OASIS4

User – defined interpolation

with a

weight-and-address file

Developer's Guide

Jean Latour

Sophie Valcke

June 2011

CERFACS Working Notes WN-CMGC-11-48

Index

1) DRIVER AND PSMILE LOGIC OVERVIEW

1.1) General picture

Usual interpolation algorithms are based on a geographical localization of the points or cells of 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 or since 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. In order to achieve this, the user has to define, in a separate file, the links associating a specific point of the target grid, with some specific points of the source grid and the weights corresponding to each link. This is the "user-defined weights and addresses file. Figures 1 and 2 illustrate the concept of the user-defined remapping.

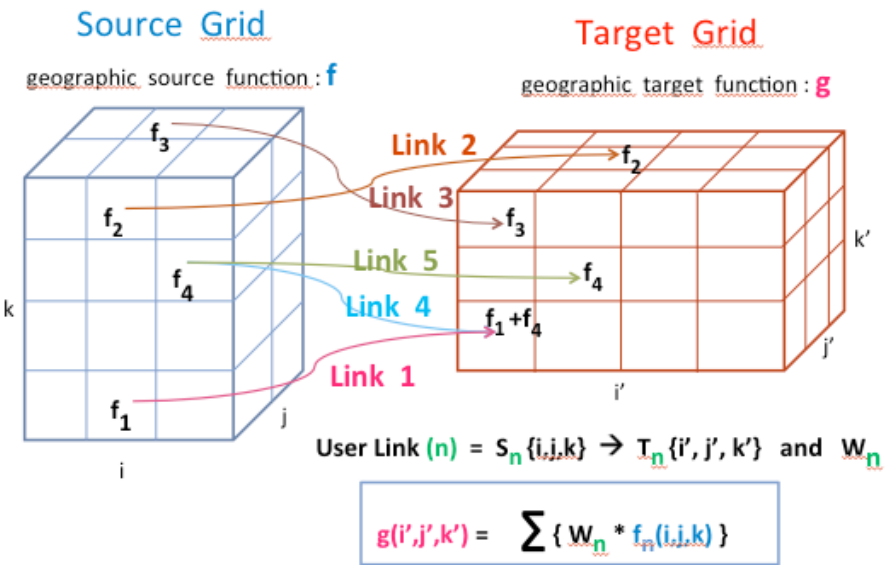


Figure 1 User-defined remapping

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 2: Gridless function and content of weight-and-address file

For each component of OASIS4, the user defines the grids, the coupling fields (transients), … i.e. all objects that will be involved in the coupling process, through XML input files containing a number of elements and attributes. These informations are gathered in OASIS4 internal structures, described by Fortran derived types. Since the number of components, or the number of transients for example is not know in advance, all of these data are gathered in dynamic arrays : allocatables arrays or pointers. Once the XML information is fully processed by the driver, this information is sent to each component model. Following the receives, each component organizes these data in their own internal structures, through several calls to PSMILe subroutines.

The modifications introduced by the "new" (for OASIS4) interpolation method, the user-defined interpolation with a Weight and Addresses file, are done within this general framework. They are limited to a few subroutines in the driver, and a few subroutines in the PSMILe user Interface. Several new data structures are also defined.

The motivations for these modifications are now described:

1.2) Constraints resulting from the Driver's code structure.

1.2.1) Transient description in XML files, target side :

We want to keep the user "manual" description of the interpolation in the SMIOC as simple as possible: the user only needs to give a few elements under the interpolation method description in a specific <origin> element of the target geographic transient function.
For example : the <origin> element of the target component will contain :

```
<origin transi_in_name="target_fnc_in1">
  <corresp_transi_out_name>source_fnc_out1</corresp_transi_out_name>
  <component_name>source</component_name>
  <middle_transformation>
   <interpolation>
    <interp3D>
     <user3D>
      <file>
       <name>weights_addresses.nc</name>
       <format>mpp_netcdf</format>
       <io_mode>iosingle</io_mode>
      </file>
     </user3D>
    </interp3D>
   </interpolation>
  </middle_transformation>
 </origin>
```

1.2.2) Transient description in XML files, source side :

The principle of the user-defined interpolation is to construct a new transient function based on a new "gridless" grid (See the User's Guide for this interpolation). However these new objects should be fully transparent to the user, they have to be created automatically by the driver, and the PSMILe when they "detect" the specific method "user3D" in the SMIOC, for each specific transient origin. This unique transient origin in a "target" component is coupled to a unique transient output in a "source" component, and the correspondence between the two is established by the driver. For this, the driver

compares : 1) the origin transi_in_name of the target, to the corresp_transi_in_name of the output transient in the source component, and 2) the corresp_transi_out_name of the origin transient in the target component to the transi_out_name of a specific transient output in the source component.
For example : the output description in the source component can be :

```
<output transi_out_name="source_fnc_out1">
 <minimal_period>
   <nbr_hours>1</nbr_hours>
 </minimal_period>
 <exchange_date>
  <period>
   <hour>1</hour>
  </period>
 </exchange_date>
 <corresp_transi_in_name>target_fnc_in1</corresp_transi_in_name>
 <component_name>target</component_name>
 <source_transformation>
  <statistics>
  </statistics>
  <source_local_transformation>
  </source_local_transformation>
 </source_transformation>
 <debug_mode>false</debug_mode>
</output>
```

1.2.3) Matching origin and output transients :

For each transient, the driver will search the details of the XML files in order to detect a "user3D" interpolation method. For a specific transient and a specific input origin it will generate the internal structures of a new transient ("user-defined transient"). This new transient has only ONE input origin in which the names (transi_in_name, and corresp_transi_out_name) are copied from those defined by the user for the geographic transient, plus a constant suffix in order to distinguish the geographic transient from the "user defined" transient.
During the processus of matching origins and outputs, the driver will detect in a specific source component which output of which transient is to be coupled with this transient origin. So for the source component detected, the driver will associate a new transient ("user defined transient"). This new transient in the source component, will have only ONE output, again with names (transi_out_name and corresp_transi_in_name) copied from those of the geographic transient plus the same constant suffix.

1.2.4) Total number of transients :

The number of transients in a component in NOT just known by a scan of the XML files with the routine "get_smioc_numbers". Before this, the driver has to go through all XML details in order to detect all "user3D" interpolations (N for example) and it has to create an equal number of transients with one input origin, and an equal number of transients with one output. These 2N new transients will match 2 by 2 during the "name-matching" process, in the same way as the geographic transients created by the user.

Before any information is sent to the components by the driver, the XML analysis has to be completed and the right number of transients calculated : those defined by the user for

the geographic variables, plus the new additional transients created each time a "user3D" method is detected. This is mandatory, in order to keep the actual PSMILe logic and structures unchanged. For the PSMILe, when the component is NOT stand_alone (which is the case with interpolations), all dimensioning numbers are received from the driver, so we must have the right dimensions for all structures in order to include the "user defined" transients.

