PRISM
Project for Integrated Earth System Modelling
An Infrastructure Project for Climate Research in Europe
funded by the European Commission
under Contract EVR1-CT2001-40012

OASIS4 User Guide

Edited by:
S. Valcke, CERFACS
R. Redler, NEC-CCRLE
R. Vogelsang, SGI Germany
D. Declat, CERFACS
H. Ritzdorf, NEC-CCRLE
T. Schoenemeyer, NEC HPCE

PRISM–Report No 3

1st Edition

December 15th, 2004

Copyright Notice

© Copyright 2004 by CERFACS, NEC-CCRLE, SGI Germany, NEC HPCE
All rights reserved.
No parts of this document should be either reproduced or commercially used without prior agreement by CERFACS, NEC-CCRLE, SGI Germany, or NEC HPCE representatives.

How to get assistance?

Assistence can be obtained as listed below.
PRISM documentations can be downloaded from the WWW PRISM web site under the URL: <http://prism.enes.org>

Phone Numbers and Electronic Mail Adresses

Name	Phone	Affiliation	e-mail
Sophie Valcke	+33-5-61-19-30-76	CERFACS	
René Redler	+49-2241-92-52-40	NEC-CCRLE	

Contents

1 Introduction **1**

2 OASIS4 sources **3**
2.1 Copyright Notice . . . 3
2.2 Reference . . . 3
2.3 How to obtain OASIS4 sources . . . 3
2.4 OASIS4 directory structure . . . 4

3 OASIS4 Driver and Transformer **5**
3.1 The Driver . . . 5
3.2 The Transformer . . . 6

4 OASIS4 Model Interface library (PSMILe) **7**
4.1 Initialisation phase . . . 8
4.1.1 prism_init . . . 8
4.1.2 prism_init_comp . . . 8
4.1.3 prism_get_localcomn . . . 9
4.1.4 prism_initialized . . . 9
4.2 Retrieval of SCC XML information . . . 10
4.2.1 prism_get_nb_ranklists . . . 10
4.2.2 prism_get_ranklists . . . 10
4.3 Grids and related quantities definition . . . 11
4.3.1 prism_def_grid . . . 12
4.3.2 prism_set_corners . . . 14
4.3.3 prism_set_scalefactor . . . 15
4.3.4 prism_set_mask . . . 16
4.3.5 prism_set_vectormask . . . 16
4.3.6 prism_set_subgrid . . . 17
4.3.7 prism_def_partition . . . 18
4.3.8 prism_set_points . . . 19
4.3.9 prism_set_vector . . . 20
4.3.10 prism_set_angle . . . 20
4.4 Declaration of Coupling/IO fields . . . 21
4.4.1 prism_def_var . . . 21
4.5 Neighborhood search and determination of communication patterns . . . 23
4.5.1 prism_enddef . . . 23
4.6 Exchange of coupling and I/O fields . . . 24
4.6.1 prism_put . . . 25
4.6.2 prism_get . . . 26
4.6.3 prism_put_inquire . . . 26
4.6.4 prism_put_restart . . . 27

4.7 Termination Phase . . . 28
4.7.1 prism_terminate . . . 28
4.7.2 prism_terminated . . . 28
4.7.3 prism_abort . . . 28
4.8 Query and Info Routines . . . 29
4.8.1 prism_get_calendar_type . . . 29
4.8.2 prism_calc_newdate . . . 29
4.8.3 prism_error . . . 29
4.8.4 prism_version . . . 30
4.8.5 prism_get_real_kind_type . . . 30
4.8.6 prism_remove_mask . . . 30

5 OASIS4 description and configuration XML files 31
5.1 Introduction to XML concepts . . . 32
5.2 The Application Description (AD) . . . 33
5.3 The Specific Coupling Configuration (SCC) . . . 33
5.4 The Potential Model Input and Output Description (PMIOD) and the Specific Model Input and Output Configuration (SMIOC) . . . 36
5.4.1 Component model general characteristics . . . 36
5.4.2 Grid families and grids . . . 37
5.4.3 Coupling/IO fields (transient variables) . . . 38
5.4.4 The 'output' element . . . 40
5.4.5 The 'input' element . . . 42
5.4.6 The element 'interpolation' . . . 43
5.4.7 The 'file' element . . . 45
5.4.8 *Persistent variables* . . . 45
5.4.9 *Dependency* . . . 45

6 Compiling and Running with OASIS4 47
6.1 Introduction . . . 47
6.2 Compiling OASIS4 and its associated PSMIle library . . . 47
6.3 Compiling OASIS4 example toy coupled models . . . 48
6.4 Running an example toy coupled model with OASIS4 . . . 48
6.5 Remarks and known problems . . . 49

7 Scalability with OASIS4 51

A Toy coupled models with OASIS4 55
A.1 General description . . . 55
A.2 The proto_ex toy coupled model . . . 56
A.2.1 The proto_ex toy coupled model general description . . . 56
A.2.2 The proto_ex toyatm AD XML file . . . 58
A.2.3 The proto_ex toylan AD XML file . . . 58
A.2.4 The proto_ex toyoce AD XML file . . . 58
A.2.5 The proto_ex toy coupled model SCC XML file . . . 58
A.2.6 The proto_ex toyatm PMIOD XML file . . . 58
A.2.7 The proto_ex toylan PMIOD XML file . . . 58
A.2.8 The proto_ex toyoce PMIOD XML file . . . 59
A.2.9 The proto_ex toyatm SMIOC XML file . . . 59
A.2.10 The proto_ex toylan SMIOC XML file . . . 59
A.2.11 The proto_ex toyoce SMIOC XML file . . . 59

B DTDs and Schemas of OASIS4 description and configuration files **61**
B.1 AD DTD 61
B.2 SCC DTD 61
B.3 PMIOD DTD and Schema 61
B.3.1 PMIOD DTD 61
B.3.2 PMIOD Schema 61
B.4 SMIOC DTD and Schema 61
B.4.1 SMIOC DTD 61
B.4.2 SMIOC Schema 61

Index **65**

Chapter 1

Introduction

A new fully parallel coupler for Earth System Models (ESMs), OASIS4, has been developed within the European PRISM project. Chapter 2 provides a more detailed description of OASIS4 sources and how to get them from the PRISM CVS server (7).

