will not be used. MPI_Finalized, MPI_Allreduce with MPI_IN_PLACE as first argument, and MPI_Waitall with MPI_STATUSES_IGNORE as 3rd argument will not be used either. Note that in this case, the element `start_mode` has to be `not_spawn` in the SCC.xml file.
  - PRISM_LAM: if LAM-MPI library is used.
  - DONT_HAVE_ERRORCODES_IGNORE: As a workaround for some MPI2 implementations that do not support the MPI parameter MPI_ERRORCODES_IGNORE (as before last argument to MPI_Comm_spawn_multiple call) this key has to be activated. If at all, it is only needed in conjunction with PRISM_WITH_MPI2.
  - SX: To achieve better performance on vector architecture this option should be set.
  - VERBOSE: Useful for debugging purposes, activation this key will cause the library and driver routines to run in verbose mode. Since all output is immediately flushed to standard output this will significantly decrease performance and is therefore not recommended for production runs.
  - DEBUG: Mainly used by OASIS4 developers. Activating this option will cause the driver and library to write out additional output for debugging purpose. This output is immediately flushed to standard output and will therefore decrease performance.

[1] http://www.w3.org/XML

  - PRISM_ASSERTION: Mainly used by OASIS4 developers; the code encapsulated by this cpp key will perform additional internal consistency checks and will provide additional information for debugging.
  - NAG_COMPILER: Mandatory to compile and run with the NAG compiler.

7.2.3 Remarks and known problems

- LAM-MPI with the `spawn` approach

  The usage of `MPI_Comm_Spawn_Multiple` is the most portable way if MPI processes shall be dynamically spawned on multiple hosts. Therefore, there is a reserved predefined key "host" for the info argument, which specifies the value of the host name, in the MPI2 standard. Nevertheless this is currently not supported by LAM-MPI. Therefore, to use LAM-MPI, it is required to use the CPP key PRISM_LAM. In this case, LAM-MPI `MPI_Comm_Spawn_Multiple` fills the processors according to the list given in the `lam.config` file used by the lamboot process (see example in `https://oasistrac.cerfacs.fr/browser/trunk/prism/dev_ex/examples/simple-mg`), using always all processors on a given node. For example, 1 Driver/Transformer process and 4 processes for the ocean and the atmosphere models would be scheduled on three 4-CPU hosts like the following: the Driver/Transformer would be on host 1, the ocean model would have 3 processes on host1 and 1 process on host 2, and the atmosphere model would have 3 processes run on host 2 and 1 on host 3, which of course is not optimal.

  With `MPI_Comm_Spawn` , LAM-MPI would be more more flexible regarding the spawning of processes. For OASIS4 this is not an option since `MPI_Comm_Spawn_Multiple` is required for

  - starting multiple binaries (not several applications); this may be required for an heterogenous cluster;
  - starting same binary with a multiple set of arguments;
  - placing multiple binaries in the same MPI_COMM_WORLD. It is intended here to place the MPI processes of an application into a MPI_COMM_WORLD which is different for each application (as in this case, the applications are not required to change the application internal communicators).

  Therefore, the `spawn` approach is not recommended with LAM-MPI. The `not_spawn` approach (see sections 4.1) should be prefered if possible.

- MPICH

  Since MPI1 is not designed for 64 Bit architectures the default MPICH.1.2.* implementation will not work on 64 Bit systems for OASIS4 and `PSMILe`. It could work on IA64 if there was no use of functions with INTEGER arguments representing an address or a displacement as is the case in OASIS4 (on IA64 architectures these integers must be 64 bits or "long" in C language; they are "int" in MPICH) .

- Portland Group Compiler

  The Portland Group Compiler Version 5.2 produces an internal compiler error for the main routine of OASIS4.

  For the Portland Group Compiler Version 6.0, the debug option (-g) must be used. No particular option is needed for version 6.1 .