Note that the detailed exploration of the SMIOC files (**get_transi_details**) is done only once by the driver. The actual logic is maintained in the new version : XML details are read in a loop on the components (i.e. on the XML files), and stored in a local array of structures : **sla_driver_transi**(:) (array of all transients in one component). This array is then copied in a global array : **sga_smioc_transi**(:) containing ALL transients of all components, one after the other.
During the detailed exploration of the SMIOC, a logical flag with value .true. is kept in a new global structure each time a transient interpolation method is found to be "user3D". Otherwise the flag keeps its default value of .false. The new global structure is in the common module : psmile_smioc (like sga_smioc_transi)

```
! Global identification of User-defined Interpolation in transients
  TYPE(PSMILe_comp_udef), Dimension(:), POINTER :: sga_comp_udef_idx
```

In the loop on components, the sub-structure pointer : **sla_driver_udef**(:) is allocated

```
!   Allocate structure for user_defined Interp for each transient in component
    ALLOCATE ( sga_comp_udef_idx(ib_c)%sla_driver_udef(iga_comp_nb_transi(ib_c))
```

This substructure itself contains the arrays and logical variables necessary to keep track of all "user3D" interpolations decided by the user. We will see later on how these are counted with a more precise description of the driver routine:
prismdrv_init_smioc_struct

1.2.5) Total number of grids :

For each geographic transient, a grid name is provided in the XML file. For example on the target side (component ocean) we have the XML description :

```
<grid local_name="ocn_grid">
  <indexing_dimension index="1" periodic="true"/>
</grid>
```

And on the source (atmosphere) component side we have :

```
<grid local_name="atm_grid">
  <indexing_dimension index="1" periodic="true"/>
</grid>
```

If a user-defined interpolation is required for the geographic transients, then each side (source and target) will automatically setup a gridless grid on which the associated gridless function will be defined. This is completely transparent to the user : there is no declaration in XML SMIOC files, but the driver will generate a name for these gridless grids, and the grid internal structures will be allocated. From **prismdrv_init_smioc_struct,** a call to **prismdrv_get_all_grids** solves this question.
For this example, the gridless grids names will be : **ocn_grid_I_01** and **atm_grid_O_01**

It should be noted that a gridless grid is needed for each input channel input or each output channel on which a “User3D” interpolation has been defined. So for example if a transient, requiring a User3D interpolation, is received in one component and re-send to another one, there will be two associated user-defined transients for this component, and two associated gridless grids. One will be associated to an Input channel, and the other one to an Ouput channel. The names generated internally by the driver, will differ.

1.3) Constraints resulting from the PSMLe’s code structure

Most of the logic and structures of PSMILe remains unchanged. However a few more features are now necessary in order to process the user-defined interpolations, without changing the user’s interface.

1.3.1) No change in PSMILe User’s Interface. Provide the “Weights and Addresses “ file.

During the simulation runs, transients variables will be exchanged between the components models. When a "user defined" interpolation has been defined for such transients, PSMILe routines will send and receive the new "user defined" transient, based on a new "gridless" grid, in place of the geographic transient, based on the geographic grid. In the component's code, the PSMILe subroutines : prism_def_grid, prism_def_partition, … prism_def_var, etc, will be called ONLY for the geographic transient, as usual. But when a "user-defined" interpolation method is in demand, the user has to provide an additional "Weight and Addresses" file.

1.3.2) Relevant informations are given by the driver and by the W & A file

This logic is made possible if the PSMILe routines are now able to detect when the "user3D" method is in use for a specific transient input origin, or for a specific transient output. This is done under the call to subroutine : **Prism_enddef**.

1.3.3) Definition of user-defined transient is done under the call to Prism_enddef.

The logic of the subroutine Prism_enddef is now :
Loop on all fields allocated : 1 to Number_of_Fields_allocated

1. detect the presence of a "user3D" interpolation method
   a. in a specific transient input origin, or
   b. in a specific transient output

   The code loops on ALL inputs : Taskin%In_channel(:), if any, and on ALL outputs : Taskout(:), if any. In other words : on nbr_in and nbr_out, such that

   ```
   nbr_in = Fields(i)%Taskin%nbr_inchannels
   nbr_out = size (Fields(i)%Taskout
   ```

   These sub-structures are part of the geographic "transient" structure "Fields(field_id)"
2. read the NetCDF file associated with a specific Interpolation. This is done in the subroutine : **psmile_set_userdef**
3. define and declare the intermediate gridless exchange grid (array with one dimension = nlinks)
4. declare the associated variable ("user defined" transient)

5. For all "user defined" transients, the Prism_enddef now calls the following subroutines :
    a. **prism_def_grid**
    b. **prism_set_point_gridless**
    c. **psmile_store_data_intern_points :** This call is needed for the later call to psmile_merge_fields
    d. **prism_def_partition**
    e. **prism_set_mask**
    f. **prism_def_var**

6. it also disables the use of the geographic transient grid for the subsequent calls by setting the variable **Grids**(grid_id)**%used_for_coupling** to .false.

At this point, for each Field and for all Taskout or Taskin within these Fields, the identities of the associated User-defined transient and gridless grid are known:
For the Input channels :

Fields(i)%Taskin%In_channel(il_i)%assoc_var_id
Fields(i)%Taskin%In_channel(il_i)%userdef_id
Grids(Methods(Fields(i)%method_id)%grid_id)%assoc_grid_id

For the Output channels :

Fields(i)%Taskout(il_o)%assoc_var_id,
Fields(i)%Taskout(il_o)%userdef_id,
Grids(Methods(Fields(i)%method_id)%grid_id)%assoc_grid_id

Note that the gridless grid generated is generally multibloc in each PE partition. It may happen also that some of the PEs partitions are empty (zero point in such partition).

1.3.4) Prism_Put and Prism_get create a new data structure.

Under the Prism_put and the Prism_get, the "data_array" argument provided by the user is transformed into the "user defined" data array to be really exchanged by the inner subroutines. This process involves a new subroutine described below.

1.3.5) Bundles are possible also for user-defined transients

If the user has defined bundles on the geographic variable, these bundles are duplicated in the "user defined" transient variable, and transferred the usual way. The final data_array (output argument in Prism_get) contains the final geographic variable with all the bundles, distinguished by the index in the last dimension.

1.3.6) User's code is only concerned by the geographic grids and transients

Finally, from the user's point of view, the component's code usage of the PSMILe routines and arguments concerns ONLY the geographic transient, and the geographic grid. Everything else concerning the additional "user defined" transient(s) and gridless grid(s) is done by the PSMILe routines under Prism_enddef, Prism_put and Prism_get.

1.4) Overall view of the new driver logic for the SMIOC exploration

For all dynamic arrays (allocatables or pointers) there is the need for a "two pass" logic :

- A first pass in the XML counts the number of some elements

- Allocate the relevant dynamic arrays to the correct dimension for the second pass
- The second pass in XML fills the values of the elements and attributes into the dynamic arrays.

The global elements extracted from XML files are :

! Number of grids, transients, persistents and unit sets per component

```
INTEGER, DIMENSION(:), ALLOCATABLE  :: iga_comp_nb_grids
INTEGER, DIMENSION(:), ALLOCATABLE  :: iga_comp_nb_transi
INTEGER, DIMENSION(:), ALLOCATABLE  :: iga_comp_nb_persis
INTEGER, DIMENSION(:), ALLOCATABLE  :: iga_comp_nb_unitsets
INTEGER, DIMENSION(:), ALLOCATABLE  :: iga_comp_nb_udef
```

In the following, the transients and the grids elements are considered. There is strictly NO change for all other elements : number of persistents, number of unit sets..
For the total numbers of elements we define two distincts counters for grids and transients : one for the "XML only" elements, and one for the total "XML + User-defined" elements.

```
ig_nb_tot_unitsets   = 0
ig_nb_tot_grids      = 0
ig_nb_tot_xml_grids  = 0
ig_nb_tot_transi     = 0
ig_nb_tot_xml_transi = 0
ig_nb_tot_persis     = 0
```

Note the new global counter : **iga_comp_nb_udef**(:) .
It contains, per component, the number of new "user defined" transients to be created, and this number is also the number of "gridless" grids to be created.

The "two pass" logic in the driver SMIOC routines can now be exposed. This is the logical content of the routines : **prismdrv_get_udef_transients** and **prismdrv_init_smioc_struct**.