An ESM coupled by OASIS4 consist of different applications (or executables), which executions are controlled by OASIS4. Each ESM application may host only one or more than one climate component models (e.g. model of the ocean, sea-ice, atmosphere, etc.). To interact with the rest of the ESM at run-time, the component models have to include specific calls to the OASIS4 PRISM System Model Interface Library (`PSMILe`). Each application and component model must be provided with XML files (11) that describe its coupling interface established through `PSMILe` calls. The configuration of one particular coupled 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 done by the user also through XML files.

During the run, OASIS4 Driver's role is first to extract the configuration information defined by the user in the XML files and then to organize the process management of the coupled simulation. OASIS4 Transformer's role is to perform the regridding needed to express, on the grid of the target models, the coupling fields provided by the source models on their grid. The Driver and the Transformer are described in chapter 3.

The `PSMILe` , linked to the component models, includes the Data Exchange Library (DEL), which performs the MPI-based (Message Passing Interface) (10) exchanges of coupling data, either directly or via additional Transformer processes, and the GFDL mpp_io library (2), which reads/writes the I/O data from/to files following the NetCDF format (9). The `PSMILe` and its Application Programming Interface (API) are described in chapter 4.

The structure and content of the descriptive and configuring XML files are then detailed in chapter 5 and in appendix B. In chapter 6, instructions on how to compile and run a coupled model using OASIS4 are given. Report on OASIS4 scalability is finally given in chapter 7.

OASIS4 functionality has been demonstrated with different "toy" models. 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 the number of component models, the number, size and interpolation of the coupling or I/O fields, the coupling or I/O frequencies, etc. In appendix A, the example toy models available with OASIS4 are described. OASIS4 has also been used to realize the coupling between the MOM4 ocean model (12) and a pseudo atmosphere model reading forcing fields from files and sending them to MOM4 . The results of this coupling are described in (4).

Other MPI-based parallel coupler performing field transformation exist, such as the 'Mesh based parallel Code Coupling (MpCCI)' (1) or the 'CCSM Coupler 6' (3). The originality of OASIS4 relies in its great flexibility, as the coupling and I/O configuration is externally defined by the user in XML files, in its parallel neighborhood search based on the geographical description of the process local domains, and its common treatment of coupling and I/O exchanges, both performed by the `PSMILe` library.

Chapter 2

OASIS4 sources

2.1 Copyright Notice

Copyright 2004 by CERFACS, NEC-CCRLE, SGI Germany, NEC HPCE, and CNRS.

This software and ancillary information called OASIS4 is free software. The public may copy, distribute, use, prepare derivative works and publicly display OASIS4 under the terms of the Lesser GNU General Public License (LGPL) as published by the Free Software Foundation, provided that this notice and any statement of authorship are reproduced on all copies. If OASIS4 is modified to produce derivative works, such modified software should be clearly marked, so as not to confuse it with the current OASIS4 version.

The developers of the OASIS4 software attempt to build a parallel, modular, and user-friendly coupler accessible to the climate modelling community. Although we use the tool ourselves and have made every effort to ensure its accuracy, we can not make any guarantees. The software is provided for free; in return, the user assume full responsibility for use of the software. The OASIS4 software comes without any warranties (implied or expressed) and is not guaranteed to work for you or on your computer. CERFACS, NEC-CCRLE, SGI Germany, NEC HPCE, and CNRS and the various individuals involved in development and maintenance of the OASIS4 software are not responsible for any damage that may result from correct or incorrect use of this software.

2.2 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:

Valcke, S., R. Redler, R. Vogelsang, D. Declat, H. Ritzdorf, and T. Schoenemeyer, 2004: OASIS4 User Guide. PRISM Report Series No 3, 66 pp.

2.3 How to obtain OASIS4 sources

OASIS4 sources, Makefiles, and toy examples can be retrieved from the PRISM CVS repository, which presently is hosted on *bedano* in Zurich. For more details, refer to (7) or to the PRISM web 'Download' page at http://prism.enes.org . The CVS module name is "PRISM_Cpl". The functionality described in this report correspond to the sources tagged "OASIS4_1". To obtain the CVS login and password, please contact us.

2.4 OASIS4 directory structure

The OASIS4 structure obtained when extracting the module "PRISM_Cpl" from the CVS server is the following[1].:

```
-PRISM_Cpl/Makefile
           make-tex
           make_dir/      Makefile environment

-PRISM_Cpl/doc            OASIS4 documentation

-PRISM_Cpl/examples       Different 'toy' examples to test OASIS4

-PRISM_Cpl/include        OASIS4 include files

-PRISM_Cpl/source/del     PSMILe Data Exchange Library sources
                 /driver  Driver and Transformer sources
                 /io      PSMILe I/O sources
                 /mpp_io  GFDL I/O library used by PSMILe
                 /xml     Routines for XML access used by Driver
                          and PSMILe

-PRISM_Cpl/util           Some utilities
```

[1] OASIS4 is not yet adapted to the PRISM Standard directory structure, compiling, and running environments.

Chapter 3

OASIS4 Driver and Transformer

OASIS4 Driver and 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 and Transformer are parallel, although only the main process is used to execute the Driver's tasks.

3.1 The Driver

The first task of the Driver is to get the process management information defined by the user in the SCC XML file (see section 5.3). The information is first extracted using the libxml C library (13), 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 (6) is required as the Driver uses the MPI2 `MPI_Comm_Spawn_Multiple` functionality. If only MPI1 implementation is available (10), the Driver and the applications must be all started at once in the running script; this is the so-called `not_spawn` approach (see also 6.4). 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.

The Driver then participates in the establishment of the different MPI communicators (see section 4.1.3), and transfers the relevant SCC information to the different component model `PSMILe`s (corresponding to their `prism_init` call, see section 4.1.1).

When the PRISM simulation context is set, the Driver accesses the SMIOCs XML files information (see section 5.4), 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` the information relevant for its coupling or I/O exchanges (e.g. source or target components 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 4.1.2). With such information, the PRISM applications and components are able to run without any other interactions with the Driver.

Analysing the XML information, the PRISM Driver is able to determine how many Transformer processes are required, if any. The Driver processes are then used to execute the Transformer routines (see Section 3.2).

When a component reaches the end of its execution, its processes send a signal to the Transformer instance by calling the PRISM_Terminate routine (see Section 4.7.1). Once the Transformer instance has received as many signals as processes active in the coupled run, the Transformer routines stop and the Driver finalizes the simulation.