  The Portland Group C compiler produces an error. In particular, with PGCC 8.0.5 and 9.0.4, an error was observed when compiling `parser.h` included in the C routine for XML reading `sasa_c_xml.c`. Use of GNU C compiler gcc is recommended instead (see `CC` in `oasis4/util/make_dir/make.xxx` files.

- Intel Fortran Compiler

To successfully compile OASIS4, Intel Fortran Compiler version 11.1.046 or higher is required (a problem with pointers pointing on pointers was detected with previous versions).

7.3 Compiling and running TOYOA4

TOYOA4, which sources, input files, data and running script are in directory `oasis4/examples/toyoa4`, is a toy coupled model providing a practical example of the coupling and I/O exchanges that can occur in a real coupled model. It is a 'toy' coupled model in the sense that the components atmoa4, oceoa4, and lanoa4 do not contain any real physics or dynamics but their coupling and I/O exchanges are realistic (i.e. the grids and the coupling fields have realistic dimensions and the exchanges and transformations performed by OASIS4 are realistic). NetCDF data files needed for running TOYOA4 are found in directory `/data`. The description and configuration XML files are found in directory `/input`. Note that the toy model available in `oasis4/examples/tutorial1` reproduces ping-pong exchanges between `model1` and `model2` (see the readme_tutorial1.pdf therein); this is probably the simplest toy model available to start learning about OASIS4.

Compiling is done with the `Makefile` in this directory. Running is done by adapting the "User's section" of the running script `oasis4/util/runscripts/run_examples_all` and by invoking it from the `oasis4/examples/toyoa4` directory (i.e. with `../../util/runscripts/run_examples_all`). The working directory `rundir` defined in `run_examples_all` is created; all files and executables needed for running are first copied into this working directory and the TOYOA4 coupled model is executed.