1.4.1 First pass logic : prismdrv_get_udef_transients

- First pass in XML "as before" : count "XML defined" items :
  This is done in the new subroutine : **prismdrv_get_udef_transients**.
  the functions of this subroutine are :

  DO on all components
  get the number of transients defined in XML, per component :
  CALL **get_smioc_grids_transi_nb** (iga_comp_id_doc_XML(ib_c), &
  iga_xml_comp_nb_grids(ib_c), &
  iga_comp_nb_transi(ib_c), &
  id_err)
  - accumulate the total number of XML transients and grids for all components
    ig_nb_tot_transi = ig_nb_tot_transi + iga_comp_nb_transi(ib_c)
    ig_nb_tot_xml_grids = ig_nb_tot_xml_grids + iga_xml_comp_nb_grids(ib_c)
  - ! Allocate a structure for the user_defined Interp for each transient in component
  - ALLOCATE (
    sga_comp_udef_idx(ib_c)%sla_driver_udef(iga_comp_nb_transi(ib_c)))

  ENDDO on components

  Allocate global structures

ALLOCATE (**sga_xml_smioc_transi (ig_nb_tot_transi)**

pointers are defined within the "'transient" structures, these 3 pointers need to be allocated with the correct dimensions : (global counters iga_)

ALLOCATE (**iga_comp_nb_stand_name(ig_nb_tot_transi)**)
ALLOCATE (**iga_comp_nb_transi_in(ig_nb_tot_transi)**)
ALLOCATE (**iga_comp_nb_transi_out(ig_nb_tot_transi)**)

DO on all components

local values for one component : ila_, and sla_
ALLOCATE (**ila**_comp_nb_stand_name(iga_comp_nb_transi(ib_c)))
ALLOCATE (**ila**_comp_nb_transi_in(iga_comp_nb_transi(ib_c)))
ALLOCATE (**ila**_comp_nb_transi_out(iga_comp_nb_transi(ib_c)))
! First pass : gather info in XML files only
ll_userdef_details = .true.
CALL **get_transi_io_numbers** (iga_comp_id_doc_XML(ib_c), &
iga_comp_nb_transi(ib_c), &
ila_comp_nb_stand_name(:), &
ila_comp_nb_transi_in(:), &
ila_comp_nb_transi_out(:), &
ib_c, &
ll_userdef_details, &
id_err)

Copy the local **ila_** counters into the **iga_** global counters

ENDDO on components

! 4.4. Allocate standard name, transient_out, and transient_in
! in a global transient structure. Loop on ALL transients in all components

DO ib_ntt = 1, ig_nb_tot_transi

ALLOCATE (**sga_xml_smioc_transi**(ib_ntt)%sg_transi_in%**sga_in_orig** &
(iga_comp_nb_transi_in(ib_ntt)))

ALLOCATE (**sga_xml_smioc_transi**(ib_ntt)%**sga_transi_out** &
(iga_comp_nb_transi_out(ib_ntt)))

ALLOCATE (**sga_xml_smioc_transi**(ib_ntt)%**cga_stand_name** &
(iga_comp_nb_stand_name(ib_ntt)))

ENDDO on all XML transients

Initialize **sga_xml_smioc_transi** global array of structures to PSMILE_undef with a call to **init_transi(…)**

DO on all components : index ib_c

! Allocate transient in and out structures for user-defined interpolations
DO ib_nt = 1, iga_comp_nb_transi(ib_c)
ALLOCATE **sga_comp_udef_idx** sub-structures
ENDDO

Initialise : sla_driver_transi, and sga_comp_udef_idx to PSMILE_undef
with a call to **init_transi(…)** and **init_comp_udef(…)**

ll_userdef_details = .true.
! extract detailed informations
CALL **get_transi_details** (iga_comp_id_doc_XML(ib_c), &
iga_comp_nb_transi(ib_c), &
sla_driver_transi(:), &
ib_c, &
ll_userdef_details, &
id_err)
In **sga_comp_udef_idx**, the flags for "USER3D" interpolations are : .true.

! 5.6. Put local transient details in global structure
sga_xml_smioc_transi (il_ntr+1:il_ntr+iga_comp_nb_transi(ib_c)) = &
sla_driver_transi(:)

Check coherency between transi_in and transi_out informations and
detect the transients "out" associated with User_Defined Interpolation
This algorithm compares cg_transi_in and out _name through 4 nested DO loops…

! Get the dimensioning numbers for "User Defined" transients :
by counting the flags : lga_trin_orig and lga_trout that were set in **get_transi_details**

! From now on we know : the number of "User Defined" transients to be created
per component = **iga_comp_nb_udef**(:)

! Allocate + fill iga_trans_udef(:) for each component it keeps the **indexes** of the
user defined transients in the XML SMIOC file, for each component.
ENDDO on components

! Reset global counters to 0 and keep present value in ig_nb_tot_xml_transi
ig_nb_tot_xml_transi = ig_nb_tot_transi
!
ig_nb_tot_transi = 0
iga_comp_nb_transi(:) = 0

ig_nb_tot_grids = 0
iga_comp_nb_grids(:) = 0

At the end of this "first pass" we have :
All dimensioning numbers for the XML transients and grids
All dimensioning numbers for the "User defined" transients
These numbers are equal to the number of grids “gridless” to be created, since there is one grid “gridless” associated to each “User defined” transient

⇨ We can add them together, and re-allocate new arrays of structures with the new global and local dimensions

1.4.2 Second pass logic : subroutine prismdrv_init_smioc_struct

Within this subroutine, the same logic is applied again, but this time with the new dimensions for all allocations : the flag ll_userdef _details is now set to .true.

A simplified vue of this routine is given below : we present here only a single DO loop on components, instead of the details of the do loops on components alternating with the global ordering of the transients

DO loop on components

```
ll_userdef_details = .true.
CALL get_smioc_numbers ( cla_file_name, il_length,       &
                         iga_comp_nb_unitsets(ib_c), &
                         iga_comp_nb_grids(ib_c),     &
                         iga_comp_nb_transi(ib_c),    &
                         iga_comp_nb_persis(ib_c),    &
                         ib_c,                        &
                         ll_userdef_details,              &
                         id_err )
```

iga_comp_nb_transi(ib_c) is now the **sum** of the XML transients, and the User defined transients, and :
iga_comp_nb_grids(ib_c) is now the **sum** of the XML grids and the User defined gridless grids.

These numbers are sent to the components, so the PSMILe routines can now allocate their own structures with the right dimensions on transients.
During the "first pass", informations on the dimensioning numbers or smioc details have NOT been sent to the components, since they were missing the values for the "user defined" transients.

For transients, the "io_numbers" now take into account the additional channels "In" or "Out" of the new user defined transients :

```
ll_first_details = .false.
CALL get_transi_io_numbers ( cla_file_name, il_length,   &
                             iga_comp_nb_transi(ib_c),   &
                             ila_comp_nb_stand_name(:),  &
                             ila_comp_nb_transi_in(:),   &
                             ila_comp_nb_transi_out(:),  &
                             ib_c,                   &
                             ll_first_details,           &
                             id_err )
```

! 5.5.1 Get details for all transients :

We do not need to execute the **get_transi_details** subroutine again. Instead, the "first pass" global array : **sga_xml_smioc_transi**, is used for initializing the local array : **sla_driver_transi(1:iga_xml_comp_nb_transi(ib_c))**.

Then the additional "User defined" transients for this component are created in the subroutine :

```
CALL prismdrv_get_all_transi ( iga_comp_nb_transi(ib_c),   &
                               sla_driver_transi(:),       &
                               ib_c,                   &
                               id_err )
```

On the return from this subroutine, the **sla_driver_transi** array of transient structures is now **complete** with :

1) the XML transients extracted from the SMIOC files during the first pass

and 2) the new, created, user defined transients needed for all "user3D" interpolations.