3.2 The Transformer

The PRISM 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 4.5.1, the neighborhood 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 PRISM 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 ten Integers sent by the component `PSMILe` . These ten 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 4.5.1). 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`.

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 will have been received and treated.

At the end of the run, the Transformer is informed by the participating processes once they are ready to finish the coupled simulation; the Transformer then gives the hand back to the Driver.

Chapter 4

OASIS4 Model Interface library (PSMILe)

To communicate with the rest of the coupled system, each component model needs to perform appropriate calls to the PRISM System Model Interface Library (PSMILe). The PSMILe is the software layer that manages the coupling data flow between any two (possibly parallel) component models, 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 model components and the details of the I/O file access. The detailed communication patterns among the possibly parallel component models 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 5.4) 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 minimize 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 PRISM 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 meta-data 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 4.3.8) 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 it is required (see section 4.3.1), since the grid type implicitly specifies the shape of the data arrays passed to the PSMILe.

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.

The PSMILe API routines that are defined and implemented are not subject to modifications between the different versions of the PRISM 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 new types of 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 model.

4.1 Initialisation phase

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

4.1.1 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 4.1: prism_init arguments

The initialisation of the PRISM interface and the 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, either directly or indirectly via prism_init_comp (see 4.1.2).

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 4.7.1) 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 3.1). 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 5.3) is transfered from the Driver to the application process PSMILe (see 3.1).

4.1.2 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 4.2: prism_init_comp arguments

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

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.

Below the prism_init_comp call, the component SMIOC XML information (see 5.4) is transfered from the Driver to the component process PSMILe(see 3.1).

4.1.3 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 4.3: prism_get_localcomm arguments

MPI communicators for the application and the component model internal communication, separated from the MPI communicators used for coupling exchanges, are provided by the PSMILe and can be accessed via prism_get_local_comm.

If comp_id argument is the component Id returned by routine prism_init_comp, local_comm is a communicator gathering all component processes which called prism_init_comp with the same comp_name argument; 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.

4.1.4 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 4.4: 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.

4.2 Retrieval of SCC XML information

This section presents PSMILe routine that can be used in the application code to retrieve SCC XML information (see 5.3).

4.2.1 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 4.5: prism_get_nb_ranklists arguments

This routine needs to be called before `prism_get_ranklists` (see 4.2.2) to obtain the number of rank lists that are specified for the component model in the SCC XML file (i.e. the number of elements `rank` specified for the element `component`, see 5.3).

4.2.2 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 4.6: 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 model; in the SCC XML file, the component model 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 5.3). For example, if processes numbered 0 to 7 are used to run a component model, 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.

4.3 Grids and related quantities defi nition

NB: Apects included in the `PSMILe` *design but not fully implemented or tested are described in slanted fonts hereafter.*

In order to describe the grids on which the variables of component models are placed, a slightly new approach was chosen.

The first step is to declare a grid (see `prism_def_grid` in 4.3.1). The grid volume elements which discretize the sphere need then to be defined by providing the corner points (vertices) of these volume elements (see `prism_set_corners` in 4.3.2). At this time, other properties of these volume elements can also provided, such as the volume element dimensions (see `prism_set_scalefactors` in 4.3.3). *A 4th dimension within the volume which represents the fraction of different subgrid classes can also be defined so that variables with subgrid informations can be represented (see* `prism_set_subgrid` *in 4.3.6.*

In a second step, different sets of points on which the component model calculates its variables can be placed in these volume elements. Usually, there will be only one definition of volume elements per grid but a larger number of sets of points for different variables on the same grid. The model developer describes where the points are located (see `prism_set_points` in 4.3.8). Points can represent means, extrema or other properties of the variables within the volume. *For vectors, three different sets of points can be placed and associated with each other (see* `prism_set_vector` *in 4.3.9) so that the different vector components need not to be at the same location (staggered grids). Local properties of the points can also be described, e.g. the local angle of the grid at the point (see* `prism_set_angle` *in 4.3.10).*

3D description of all grids

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`).

Fields not located on a geographical grid ('gridless' fields)

The description of the grid and related quantities is done locally for the domain treated by the local process. The patterns used to exchange (or read/write) the coupling (or I/O) fields will usually be based on the geographical description of the process local domain. *For fields not located on a geographical grid, the coupling exchanges (or the I/O) are also supported, based on the description of the process local partition in terms of indices in the global index space (see 4.3.1 and 4.3.7).*

Grids evolving during the run

If the grid definition changes during the run, it is allowed in the code to recall the different routines defining the grid described in this section. A comparison of data arrays will automatically be performed by `PSMILe` *that will detect which part of the data information has changed. Appropriate actions will then be performed concerning the modified part only.*