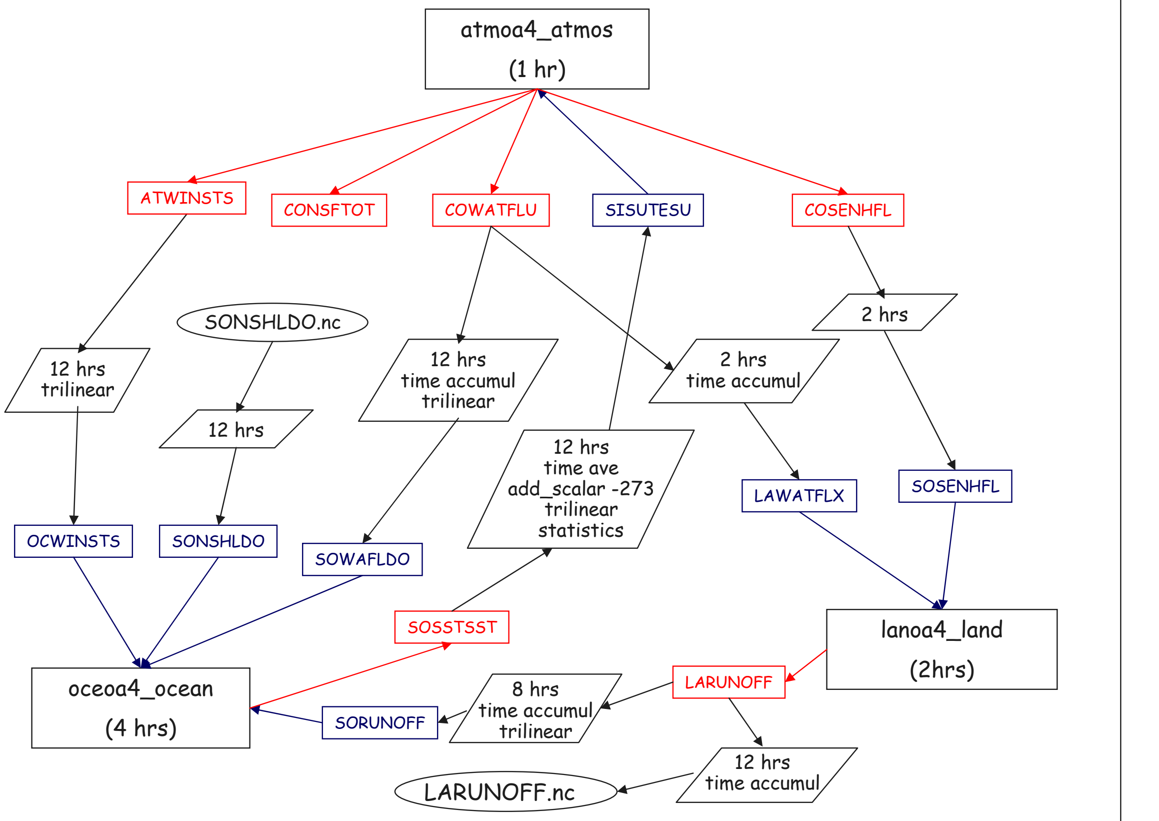


Figure 7.1: TOYOA4 toy coupled model coupling and I/O configuration

Figure 7.1 illustrates the coupling and I/O exchanges occuring between the 3 toy component models atmoa4, oceoa4, and lanoa4.

Both atmoa4 and lanoa4 work on a T31 Gaussian grid, but their parallel partitioning is a function of their number of processes which can be different. The third model, oceoa4, is not parallel and uses a real ocean model stretched and rotated grid with spherical polar coordinates of 182 x 149 grid points.

All coupling and I/O fields are scalar fields. The model atmoa4 declares 1 input field `SISUTESU`, and 4 output field `CONSFTOT, COSENHFL, COWATFLU, ATWINSTS` as is listed in its PMIOD file `atmoa4_atmos_pmiod.xml`. The model lanoa4 declares 2 input fields `LAWATFLX` and `SOSENHFL`, and 1 output field `LARUNOFF` as is listed in its PMIOD file `lanoa4_land_pmiod.xml`. The model oceoa4 declares 4 input fields `SONSHLDO, SOWAFLDO, SORUNOFF` and `OCWINSTS`, and 1 output field `SOSSTSST`.

At run-time, the OASIS4 Driver/Transformer and the PSMILe model interface linked to the component models act according to the specifications written by the user in the configuration SMIOC XML files.

In the atmoa4 SMIOC file `atmoa4_atmos_smioc.xml`, it is specified that `ATWINSTS` will be sent to oceoa4, `COSENHFL` to lanoa4, `COWATFLU` both to oceoa4 and lanoa4, while `CONSFTOT` is not sent at all; it is also specified that `SISUTESU` will come from oceoa4. The lanoa4 SMIOC file `lanoa4_land_smioc.xml` specifies that `LARUNOFF` will both go to oceoa4 and be written to a file `LARUNOFF.nc` and that `LAWATFLX` and `SOSENHFL` will be received from atmoa4. Finally, in the oceoa4 SMIOC file `oceoa4_ocean_smioc.xml`, it is specified that `OCWINSTS` and `SOWAFLDO` will be received from atmoa4, `SORUNOFF` from lanoa4, while `SONSHLDO` will be read from a file `SONSHLDO.nc`; `SOSSTSST` will be sent to atmoa4.

Different operations are performed by the PSMILe model interface on the coupling or I/O fields such as statistics, time accumulation time averaging, as specified in the SMIOC files. The exchanges of the coupling fields between atmoa4 and lanoa4 (and vice-versa) are direct, involving possibly some repartitioning if their parallel partitioning are different. As atmoa4 and oceoa4 do not have the same grid, their exchanges of coupling fields go through the Transformer (not illustrated on figure 7.1) where a linear interpolation is performed. The different coupling and I/O periods are also specified in the different SMIOC files.