Within the loop on components, now all transient details are sent to each component where the PSMILe structures are filled. All "XML" transients and all "User defined" transients are then initialized in the PSMILe.

ENDDO on components

2) NEW "USERDEF" STRUCTURES

Several global and local structures have been created for the support of the automatic generation of the "user defined" transients.

2.1 PSMILe Common structures : lib/common_oa4/src/psmile_smioc.F90

```
!
! Structure for search of User-defined interpolations
! ======================================================
!  Allocate one per component
!
!  ig_xml_udef : number of XML transients of this component with  Udef  Interpolation
!
!  ig_tot_comp_ugl : total number of additional "gridless" transients for this component
!                depends on the number of origins or outputs of the XML "Udef" transients
!
!  iga_xml_trindex : dimension = ig_tot_comp_ugl, contain index of XML "Udef" transient
!                for each gridless transient in component
!
!  iga_trans_udef : dimension = ig_xml_udef, indexes of geographic transients "ud" in
component
!
!  sla_driver_udef  : array of structures : one per XML transient in component
!
  TYPE PSMILe_comp_udef

    Integer                  :: ig_xml_udef
    Integer                  :: ig_tot_comp_ugl
    Integer, pointer         :: iga_xml_trindex(:)
    Integer, pointer         :: iga_trans_udef(:)
    Type(PSMILe_udef_idx), pointer :: sla_driver_udef(:)

  END TYPE  PSMILe_comp_udef

!============================================================
!
!  Allocate one per XML transient
!
!  lg_trans_ud :  true if transient needs a User-defined Interpolation
!
!  ig_dim_orig : dimension of pointer lga_trin_orig, = ig_nb_in_orig
!
!  ig_dim_out  : dimension of pointer lga_trout,     = ig_nb_transi_out
!
!  lga_trin_orig : true if transient_in%origin needs a User-defined Interpolation
!                    Allocated to dimension ig_dim_orig = ig_nb_in_orig
!
!  lga_trout : true if transient output needs a User-defined Interpolation
!                Allocated to dimension : ig_dim_out = ig_nb_transi_out
!
!  cg_local_name : local name of transient found in XML.
!                     Used for generating the associated gridless grid name
```

TYPE PSMILe_udef_idx

```
    Logical           :: lg_trans_ud

    Integer           :: ig_dim_orig
    Integer           :: ig_dim_out
    Logical, pointer    :: lga_trin_orig(:)
    Logical, pointer    :: lga_trout(:)
    Character(len=max_name)  ::  cg_local_name
```

END TYPE PSMILe_udef_idx

```
!=============================================================
```

2.2 Global variables in psmile_smioc module lib/common_oa4/src/psmile_smioc.F90 (Code after the SAVE)

```
! Global counter for first pass
 INTEGER, DIMENSION(:), ALLOCATABLE  :: iga_xml_comp_nb_grids
 INTEGER, DIMENSION(:), ALLOCATABLE  :: iga_xml_comp_nb_transi

! Global pointer for first pass (search of User-defined Interpolations)
  TYPE(transient), DIMENSION(:), POINTER  :: sga_xml_smioc_transi

! Global identification of User-defined Interpolation in transients
  TYPE(PSMILe_comp_udef), Dimension(:), POINTER :: sga_comp_udef_idx
!
! Total number of grids in XML SMIOC files
  INTEGER :: ig_nb_tot_xml_grids
! Total number of transients in XML SMIOC files
  INTEGER :: ig_nb_tot_xml_transi

!=============================================================
```

2.3) New structures defined in lib/psmile_oa4/src/psmile.F90

```
!=============================================================
!
! Derived type Userdef to store info on user-defined links for interpolation
!
! var_id           : global index in the Fields array of the geographic variable
!
! igl_grid_id      : grid id of the gridless grid built from the user defined links
!
! status           : status of this entry
!
! ig_transi_side    : 0 for the source side (Tranient Out)
!                     1 for the target side (Transient In)
!
! ig_nb_links      : number of links defined in the weight and addresses file
!
! ig_celldim       : number of dimensions of geographical cells
```

```
!                 (can be 1, 2 or 3, and depend on side)
!
! ig_nb_ppp        : number of links relevant to the partition of this PE
!
! ig_nbr_fields    : number of "bundles" of geographic function (>= 1)
!
! lg_nolink        : .true. for a PE having no geographic point in its partition
! (obsolete...)          to match with any user-defined link
!
! iga_igl(:,:)     : list of indexes of geographical grid (local to partition)
!                 the list of indexes stored in iga_igl concerns only one side
!                 either the source side (for the prism_put)
!                 or the target side (for the prism_get)
!
! iga_links(:)     : list of links defined by the user (local to partition)
!                 the list stored in iga_links concern only one side :
!                 either the source side, or the target side
!
! trans_grless(:,:,:,:) : Local non continuous gridless function based on
!                   the geographic function and the links in w&a file
!
```

Type Userdef

```
      Integer :: var_id
      Integer :: igl_grid_id
      Integer :: status
      Integer :: ig_transi_side
      Integer :: ig_nb_links
      Integer :: ig_celldim
      Integer :: ig_nb_ppp
      Integer :: ig_nbr_fields

      Integer, pointer               :: iga_igl(:,:)
      Real (PSMILe_float_kind), pointer :: dga_wght(:)
!
      REAL, pointer   :: real_gridless(:,:,:,:)
      DOUBLE PRECISION, pointer   :: dble_gridless(:,:,:,:)
```

End Type Userdef

```
!=================================================================
!
! Derived type PSMILe_Link  to store info about each link defined by user
!
! cell_id(3)       : indexes of the geographical cell in the 3 dimensions
!
! weight           : weight attached to the geographical field value
```

Type PSMILe_Link

```
      Integer :: cell_id(3)
      Real (PSMILe_float_kind) :: weight
```

End Type PSMILe_Link

```
!=============================================================
```

2.4) Modified existing structures

PSMILe structures are mostly left unchanged. However the association of a "geographic transient" with a "user defined" transient, in case of an interpolation method "user3D" introduced a few variables in the existing structures : Grids, Taskin and Taskout

2.4.1 Grid type

assoc_grid_id is the grid ID of the associated gridless grid, needed for the user defined transient

Type Grid

```
Integer                 :: global_grid_id
Integer                 :: status
Integer                 :: comp_id
Integer                 :: grid_type
Integer                 :: grid_shape (2, ndim_3d)
Integer                 :: grid_structure
Integer                 :: n_dim
Integer (kind=int64)        :: size
Integer (kind=int64)        :: global_size(ndim_3d)
Logical                 :: used_for_coupling
Logical                 :: pole_covered
Integer                 :: smioc_index
Character(len=max_name)     :: grid_name
Type (Corner_Block), Pointer  :: corner_pointer

Type (Enddef_mg), Pointer    :: mg_infos (:)
Integer                 :: nlev
Integer                 :: ijk0 (ndim_3d)

Integer                 :: periodic(ndim_3d)
Logical                 :: cyclic(ndim_3d)
Integer                 :: len_periodic (ndim_3d)

Integer                 :: nbr_halo_segments
Type (Halo_info), Pointer    :: halo (:)
!
! Specific for Gauss-reduced grids
!
Integer                 :: nbr_latitudes
Integer, Pointer            :: nbr_points_per_lat(:)
Integer, Pointer            :: partition(:,:)
Integer, Pointer            :: extent(:,:)
Type (Corner_Block), Pointer  :: gcorner_pointer
Integer, Pointer            :: star(:,:)
Integer, Pointer            :: face(:,:)
Integer, Pointer            :: global_beg(:)
Integer, Pointer            :: global_end(:)
```

```
      Integer, Pointer            :: g2l(:,:)
      Integer, Pointer            :: l2g(:,:)
      Integer, Pointer            :: g_irange(:,:)

      Integer, Pointer            :: remote_index(:)
      Integer, Pointer            :: send_list(:)
      Integer, Pointer            :: recv_list(:)
      Type(integer_vector), Pointer :: get_list(:)
      Type(integer_vector), Pointer :: put_list(:)
      Type(integer_vector), Pointer :: put_loc_list(:)
      Integer                     :: ijk0_r (ndim_3d)
      !
      ! Userdef interpolation : associated grid_id
      !
      Integer                     :: assoc_grid_id
```