4.3.1 prism_def_grid

```
prism_def_grid (grid_id, grid_name, comp_id, grid_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 in the PMIOD and SMIOC XML files (unique within the component)
comp_id	In	Integer	component Id as provided by prism_init_comp
grid_valid_shape	In	Integer	array(2, ndim) (see Table 4.8) 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 4.8)
ierror	Out	Integer	returned error code

Table 4.7: prism_def_grid arguments

This routine declares a grid and describes its structure.

- grid_valid_shape

  The array grid_valid_shape is dimensioned (2,ndim) and gives, for each of the ndim dimensions (see Table 4.8), the minimum and maximum local index values corresponding to the "valid" part of the arrays treated by the process, without the halo region, i.e. $iloc_{low}, iloc_{high}, jloc_{low}, jloc_{high}$ on figure 4.1. For example, if the extent of the first dimension is 100, it may be that the "valid" part of the array goes from 2 to 98.

- grid_type

  The argument grid_type describes the grid type and implicitly specifies the shape of the grid description and field arrays passed to the PSMILe. Grids that are supported cover regular grids, irregular grids in longitude and latitude and/or the vertical, completely unstructured grids, and horizontally unstructured grids with regular or irregular vertical axis. Table 4.8 lists:

  - the possible values of grid_type for the different grids supported by PSMILe;
  - the corresponding shape of the grid description arrays sclf_1st_array, sclf_2nd_array, sclf_3rd_array in prism_set_scalefactors, or points_1st_array, points_2nd_array, points_3rd_array in prism_set_points;
  - the corresponding shape of the arrays mask_array, angle_array, subgrid_array and var_array respectively in prism_set_mask, prism_set_angle, prism_set_subgrid, and prism_put / prism_get ;
  - corresponding number of dimensions ndim.

For fields not located on a geographical grid ('gridless' fields), prism_def_grid still has to be called with grid_type=PRISM_gridless and the 'grid' is described solely by its shape and by its partition (see 4.3.7), with no geographical information attached.

The PSMILe API as it is currently defined is able to receive and store coordinates of unstuctured grids (see grid types PRISM_unstructlonlat_regvrt, PRISM_unstructlonlat_sigmavrt, and PRISM_unstructlonlatvrt below). I/O of fields defined on unstructured grids are supported as long as the partition definition (see 4.3.7) of the grid is provided. *However, additional information and*

Figure 4.1: Halo region and local (valid) domain.

grid_type	1st_array	2nd_array	3rd_array	mask_array angle_array subgrid_array var_array	ndim
`PRISM_reglonlatvrt`	(i)	(j)	(k)	(i,j,k)	3
`PRISM_irrlonlat_regvrt`	(i,j)	(i,j)	(k)	(i,j,k)	3
`PRISM_irrlonlatvrt`	(i,j,k)	(i,j,k)	(i,j,k)	(i,j,k)	3
`PRISM_irrlonlat_sigmavrt`	(i,j)	(i,j)	(i,j,k)	(i,j,k)	3
`PRISM_reglonlat_sigmavrt`	(i)	(j)	(i,j,k)	(i,j,k)	3
`PRISM_unstructlonlat_regvrt`	(npt_hor)	(npt_hor)	z(k)	(npt_hor,k)	2
`PRISM_unstructlonlat_sigmavrt`	(npt_hor)	(npt_hor)	(npt_hor,k)	(npt_hor,k)	2
`PRISM_unstructlonlatvrt`	(npt_tot)	(npt_tot)	(npt_tot)	(npt_tot)	1
`PRISM_gridless`					3

Table 4.8: Possible values of `grid_type` for the different grids supported by `PSMILe`.

related API routines have to be defined for processing coupling fields provided on those grids. Therefore, coupling of fields defined on an unstructured grid is not covered yet.

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

4.3.2 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 4.10)
corner_2nd_array	In	Real or Double	corner latitude (see Table 4.10)
corner_3rd_array	In	Real or Double	corner vertical position (see Table 4.10)
ierror	Out	Integer	returned error code

Table 4.9: prism_set_corners arguments

For geographical grids, the volume elements which discretize the computing domain covered locally by the process are defined by giving the corner points (vertices) of those volume elements. The exchange and repartitioning between two coupled component models of a field provided on a geographical grid ('gridded field') 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 4.8), the minimum and maximum local index values corresponding to the "actual" part of the arrays treated by the process including halo regions. `corner_actual_shape` is therefore greater or equal to the `grid_valid_shape` (see section 4.3.1). For example, if the actual extent of the first dimension is 100, it may be that the valid index goes from 0 to 99, or from 1 to 100.

- `corner_xxx_array`

  Units of `corner_xxx_array`

  Units of arrays `corner_xxx_array` describing the grid volume elements should be indicated in the PMIOD and SMIOC XML files; they are not included in the `prism_set_corners` call.

  In the current `PSMILe` version, `corner_1st_array` and `corner_2nd_array` must be respectively provided in degrees East and degrees North (spherical coordinate system). Other units will eventually be supported later when appropriate automatic conversion will be implemented.

  For `corner_3rd_array`, units must be the same on the source and target sides. *In a later stage, `PSMILe` will allow the transfer of functions and associated variables; it will then be able to perform transformation of units (e.g. calculate the pressure at a particular hybrid level k, given the components of the hybrid coordinate at level k, a field of reference pressure, and the surface pressure field).* Furthermore, in case vertical interpolation is needed, only units in which linear interpolation make sense are supported (e.g. meters or pressure, but not hybrid); if no vertical interpolation is required, all vertical units are allowed (as long as they are the same on the source and target sides).

  Shape of `corner_actual_shape`

  Table 4.10 gives the expected shape of the `corner_xxx_array` for the various `grid_type`.

grid_type	corner_1st_array	corner_2nd_array	corner_3rd_array
PRISM_reglonlatvrt	(i,2)	(j,2)	(k,2)
PRISM_irrlonlat_regvrt	(i,j, nc_{half})	(i,j, nc_{half})	(k,2)
PRISM_irrlonlatvrt	(i,j,k,nc)	(i,j,k,nc)	(i,j,k,nc)
PRISM_irrlonlat_sigmavrt	(i,j,nc_{half})	(i,j,nc_{half})	(i,j,k,nc)
PRISM_reglonlat_sigmavrt	(i,2)	(j,2)	(i,j,k,nc)
PRISM_unstructlonlat_regvrt	(npt_hor,nc_{half})	(npt_hor,nc_{half})	(k,2)
PRISM_unstructlonlat_sigmavrt	(npt_hor,nc_{half})	(npt_hor,nc_{half})	(npt_hor,k,nc)
PRISM_unstructlonlatvrt	(npt_tot,nc)	(npt_tot,nc)	(npt_tot,nc)

Table 4.10: 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, and npt_tot are the extent of the respective numerical dimensions (see Table 4.8).

4.3.3 prism_set_scalefactor

```
prism_set_scalefactor(grid_id, sclf_actual_shape, sclf_1st_array,
                      sclf_2nd_array, sclf_3rd_array, ierror ierror)
```

Argument	Intent	Type	Definition
grid_id	In	Integer	grid Id returned by prism_def_grid
sclf_actual_shape	In	Integer	array(2,ndim) giving for each ndim dimension of sclf_xxx_array the minimum and maximum index of actual range (see corner_actual_shape in 4.3.2)
sclf_1st_array	In	Real or Double	Array containing the first scalefactor.
sclf_2nd_array	In	Real or Double	Array containing the second scalefactor.
sclf_3rd_array	In	Real or Double	Array containing the third scalefactor.
ierror	Out	Integer	returned error code

Table 4.11: prism_set_scalefactor arguments. The shapes of sclf_1st_array, sclf_2nd_array and sclf_3rd_array depends on grid_type (see Table 4.8).

The volume element pseudo-dimensions are provided with this PSMILe call. An exact or pseudo horizontal area in the latitude-longitude plane can be calculated by the product of the first two scalefactors; the exact volume of the volume element can be calculated by the product of the 3 scalefactors. This routine is implemented but not used by OASIS4 yet.

4.3.4 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 (see corner_actual_shape in 4.3.2)
mask_array	In	Logical	array of logicals; see Table 4.8 for its dimensions; if an array element is .true. (.false.), the corresponding field grid point is (is not) valid.
new_mask	In	Logical	if .true. a mask is specified for the first time for this mask_id (Out); if .false. mask values for this mask_id (In) are updated
ierror	Out	Integer	returned error code

Table 4.12: 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 4.4.1).

4.3.5 prism_set_vectormask