TOYOA4 also illustrates the use of a coupling restart file for field `COSENHFL` for which a positive lag of 1 is defined. The first time TOYOA4 is run, the variable `run` should be set to `start` in `run_examples_all`. In that case, the file `scc.xml.start` is copied in `scc.xml` and used, TOYOA4 is run for 3 days starting January 1^{st} 2000, and the first field `COSENHFL` received by lanoa4 comes from the restart file `COSENHFL_atmoa4_atmos_rst.2000-01-01T00_00_00.nc`; at the end of the run, the restart file for the next run, `COSENHFL_atmoa4_atmos_rst.2000-01-04T00_00_00.nc`, is created by the last call to prism_put for `COSENHFL` in atmoa4. A next run of 3 days starting January 4^{th} 2000 can then be run by changing `run=restart` in `run_example_all` and running it again.

A successfull execution of TOYOA4 (with `run` set to `start` in `run_examples_all`) produces files that can be compared to results in `oasis4/examples/toyoa4/outdata`. In particular, files containing standard output from the different components (e.g. atmoa4.0, lanoa4.0, oceoa4.0) should end with lines like

```
-------------------------------------
--- Note: MPI_Finalize was called ---
---       from prism_terminate.   ---
-------------------------------------
```

7.4 Getting some internal CPU and elapse time statistics

This section describes how to get some CPU and elapse time statistics for the internal `PSMILe` and Driver/Transformer routines using the routines in module `oasis4/lib/common_oa4/src/psmile_timer.F90`. To use this functionality, one has to:

- call `psmile_timer_init` at the beginning of the code with, as arguments, the number of measures, a vector giving a label for each measure, the application name, the name of the file where the statistics will be printed out, and the local communicator of the application (see an example in `prism_init.F90`)
- for each x measure of time, call `psmile_timer_start(x)` and `psmile_timer_stop(x)`; if these two routines are called multiple times for the same x, the time will get accumulated
- call `psmile_timeprint` at the end of the run (see example in `prism_terminate.F90`)

The statistics will get printed for all processes in a file with the name given as argument of the `psmile_timer_init`. Currently, x=1 and x=2 are used under CPP key `PROFILE` in the `PSMILe` to measure the total time and the time used in `prism_enddef` (see `prism_init.F90`, `prism_enddef.F90` and `prism_terminate.F90`). In the Driver/Transformer, x=1 is used under CPP key `PROFILE` to measure the total time used by the Driver/Transformer (see `prismdrv_init_appl.F90` and `prismdrv_finalize.F90`).

Bibliography

Balaji, V., 2001: Parallel Numerical Kernels for Climate Models. W. Zwieflhofer and N. Kreitz, eds., *DEVELOPMENTS IN TERACOMPUTING: Proceedings of the Ninth ECMWF Workshop on the Use of High Performance Computing in Meteorology*, 277 – 295, European Centre for Medium-Range Weather Forecasts, World Scientific Press, Reading, Reading, UK.

Eaton, B., J. Gregory, B. Drach, K. Taylor, and S. Hankin, 2003: NetCDF Climate and Forecast (CF) Metadata Conventions.

Gropp, W., S. Huss-Lederman, A. Lumsdain, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir, 1998: *MPI: The Complete Reference. Vol 2: The MPI-2 extensions.*. MIT Press.

Jones, P., 1999: First- and Second-order Conservative Remapping Schemes for Grids in Spherical Coordinates. *Mon. Weath. Rev.*, **127**, 2204 – 2210.

Li, J., W. Liao, A. Choudhary, R. Ross, R. Thakur, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, 2003: Parallel NetCDF: A High-performance Scientific IO Interface. *Proceedings of the SC'03, Nov 15-21*, Phoenix, Arizona, USA.

Redler, R., S. Valcke, and H. Ritzdorf, 2010: OASIS4 - A Coupling Software for Next Generation Earth System Modelling. *Geoscience Model Development*, **3**, 87–104.

Snir, M., S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, 1998: *MPI: The Complete Reference. Vol 1: The MPI Core*. MIT Press.

Index