End Type Grid

2.4.2 Taskout

assoc_var_id is the ID of the user defined transient variable associated with the actual geographic variable.
userdef_id is the ID pointing to the "Userdefs" global array entry for the "user defined" transient
The structure Userdefs contains for example the local values of the transient variable, computed in Prism_put, or received in Prism_get

Type Taskout_type

```
      Integer               :: origin_type
      Integer               :: remote_transi_id
      Integer               :: global_transi_id
      Integer               :: remote_comp_id
!
!   Userdef  ids (if interpolation is "user3D")
      Integer               :: assoc_var_id
      Integer               :: userdef_id

      Double Precision      :: start_day
      Double Precision      :: end_day
      Double Precision      :: start_sec
      Double Precision      :: end_sec
      Integer               :: nsum
      Integer               :: Time_length
      Integer, Pointer      :: buffer_int(:)
      Real, Pointer         :: buffer_real(:)
      Double Precision, Pointer :: buffer_dble(:)
#if defined ( PRISM_QUAD_TYPE )
      Real (kind=PRISM_QUAD_TYPE), Pointer :: buffer_quad(:)
#endif
      Type (PSMILe_Time_Struct)          :: Judate_Lbnd
      Type (PSMILe_Time_Struct)          :: Judate_Ubnd
      Type (PSMILe_Time_Struct), Pointer :: Judate_Axis(:)
```

```
Type (Interp_type)      :: interp
Type (Combi_type)       :: combi

Integer                 :: n_send_direct
Integer                 :: n_send_coupler
Integer                 :: n_send_appl

Integer                 :: n_alloc_send_direct
Integer                 :: n_alloc_send_coupler
Integer                 :: n_alloc_send_appl

Type (Send_field_information), Pointer :: send_direct  (:)
Type (Send_field_information), Pointer :: send_coupler (:)
Type (Send_field_information), Pointer :: send_appl (:)
```

End Type Taskout_type

2.4.3 ch_ptr type part of : Taskin% In_channel(:)

Same variables as in Taskout structure.

Type ch_ptr

```
Integer                 :: origin_type
Integer                 :: remote_transi_id
Integer                 :: global_transi_id
Integer                 :: remote_comp_id

!   if User defined interpolation
    Integer             :: assoc_var_id
    Integer             :: userdef_id
!
Type (Interp_type)      :: interp
Type (Combi_type)       :: combi
```

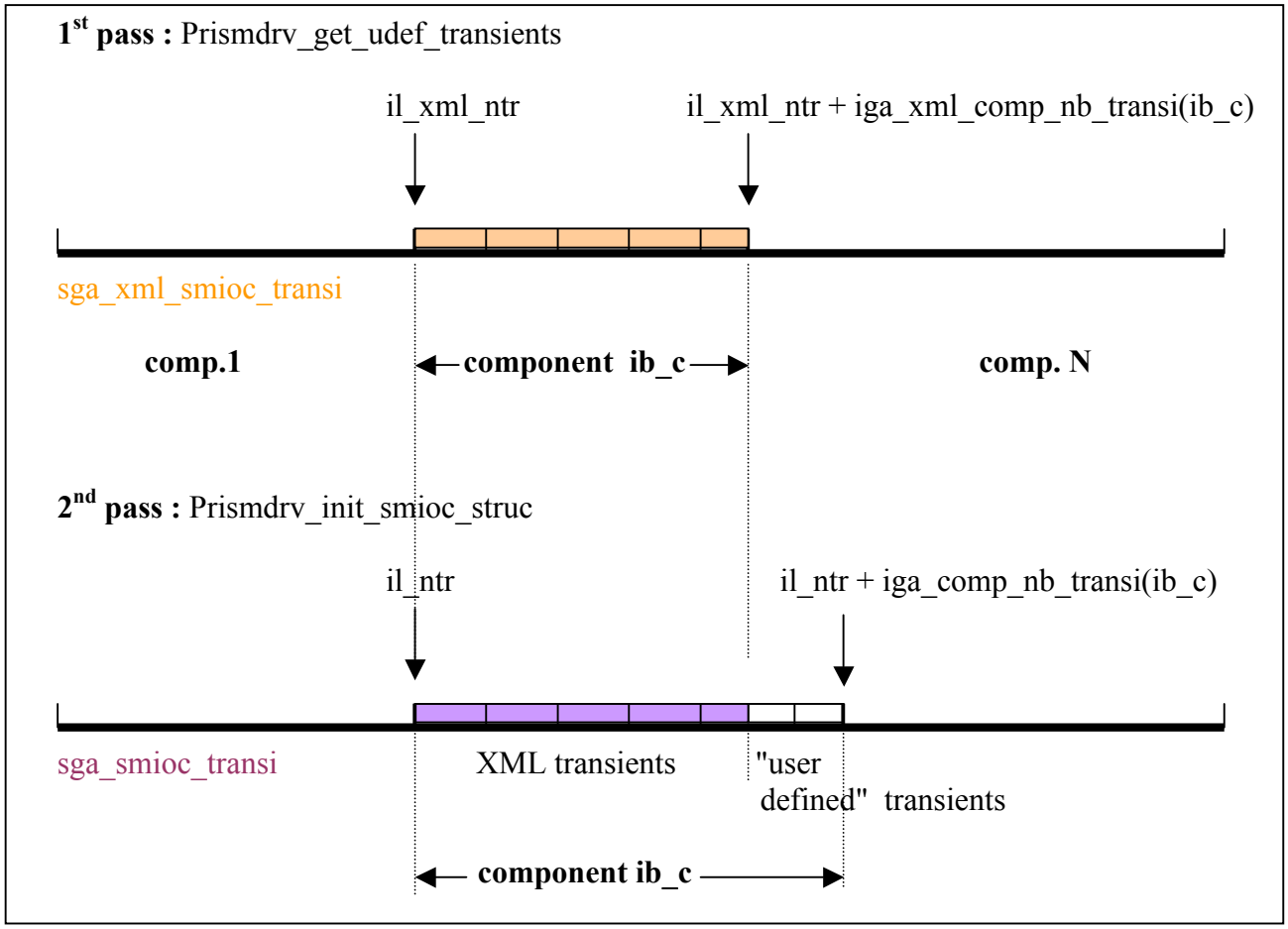
End Type ch_ptr

3) DRIVER'S VARIABLES DETAILS

3.1) differences between the first and second steps in the SMIOC files processing

In the second step the information gathered in the global array **: sga_xml_smioc_transi,** is just copied into the new global array : **sga_smioc transi,** in order to avoid a second scan of the XML files with the "sasa" routines. This process can be time consuming, mainly due to the fact that with "user3D" interpolations, one must explore the XML hierachical structures up to the 10th level. A full exploration has been done in the subroutine Prismdrv_get_udef_transients, so it can be avoided in the second step carried by subroutine Prismdrv_init_smioc_struct.