```
prism_set_vectormask(vectormask_id, mask_ids, new_vectormask, ierror)
```

Argument	Intent	Type	Definition
vectormask_id	InOut	Integer	Id of a set of 3 masks
mask_ids	In	Integer	array(3) containing the mask_ids that correspond to the masks of the 3 component of a vector field
new_vectormask	In	Logical	if .true. a set of 3 masks is specified for the first time for this vectormask_id (Out); if .false. mask_id values for this vectormask_id (In) are updated
ierror	Out	Integer	returned error code

Table 4.13: prism_set_vectormask arguments

This routine associates the mask_ids that correspond to the masks of the 3 component of a vector field. This set of 3 mask_ids will be attached to a vector field in the corresponding prism_def_var call (see section 4.4.1). *I/O of vector fields are currently supported but coupling of vector fields are not.*

4.3.6 prism_set_subgrid

```
prism_set_subgrid (subgrid_id, subgrid_name, grid_id, nbr_subgrids,
                   subgrid_actual_shape, subgrid_array, new_subgrid, ierror)
```

Argument	Intent	Type	Definition
`subgrid_id`	InOut	Integer	subgrid Id
`subgrid_name`	In	character(len=*)	name of the subgrid in the PMIOD and SMIOC XML files (unique within the component)
`grid_id`	In	Integer	grid Id returned by `prism_def_grid`
`nbr_subgrids`	In	Integer	number of subgrid classes
`subgrid_actual_shape`	In	Integer	array(2,ndim) giving for each ndim dimension of `subgrid_array` the minimum and maximum index of actual range (see `corner_actual_shape` in 4.3.2)
`subgrid_array`	In	Real or Double	array giving the fraction for each subgrid class in each cell; see Table 4.8 for its dimensions
`new_subgrid`	In	Logical	if .true. subgrid classes are specified for the first time for this subgrid_id (Out); if .false. subgrid classes for this subgrid_id (In) are updated
`ierror`	Out	Integer	returned error code

Table 4.14: prism_set_subgrid arguments

With `prism_set_subgrid` , a 4th dimension within the volume elements, which represents the fraction of different subgrid classes, can also be defined so that variables with subgrid informations can be exchanged. This routine is implemented but not fully tested yet.

4.3.7 prism_def_partition

```
prism_def_partition (grid_id, nbr_subdomains, offset_array, extent_array,
                     ierror)
```

Argument	Intent	Type	Definition
grid_id	In	Integer	grid Id returned by prism_def_grid
nbr_subdomains	In	Integer	number of subdomains, in the global index space, covered by the grid_valid_shape domain
offset_array	In	Integer	array(nbr_subdomains, ndim) containing for each subdomain the offset in each ndim dimension in the global index space.
extent_array	In	Integer	array(nbr_subdomains, ndim) containing for each subdomain the extent in each ndim dimension in the global index space.
ierror	Out	Integer	returned error code

Table 4.15: prism_def_partition arguments

The local partition treated by the model process can also be described in term of indices in the global index space.

The global index space is a unique and common indexing for all grid points of the component model. For example, if a component model 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 (assuming that arrays have only 1 numerical dimension, i.e. ndim=1) will 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 subdomain and has to be described by its offset and extent in the global index space. Let's suppose, for example, that the 200 grid points of a component model are distributed over two processes such that points 1 to 50 and 76 to 100 are treated by the first process and such that points 51 to 75 and 101 to 200 are treated by the second process. In this case, the number of subdomains for each process is 2, and the first process subdomains can be described (still assuming that arrays have only 1 numerical dimension i.e. ndim=1) 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 subdomains 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).

For I/O of fields given on unstructured grid, this information is mandatory to use the parallel I/O mode (see section 4.6).

In a future version, coupling exchanges and/or I/O of 'gridless' fields, i.e. fields not located on a geographical grid, will also be supported, based on the description of the process local partition in terms of indices in the global index space. For 'gridded' fields, this description has the potential advantage, for full matching source and target grids, of optimizing the neighborhood search (not implemented yet).

4.3.8 prism_set_points

```
prism_set_points ( point_id, point_name, grid_id, points_actual_shape,
                   points_1st_array, points_2nd_array, points_3rd_array,
                   new_points, ierror)
```

Argument	Intent	Type	Definition
point_id	InOut	Integer	set of points Id
point_name	In	character(len=*)	name of set of points name in the PMIOD and SMIOC XML files (unique within the component)
grid_id	In	Integer	grid Id returned by prism_def_grid
points_actual_shape	In	Integer	array(2,ndim) giving for each ndim dimension of points_xxx_array the minimum and maximum index of actual range (see corner_actual_shape in 4.3.2)
points_1st_array	In	Real or Double	array giving the longitudes for this set of grid points; see Table 4.8 for its dimensions
points_2nd_array	In	Real or Double	array giving the latitudes for this set of grid points; see Table 4.8 for its dimensions
points_3rd_array	In	Real or Double	array giving the vertical positions for this set the grid points; see Table 4.8 for its dimensions
new_points	In	Logical	if .true. points are specified for the first time for this point_id (Out); if .false. points for this point_id (In) are updated
ierror	Out	Integer	returned error code

Table 4.16: prism_set_points arguments

With prism_set_points the model developer describes the geographical location of the variables on the grid. Variables can represent means, extrema or other properties of the variables within 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 discretizing a 2D surface must be given a vertical position. Units for points_1st_array, points_2nd_array and points_3rd_array must be respectively the same than the ones for corner_1st_array, corner_2nd_array and corner_3rd_array (see section 4.3.2).

4.3.9 prism_set_vector