The ordering of the transients structures in **sga_xml_smioc_transi** and **sga_smioc_transi** is easy to understand with the follwing picture :

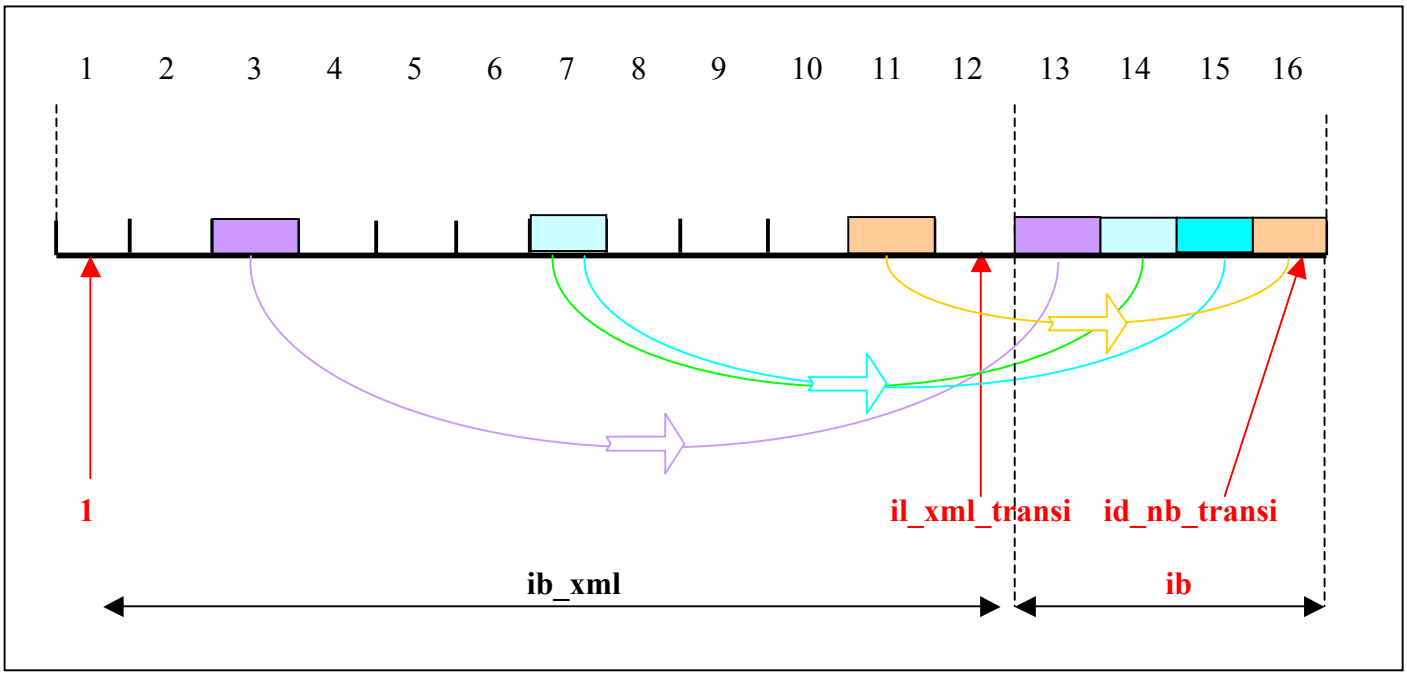


3.2) Indexing of the transients within a component

The numbering of transients is in fact the do loop index "**ib_c"** in **get_transi_details.**
The interpolation method is specific to a single output of the transient "source" or a single origin of the transient target. Indexing of these structures is the loop index ib_o for the outputs and ib_i for the input-origins.
The dimension of the Taskout(:) pointer is not a variable in the GridFunction type, so we called nbr_out = size("Taskout(:)") .
For input origins, the size of the pointer In_channels(:) is the Taskin variable : Taskin%nbr_inchannels

3.3) Creation of the additional "user defined" transients for the auxiliary function

In subroutine **create_all_transi** we loop on the XML transients in which a "user3D" method has been detected. For these transients, one or more new "user-defined" transient must be created : this process is a copy of the XML "transient" structure, of index : "ib_xml" into a transient structure whose index : "ib" varies from : il_xml_transi+1 to iga_comp_nb_transi(ib_c).
For a given component, the array of transient structures : **sla_driver_transi** has been allocated, to dimension : **iga_comp_nb_transi(ib_c)**.
The logic of the transient copy is in the subroutine create_all_transi : with an example it is easy to understand :
Transients for the component "ib_c" for example has 12 "XML" transients (in SMIOC file)
Three of them (3, 7 and 11) rely on the interpolation method "user3D".
Transient 3 has one output, transient 7 has 2 outputs, and transient 11 has 1 input channel



The copy of XML transients structures into "user defined" transient structures is in the loop :
DO ib_xud = 1, sga_comp_udef_idx(id_comp)%ig_xml_udef

In the above example **ig_xml_udef = 3**

The index of the transients to be copied (3,7,11) is in : iga_trans_udef(:) array. Each index is copied in the variable **ib_xml = sga_comp_udef_idx(id_comp)%iga_trans_udef(ib_xud)**.

Now we must check which output, and which input channel uses an interpolation method "user3D". This information is kept in the logical flag associated with each of these channels for this transient :
for input channels : if this flag is true, we create a copy :
sga_comp_udef_idx(id_comp)%sla_driver_udef(ib_xml)%lga_trin_orig(il_ch)

all possible values of il_ch are scanned through the loop on the values : 1 to
il_dim_i = sga_comp_udef_idx(id_comp)%sla_driver_udef(ib_xml)%ig_dim_orig

Similarly, for the output channels, we execute the loop : il_ch = 1 to :
il_dim_o = sga_comp_udef_idx(id_comp)%sla_driver_udef(ib_xml)%ig_dim_out

and we check the flag :
sga_comp_udef_idx(id_comp)%sla_driver_udef(ib_xml)%lga_trout(il_ch)

The last point is to give an index to the new "transient" (user-defined) structure within the array sla_driver_transi(iga_comp_nb_transi(ib_c)). For this we have defined the index "ib" .
The first argument in the call to "create_all_transi" is the total number

id_nb_transi = iga_comp_nb_transi(ib_c)

In addition to this information, the global variable : iga_comp_nb_udef(id_comp) contains the total number of user-defined transients to be created for this component, so we compute the upper value of the index of the XML transients already presents in the sla_driver_transi array :

il_xml_transi = id_nb_transi - iga_comp_nb_udef(id_comp)

Note that the index of the component is ib_c in the calling routine, and id_comp in the callee.

For the new transients : ib minimum value is then il_xml_transi + 1, and the maximum value is id_nb_transi. "ib" is then incremented by 1 each time a new transient has to be copied / created.

The copy of the transient structure is done in subroutine : "create_transi_udef.F90" in the common routines subdirectory. During this copy, several elements are NOT copied like the most of the "id" variables, or the information about the "user3D" interpolation, since there is no interpolation for this new transient, only a direct transfer from source component to the target component.

Names need a special treatment : on the input channel side, the variables :
sga_in_orig(ib_i)%**cg_transi_in_name** and sga_in_orig(ib_i)%**cg_orig_transi**
are modified with the addition of the suffix "**_glC**"
And on the output side, the variables :
sga_transi_out(ib_o)%**cg_transi_out_name** and sga_transi_out(ib_o)%**cg_dest_transi**
are modified in the same way
By adding to theses name a constant suffix, (the same suffix will be applied to ALL of these "channel" names for ALL transients created, in ALL components), we do not modify the result of the matching algorithm done subsequently by the driver in order to find the associations between "outputs" and "origins". Theses suffixes are needed since this association must be done between the new user-defined transients ONLY. If the names were left unchanged, we could mix channels of the geographic (XML) transients with those of the used-defined transients.

On the other hand, we have also to modify the "**cg_local_name**" and the
"**cg_grid_family_name**"
in order to distinguish them from those of the intial (XML) transient. But this time we must create a UNIQUE name within the global application (all components), since we are creating a specific transient for each output or each input channel of a given transient. The suffix now depends also on the "side" : I for input channel, O for output channel, and from the index of this channel (on 2 digits). For example :

XML transient local name : "**source_fnc**", output channel 1, will have for associated user defined transient : local name = "**source_fnc_glO_01**"

Similarly, XML transient local_name : "**target_fnc**" has for associated user defined transient : local name = "**target_fnc_glI_01**"

The cg_grid_family_name is also transformed with this "variable" suffix. A special subroutine is called for this job : put_udef_suffix.F90, in the common subdirectory. This subroutine is also called on the PSMILe side under Prism_enddef for the association between the grid_name and the transient name.

3.4) De-allocation of the global structures

Global arrays **sga_smioc_transi** and **sga_xml_smioc_transi** must be kept in memory up to the end of the subroutine **Prismdrv_set_smioc_info**, where the driver computes the total number of Communications, total number of Interpolations, and total number of Transformations.

Both structures are linked by the = sign that has a special significance for the pointers inside these structures : **cga_stand_names**, **sga_transi_out** and **sga_in_orig**.

De-allocation is made by subroutine **Prismdrv_finalize_smioc_struct**, called at end of **Prismdrv_set_smioc_info**

4) PSMILE'S VARIABLES DETAILS

4.1) Prism_init modifications

A new global array of structures is introduced with the "user defined interpolation" : the **Userdefs**, at the same level as the Grids, Fields, etc…
In Prism_init the same allocation scheme is in use :

```
!===> Pre-allocate Userdef structures
!
    Number_of_Userdefs_allocated = 8

    Allocate (Userdefs(Number_of_Userdefs_allocated), STAT = ierror)

    Userdefs(:)%ig_transi_side = PRISM_Undefined
    Userdefs(:)%ig_nb_links = 0
    Userdefs(:)%status    = PSMILe_status_free

    do i = 1, Number_of_Userdefs_allocated
      Nullify ( Userdefs(i)%dga_wght )
      Nullify ( Userdefs(i)%iga_igl )
      Nullify ( Userdefs(i)%real_gridless )
      Nullify ( Userdefs(i)%dble_gridless )
    enddo
```

Subsequent allocations of more structures, if needed, will be done by a call to : **psmile_get_userdef_handle**. This subroutine returns a "userdef_id" = index of the structure in the array "**Userdefs**", and eventually, extends this array by a copy in a new, larger array and a de-allocation of the old array.

4.2) Prism_enddef logic

In this routine we follow exactly the same logic as in the driver routine : create_all_transi.
The driver has sent to each component all the information contained in the SMIOC files (and in the user-defined transients created). So by searching the Field(field_id)%Taskout(:), and Taskin%In_channels(:) arrays, we will find exactly the same number of "user3D" interpolations in the geographic XML "gridfunctions". At this point, the user has only defined the gridfunctions for the geographic variables in the source and target component's codes.
By exploring all input channels, and all outputs we will find the cases where a "user3D" interpolation is in use. For these we will then build the gridless grid, and define its associated variable. This work is done in the PSMILe subroutine : **psmile_set_userdef**.

The test done by Prism_enddef on the existence of a matching prism_def_var call for every SMIOC field name will be successful for the new "user defined" variable, since its name has been built in the "user defined" transient with the same syntax and the same additional suffix.

An important point is to associate the geographic variable (set by the user) with a geographic grid, to the "hidden" user-defined variable and its gridless grid. Morover, only the gridless grid should play a role in the search for intersections that follows ! This is why we have the following sequence at the end of **psmile_set_userdef** (fp points to the geographic field, and gp to the geographic grid):

```
!   4. Updates geographical grid and field structures with associated userdef values
!
```

```
      gp%assoc_grid_id = grid_id_2
      gp%used_for_coupling = .false.
!
      if  ( il_side == 0 ) then
!     Geographic transient
        fp%Taskout(chan_id)%assoc_var_id  = ass_var_id_2
        fp%Taskout(chan_id)%userdef_id    = userdef_id_2
      elseif ( il_side == 1 ) then
!     Geographic transient
        fp%Taskin%In_channel(chan_id)%assoc_var_id  = ass_var_id_2
        fp%Taskin%In_channel(chan_id)%userdef_id    = userdef_id_2
      endif
```

In case the gridless grid partition of a PE has no point in it (empty partition), this means that the links defined in the weight and addresses file do not concern the geographic grid partition for this PE. In such case, this PE must be excluded from coupling for this field. This is important for the subsequent search of intersections. In this case we have the logic :

```
! Case where there is no link for this PE :
  IF (il_nb_ppp .EQ. 0) THEN
      PRINT *, " Warning : No gridless point for this PE"
      CALL PSMILe_Flushstd
      gp%used_for_coupling = .false.
      fp%used_for_coupling = .false.
      return
  ENDIF
```

4.3) Prism_put logic

The items 0, 1, 2, 3, and 4 have not been changed compared to previous versions.
At item 5 we introduce the changes for the "user defined" interpolations.
Since prism_put deals only with outputs, we have to count them and detect which ones have "user3D" interpolations to do.

```
     fp => Fields(field_id)

!  count all Output channels
     nbr_out = 0
     if ( Associated(fp%Taskout) ) then
       nbr_out = size (fp%Taskout)
     endif
```

Then we allocate a new local integer arrays (2D) in order to keep track of the actions to be performed on all output channels :

```
!  Allocate array ila_ch_act(nbr_out,4)
!
     Allocate ( ila_ch_act(nbr_out,4), STAT=ierror )

!  Loop on all output channnels
!
     do il_o = 1, nbr_out
```

```
      ila_ch_act(il_o,1) = field_id                ! field_id used for put
      ila_ch_act(il_o,2) = il_o                    ! channel output of field_id
      ila_ch_act(il_o,3) = PSMILe_false            ! "flag" for this il_o channel
!
      il_udef_id = fp%Taskout(il_o)%userdef_id
      ila_ch_act(il_o,4) = il_udef_id              ! associated userdef id
```

The userdef_id is necessary to refer to the Userdefs structure containing all necessary informations about "user3D" interpolations. The test consists in finding, or not, an active userdef_id ; if the value is PSMILe_undef, the channel of this field does NOT uses "user3D" interpolation, and the values in ila_ch_act will remain unchanged.
Otherwise if have to set the new values in ila_ch_act that will switch the subsequent treatments to the "user-defined" variable and its gridless grid.

```
      if ( il_udef_id /= PSMILe_undef ) then
!     gridless function will be used for the put
        ug => Userdefs(il_udef_id)
        il_side = ug%ig_transi_side
        il_dim1 = size ( fp%var_shape(:,:), dim=1 )
        il_dim2 = size ( fp%var_shape(:,:), dim=2 )
        field_id_2 = fp%Taskout(1)%assoc_var_id
!
        ila_ch_act(il_o,1) = field_id_2              ! field_id_2 is used for put
        ila_ch_act(il_o,2) = 1                       ! channel output of field_id_2
        ila_ch_act(il_o,3) = PSMILe_true             ! "flag" for this il_o channerl
        ila_ch_act(il_o,4) = il_udef_id              ! associated userdef id
```

next we check the existence of bundles and the length of the data to be transferred :

```
!       Check length of data
!
        il_fsize = Fields(field_id_2)%size   ! computed from actual_shape_pr
        il_gsppp = ug%ig_nb_ppp
        il_nbfld = ug%ig_nbr_fields          ! defined for prism_def_var
! Size of a single field (ig_nbr_fields is 1 or nb_bundles)
        il_size1 = il_fsize / il_nbfld     ! In case of bundle : size of 1 field
```