```
prism_set_vector (vector_id, vector_name, array_ids, new_vector, ierror)
```

Argument	Intent	Type	Definition
vector_id	InOut	Integer	Id of the vector sets of points
vector_name	In	character(len=*)	name of the vector sets of points in the PMIOD and SMIOC XML files (unique within the component)
array_ids	In	Integer	array(3) containing the point_ids returned by previous calls to prism_set_points used to define the set of points for each vector component
new_vector	In	Logical	if .true. vector sets of points are specified for the first time for this vector_id (Out); if .false. vector sets of points for this vector_id (In) are updated
ierror	Out	Integer	returned error code

Table 4.17: prism_set_vector arguments

For vector fields, sets of points which have been defined for each vector component by a previous call to `prism_set_points` can be linked together with a call to `prism_set_vector` (e.g. on a Arakawa C grid all three vector components are located on different sets of point in the physical space). In any case, three valid `point_ids` need to be specified in `array_ids`. *I/O of vector fields are currently supported but coupling of vector fields are not.*

4.3.10 prism_set_angle

```
prism_set_angle(point_id, angle_actual_shape, angle_array, new_angle, ierror)
```

Argument	Intent	Type	Definition
point_id	In	Integer	point set Id returned by prism_set_points
angle_actual_shape	In	Integer	array(2,ndim) giving for each ndim dimension of angle_array the minimum and maximum index of actual range (see corner_actual_shape in 4.3.2)
angle_array	In	Real or Double	array giving the local angle of the grid at the each grid point; see Table 4.8 for its dimensions
new_angle	In	Logical	if .true. angles are defined for the first time for this point_id (Out); if .false. angles for this point_id (In) are updated
ierror	Out	Integer	returned error code

Table 4.18: prism_set_angle arguments

The local angle of the grid at the each grid point can be described. If a vector sets of points is defined by 3 calls to `prism_set_point` and associated by `prism_set_vector`, than the angles for each of these 3 sets of points will have to be defined by a separate `prism_set_angle` . This routine is implemented but not fully tested yet.

4.4 Declaration of Coupling/IO fields

4.4.1 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=*)	name of the field in the PMIOD and SMIOX XML files (attribute local_name of element transient) (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(for scalar field), or Id of the subgrid as returned by prism_set_subgrid(for subgrid field), or Id of the vector sets of points as returned by prism_set_vector(for vector field), or PRISM_UNDEFINED (for 'gridless' field)
mask_id	In	Integer	Id of the field mask as returned by prism_set_mask, or Id of the set of 3 masks as returned by prism_set_vectormask (for vector field), or PRISM_UNDEFINED.
var_nodims	In	Integer	var_nodims(1): the number of dimensions of var_array that will contain the coupling/IO field (see 4.6), i.e. ndim (see Table 4.8) except for subgrid, vector, and bundle fields, for which it is ndim+1; var_nodims(2): number of subgrids, bundles, or vector components (3), 0 otherwise.
var_actual_shape	In	Integer	array(2,ndim) giving for each ndim dimension of var_array the minimum and maximum index of actual range (see corner_actual_shape in 4.3.2)
var_type	In	Integer	field type: PRISM integer named parameter PRISM_Integer, PRISM_Real or PRISM_Double_Precision
ierror	Out	Integer	returned error code

Table 4.19: prism_def_var arguments

After the initialisation and grid definition phases, each field that will be send/received to/from another

component model (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 the previously defined grids and masks.

The units of a coupling/IO field is indicated in the PMIOD and SMIOC XML files, not in its declaration call. By consulting the appropriate PMIODs, the user is therefore 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 SMIOCs.

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.*

4.5 Neighborhood search and determination of communication patterns

4.5.1 prism_enddef

`prism_enddef (ierror)`

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

Table 4.20: 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.

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 5.4).

For each coupling exchange involving a regridding between the source and the target grids, the neighborhood 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 'neighborhood 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. In an initial step, each process calculates a bounding box covering its local geographical volume domain previously defined by `prism_set_corners` (see section 4.3.2). 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 'gridless fields', see 4.3.1, the intersection calculation is based on the local domain description in the global index space, see 4.3.7.) 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. 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 **??**).

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, minimizing the amount of data to be transferred.

4.6 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 model produces them or whether they are read in from a file. The target or the source (another component model or a file) for each field is defined by the user in the SMIOC XML file (see section 5.4) 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 model. Furthermore source data can be directed to more than one target (other component models 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 and date bounds corresponds to a time at which it should be activated, given the field coupling or I/O dates indicated by the user in the SMIOC; a change in the coupling or I/O dates is therefore also totally transparent for the component model 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.

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 section 5.4.4).

When the action is activated at a coupling or I/O date, 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, then occurs, either directly between the component models, or via additional Transformer processes if regridding needed (see section 3.2).

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 (2) for its file I/O. The supported file format is NetCDF according to the CF convention (5). 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 (bundle or vector) 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 two 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 model during the read operation. This domain stitching or distribution is automatically done by the PSMILe of the component model master process and happens transparently for the parallel component model itself. For unstructured grids, this mode is supported only if the definion of the local partition in terms of indices in the global index space is provided with prism_def_partition (see section 4.3.7).

A fully parallel I/O using the parallel NetCDF (8) library and MPI-IO is currently under investigation. This would allow 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.

The PSMILe I/O layer also copes with the fact that the input data may be spread accross a number of different files[1], 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.

4.6.1 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 the beginning of time step 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 4.8 for its dimensions)
info	Out	Integer	returned info about action performed
ierror	Out	Integer	returned error code

Table 4.21: prism_put arguments