We must now allocate the space for the grid function and calculate its values (local to this partition), before sending them. This implies the data provided by the user "data_array" and the informations kept for each cell in the Userdefs structure (obtained from the W&A file)
This work is done by two new routines : **psmile_gridless_func_real**, for real data, or **psmile_gridless_func_dble** for double precision data.

```
!       build the gridless function according to data type and dimensions

        if ( fp%dataType == PRISM_Real ) then
          call psmile_gridless_func_real ( field_id, il_udef_id, il_side, &
                                data_array, ierror )
        else if ( fp%dataType == PRISM_Double_Precision ) then
          call psmile_gridless_func_dble ( field_id, il_udef_id, il_side, &
                                data_array, ierror)
        endif
```

We are now ready to loop over the output channels : Note that **il_omax** is identical to **nbr_out**

il_omax = sga_smioc_transi(Fields(field_id)%smioc_loc)%**ig_nb_transi_out**

```
do il_o = 1, il_omaxi
!
!  keep original geographical field  (only if it DOES NOT uses "user3D interpolation)
!  or substitute the gridless (udef) field  : the "field_id" = "field_id_2"
       field_id   = ila_ch_act(il_o,1)          ! field_id really used for put
       i          = ila_ch_act(il_o,2)          ! real channel output of field_id
       il_userdef = ila_ch_act(il_o,3)          ! "flag" for this il_o channel
       il_udef_id = ila_ch_act(il_o,4)          ! (optional) associated userdef id

 …..
enddo
```

The important point is that the **field_id** in this loop is referring EITHER to a geographic variable, or to a "user defined" variable, depending on the user's choice on interpolation.

Depending on the test on **il_userdef**, the geographic variable will be sent, or the user-defined variable. This syntax is necessary, since the Fortran variable name is different, depending on the case.

At the end of Prism_put, we restore the meaning of field_id to the geographic variable :

```
!  Restore input variable field_id as the geographical field ID

    Nullify (fp)
    field_id = field_id_1
```

The nice part is that below the level of "**psmile_put_real**", etc… nothing needs to be changed in the PSMILe code.

All arrays allocated for the special case of "user3D" interpolations are de-allocated before leaving Prism_Put : ug%real_gridless, ug%dble_gridless and ila_ch_act.

4.4) Prism_get logic

This is essentially the same as the Prism_put logic. The obvious change is that we are now dealing with **input** channels. Items 0 to 4 are unchanged, and before item 5 we have to define nbr_in :

```
    field_id_1 = field_id
    fp => Fields(field_id)

!  Future loop on In_channels
    nbr_in = fp%Taskin%nbr_inchannels
```

Another change is that currently the PSMILe code supports only ONE input channel, so there is no need for a loop. Instead we have :

```
!    do il_i = 1, nbr_in
!      il_udef_id = fp%Taskin%In_channel(il_i)%userdef_id
```

```
!  Current state of the code ... only one channel here

      il_udef_id = fp%Taskin%In_channel(1)%userdef_id
      if ( il_udef_id /= PSMILe_undef ) then
!
!       data_array received will contain the gridless function
        ug => Userdefs(il_udef_id)
        ll_userdef = .true.
        il_side = ug%ig_transi_side
        il_dim1 = size ( fp%var_shape(:,:), dim=1 )
        il_dim2 = size ( fp%var_shape(:,:), dim=2 )
        field_id_2 = fp%Taskin%In_channel(1)%assoc_var_id
```

The rest of the code follows the same logic as in Prism_put, without the necessity to keep the channel number.

Note that with the new version of the PSMILe and driver's codes, it is still possible that a component may receive AND re-send the same transient with "userd3D" interpolations.
This will generates TWO different "user defined" transients, that will be substituted to the geographic variables so the exchanges will take place in the same way. (still need to be tested on a realistic case)

The occurrence of bundles in the geographic variable has been tested successfully. In this case all bundles are concerned by the same W&A file, and the same geographic grid.

Before getting the gridless function we allocate the array that will receive it :

```
        if ( Fields(field_id)%dataType == PRISM_Real ) then
          Allocate ( ug%real_gridless(1:il_gsppp,1,1,il_nbfld), STAT=ierror )
        elseif ( Fields(field_id)%dataType == PRISM_Double_Precision ) then
          Allocate ( ug%dble_gridless(1:il_gsppp,1,1,il_nbfld), STAT=ierror )
        endif
!
!    user-defined interpolation : substitute gridless function and gridless grid
!    From now on : use the gridless function and the gridless grid in place of
!                        the geographical function and geographical grid
!
        Nullify (fp)
        field_id = field_id_2
```

The reception of the gridless function is then followed by a call to **psmile_gridless_func_real**, or **psmile_gridless_func_dble**. These routines compute the value of the geographic function according to the links defined by the user in the weights an addresses file.
For example for real values :

```
      if ( ll_userdef ) then
!  1.    Get the gridless function in gridless structures
!  2.    Restore the geographical field in data_array

        if ( Fields(field_id)%dataType == PRISM_Real ) then
          call psmile_get_real ( field_id, julian_day, julian_sec,  &
              julian_dayb, julian_secb, ug%real_gridless, action(1), action(2), &
              info, ierror )
```

```
        call psmile_gridless_func_real ( field_id_1, il_udef_id, il_side, &
                                         data_array, ierror )
      endif

else
    …. “normal case” where the variable “data_array” receives directly the geographical
                     function.

endif
```

At the end of Prism_get, we restore the meaning of field_id to the geographic variable :

```
!  Restore input variable field_id as the geographical field ID

    Nullify (fp)
    field_id = field_id_1
```

4.5) Psmile_gridless_func_real (or dble) logic

In this new subroutine, we compute the gridless function from the geographic function values and the informations given in the weight and addresses file, if we are on the source side (Prism_put), and conversely, we restore the geographic function from the gridless function values and the links defined in the weights and addresses file if we are on the target side (Prism_get).

Note that the geographic function is defined as data_array(*). In other words, the shape information of data_array is lost: we have to deal with a one-dimensional set of numbers. The dimensions are recovered from the var_shape array which is a subarray in the Field datatype.

```
      dim1 = fp%var_shape(2,1) - fp%var_shape(1,1) + 1
      dim2 = fp%var_shape(2,2) - fp%var_shape(1,2) + 1
    if ( ndim == 3 ) then
      dim3 = fp%var_shape(2,3) - fp%var_shape(1,3) + 1
```

The the logic of source and target sides are similar : we use indirect addressing contained in the user-defined “ug” structures in order to compute the value of the gridless function (on Prism_put side), or the value of the geographical function (on Prism_get side).

5) CONCLUSIONS

The OASIS4 coupler has been adapted so to be able to remap a coupling field provided on a source model geographic grid to a target model geographic grid according to a set of weights and addresses predefined by the user in an external file. This functionality has been successfully tested with a toy coupled model “user3d-auto” which sources are available at https://oasistrac.cerfacs.fr/browser/trunk/prism/dev_ex/user3d-auto .
This “user-defined” remapping uses the “gridless” remapping already implemented in OASIS4, which involves a simple redistribution of source data to the target model. All steps are totally transparent for the user who simply has to provide the weight-and-address file and to specify a “user3D” interpolation in the SMIOC configuration files.