This routine should be 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 5.4). This routine can be called in the component model code at each timestep; the actual date at which the call is performed and the date bounds for which it is valid must be given as arguments as PRISM_Time_Struct structures (see PRISM_Cpl/include/prismf.F90); the sending is actually performed only if the date and date bounds corresponds to a time at which it should be activated, given the field coupling or I/O dates indicated by the user in the SMIOC XML file. The meaning of the different info returned can be accessed using the routine prism_error (see section 4.8.3).

This routine will return even if the corresponding prism_get has not been performed on the target side, both for exchange through the Transformer and for direct exchange (as the content of the var_array is buffered in the PSMILe).

[1]The system calls 'scandir' and 'alphasort' are used to implement this feature (see routine PRISM_Cpl/source/io/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.

4.6.2 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 the beginning of time step 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`	`Out`	`Integer, Real or Double`	field array to be received (see Table 4.8 for its dimensions)
`info`	`Out`	`Integer`	returned info about action performed
`ierror`	`Out`	`Integer`	returned error code

Table 4.22: prism_get arguments

This routine should be 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 5.4). As for `prism_put` , this routine can be called in the component model code at each timestep; the actual date at which the call is performed and the date bounds for which it is valid must be given as arguments; the receiving is actually performed only if the date and date bounds corresponds to a time at which it should be activated, given the field coupling or I/O dates indicated by the user in the SMIOC XML file. The meaning of the different `info` returned can be accessed using the routine `prism_error` (see section 4.8.3). This routine will return only when the corresponding `prism_put` will have been performed on the source side and when data will be available in `var_array`, after regridding if needed.

4.6.3 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 the beginning of time step 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: 0 the corresponding `prism_put` would not be activated; 1, the corresponding `prism_get` would be activated
`ierror`	`Out`	`Integer`	returned error code

Table 4.23: 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.

4.6.4 prism_put_restart

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

Argument	Intent	Type	Definition
var_id	In	Integer	transient handle from prism_def_var
date	In	Type(PRISM_Time_Struct)	date at the beginning of time step 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
info	Out	Integer	returned info about action performed
ierror	Out	Integer	returned error code

Table 4.24: prism_put_restart arguments

This function forces the writing of a field into a coupling restart file. If a coupling restart file of a coupling field is needed[2] but not available, it might be useful to run the source component model beforehand to create the first coupling restart file of an experiment explicitly with a call to `prism_put_restart` (in this case, the coupling field lag has to be equal to 0). Note that since the `prism_enddef` performs some IO related initialisation, a `prism_put_restart` cannot be invoked before the `prism_enddef` is completed. For more information see section A.2.

The time information for each data set that is written into the restart file corresponds to the upper boundary of the time interval which is represented by the data set. To restart from a particular data set the job start date indicated in the SCC.XML needs to correspond to the required time info in the restart file.

Note: Currently it is only possible to dump raw fields into the NetCDF file. 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.

[2]For coupling fields with lag $>$ 0 (see element `lag` in section 5.4.4), a coupling restart file is needed to start the run. In this case, two restart files are opened, one for reading and one for writing. At the beginning of a run, the respective source `PSMILe` processes read their partitions in the coupling restart file during the `prism_enddef` phase and send their local information to the Transformer which performs the interpolation and sends the interpolated fields to the target component model. The name of the reading restart file must be <field_local_name>_<component_local_name>-_<application_local_name>_rst.<date>, where <date> is the current run start date. Below the last call to `prism_put` in the run, the coupling field is also automatically written to its writing coupling restart file; in this case the <date> is the current run end date.

4.7 Termination Phase

4.7.1 prism_terminate

`prism_terminate (ierror)`

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

Table 4.25: prism_terminate arguments

In analogy to the initialisation phase, a call to `prism_terminate`, which again is a collective call, 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 anymore 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`. Furthermore, 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 4.1.1). Otherwise `prism_terminate` will call `MPI_Finalize`.

4.7.2 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 4.26: 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.

4.7.3 prism_abort

`prism_abort (ierror)`

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

Table 4.27: 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-parallelized 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`.

4.8 Query and Info Routines

4.8.1 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 4.28: 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[3] extended to dates before 15 Oct 1582) and its Id is 1 (i.e. the PRISM integer name parameter PRISM_Cal_Gregorian = 1, see PRISM_Cpl/include/prism.inc). In a further 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.

4.8.2 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 4.29: 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. For the date structure PRISM_Time_Struct, see PRISM_Cpl/include/prismf.F90.

4.8.3 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 4.30: 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.

[3] 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.

4.8.4 prism_version

`prism_version()`

This routine prints a message giving the version of the `PSMILe` library currently used.

4.8.5 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 4.31: 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 `PRISM_Cpl/include/prism.inc`).

4.8.6 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 4.32: prism_remove_mask arguments

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

Chapter 5

OASIS4 description and configuration XML files

This document describes the XML files used with Oasis4 in PRISM to:

- describe each application:
  the "Application Description" (AD)
- specify the general characteristics of a (coupled) model run:
  the "Specific Coupling Configuration" (SCC)
- describe the relations a component model is able to establish with the rest of the coupled model through inputs and outputs:
  the "Potential Model Input and Output Description" (PMIOD)
- specify the relations the component model will establish with the rest of the coupled model through inputs and outputs for a specific run:
  the Specific Model Input and Output Configuration (SMIOC).

5.1 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 DTD described in ad.dtd, scc.dtd, pmiod.dtd and smioc.dtd respectively. The Schema corresponding to the PMIOD and the SMIOC, pmiod.xsd and smioc.xsd, have also been written. Those DTD and Schema are available in appendix B, and in the directory `PRISM_Cpl/util/xmlfiles`.

The *xmllint* command with the following options can be used to validate an XML file *file.xml* against a DTD *file.dtd*:

```
xmllint --noout --valid --postvalid --dtdvalid file.dtd file.xml
```

or 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.dtd` are “component” or “code”. Elements can contain text (“#PCDATA” in the DTD), other elements, or be empty.

  When an element is marked in the DTD with

  – *, it may appear 0, 1, or many times in the XML file;

  – +, it may appear 1, or many times in the XML file;

  – ?, it may not appear or appear once in the XML file.

- Tags

  Tags are used to markup elements.

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

  Attributes are placed inside the start tag of an element.

  Example: <element_name attribute_name=“attribute_value” />

  The name of the element is “element_name”. The name of the attribute is “attribute_name”. The value of the attribute is “attribute_value”. Note that here since the element itself is empty it is closed by a “ /”.

  As indicated in the DTD, an attribute may be “REQUIRED” (i.e. it must be present in the XML file), “IMPLIED”(i.e. it may or may not appear in the XML file) or “FIXED” (i.e. it must be present in the XML file and has a fixed and pre-defined value).

5.2 The Application Description (AD)

The Application Description (AD) describes the general characteristics of one application. The AD DTD is given in Appendix B.1 and in directory `PRISM_Cpl/util/xmlfiles`. There is one AD per application, i.e. per code which when compiled forms one executable. The AD is written by the application developer[1]. The AD file name must be <application_local_name>_ad.xml where <application_local_name> is the application local_name attribute in the scc.xml file (see section 5.3). *The AD information will be available to the user through the GUI when he selects the application for his coupled model.*

The AD contains the element 'application' which is composed of:

- the application name (attribute 'local_name')which must match argument `appl_name` of `PSMILe` call `prism_init` (see section 4.1.1);
- 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 3.1);
- 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 must match the argument `comp_name` of `PSMILe` call `prism_init_comp` (see section 4.1.2);
  - a description of the component (attribute 'long_name');
  - the simulated part of the climate system (attribute 'simulated': either `ocean`, `sea_ice`, `ocean_biogeochemistry`, `atmosphere`, `atmospheric_chemistry`, or `land`); if an AD contains more than one component simulating the same part of the climate system, the user will have in the SCC (see below) to choose among these components;
  - 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').

5.3 The Specific Coupling Configuration (SCC)

The Specific Coupling Configuration (SCC) contains the general characteristics and process management information of one coupled model simulation. The SCC DTD is given in Appendix B.2 and in directory `PRISM_Cpl/util/xmlfiles`. There is one SCC per coupled model (or per stand-alone application). *In the complete PRISM system, the SCC will be generated by the GUI; it includes choices made by the user based on the information contained in the ADs of selected applications, but also other user's choices and running environment information.* The SCC file name must be `scc.xml`[2].

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 4.2). In many cases, coherence with the compiling and running environment and scripts has to be ensured.

[1]On the longer term, in order to avoid duplication of information, it is foreseen to develop a tool to extract automatically all AD information which is already in the code (e.g. the component names given argument `comp_name` of `prism_init_comp` calls).

[2]When using the `prismrun` command to execute the toy examples (see section 6.4), a different file name can be specified with "-f". Note that this will overwrite an existing `scc.xml` in the directory where the `prismrun` is started.

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 3.1); this user's choice, restricted by the possibilities given in the ADs, determines the way the applications should be started in the run script (see section 6.4).
  - the number of processes used for the Driver/Transformer (element 'nbr_procs' of element 'driver') This argument is needed in the `not_spawn` case in order to split the global communicator. In the `spawn` case it must be identical to the number of processes used to start the Driver/Transformer (i.e. the size of the initial communicator given by the "-np" parameter of the `mpirun` command or the "-drv_np" parameter of the `prismrun` script (see section 6.4).
  - 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, defined by the running environment, which must be therefore changed for each run (element 'run'):
  - the start date of the run (element 'start_date'); the start date should 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 should correspond to the upper bound of the time interval which is represented by the last time step of the run. the end date of the current run has to be used as start date for the subsequent run, especially when a lag is defined.
- 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 `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 or not application stdout is redirect or not (user's choice) (attribute 'redirect', either `true` or `false`)
  - a list of launching arguments (chosen by the user in the list given in the corresponding AD) (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', chosen by the user 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 4.1.2);
    * 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).

5.4 The Potential Model Input and Output Description (PMIOD) and the Specific Model Input and Output Configuration (SMIOC)

NB: For completeness, all the elements and attributes that are currently in the PMIOD and SMIOC DTD and Schema will be described in the following sections. However, some of those elements or attributes are not considered by the current OASIS4 and `PSMILe` version, or their corresponding functionality is not yet supported. The description of those elements and attributes, that are all optional, will appear in slanted characters.

The Potential Model Input and Output Description (PMIOD) describes the relations a component model is able to establish with the rest of the coupled model through inputs and outputs. There is one PMIOD per component model. The PMIOD DTD and Schema are given in Appendices B.3.1 and B.3.2, and in directory `PRISM_Cpl/util/xmlfiles`. The PMIOD is written by the component developer[3]. The PMIOD file name must be <application_local_name>_<component_local_name>_pmiod.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. In the complete PRISM system, the PMIOD of components included in selected applications will be available through the GUI to the user in the definition phase of a coupled model.

The Specific Model Input and Output Configuration (SMIOC) specifies the relations the component model establishes at run time with the rest of the coupled model through inputs and outputs for a specific run. The SMIOC DTD and Schema are given in Appendix B.4.1 and B.4.2and in directory `PRISM_Cpl/util/xmlfiles`. It is generated in the composition phase by the user through the GUI for each component model based on the corresponding PMIOD information. The SMIOC file name must be <application_local_name>_<component_local_name> 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. The PMIOD and the SMIOC have analog structure, syntax and content rules[4].

For coupled components, a coherence between the respective SMIOCs has to be ensured. For example, if component B is specified, in component A SMIOC, as the target for an ouput variable, then component A must be identified, in component B SMIOC, as the source for the corresponding input variable. In the complete PRISM system, this coherence will be automatically ensured by the GUI.

The PMIOD and SMIOC contains 6 types of information:

- general characteristics of the component
- information on the grids
- information on the coupling/IO fields, also called 'transient variables'
- *information on the persistent variables (to be further defined)*
- *information on internal dependency between elements (still to be defined)*

5.4.1 Component model general characteristics

This type of information gives an overview of the component model; it is defined by the component developer in the PMIOD and should appear as is in the SMIOC:

- the component name (attribute 'local_name'), which must match the 2nd argument of `PSMILe` call `prism_init_comp`(see section 4.1.2);
- a short general description of the component model (attribute 'long_name')

[3]On the longer term, in order to avoid duplication of information, it is foreseen to develop a tool to extract automatically all PMIOD information which is already in the code (e.g. the component name given argument `comp_name` of `prism_init_comp` calls)

[4]However, on the longer term it is foreseen to keep all the descriptive information only in the PMIOD, and all the configuring information only in the SMIOC

- the simulated part of the climate system (attribute 'simulated': either `ocean`, `sea_ice`, `ocean_biogeochemistry`, `atmosphere`, `atmospheric_chemistry`, or `land`);
- 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')
- the Fortran Units used by the component (element 'Fortran_units' in element 'code'), given as lists of 3 numbers giving, in each list, a minimum value, a maximum value, and an increment value (see also the description of the lists of ranks given in section 5.2).

5.4.2 Grid families and grids

This part contains information on the grids used by the component model. There might one or more grid families per component; for each grid family (element 'grid_family'), there may be one or more grids (elements 'grid'). All grids of all families are described by the component developer in the PMIOD. In the SMIOC, the user chooses one grid (with all its sub-elements, which he is not allowed to modify) for each grid family.

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 4.3.1).
- for the physical domain covered by the grid (element 'physical_space'):
  - a general description (attribute 'long_name')
  - for the longitude dimension (element 'longitude_dimension'): the domain minimum and maximum and the units (elements 'valid_min', 'valid_max', and attribute 'units': for now, only `degrees_east` supported, see also section 4.3.2)
  - for the latitude dimension (element 'latitude_dimension'): the domain minimum and maximum and the units (elements 'valid_min', 'valid_max', and attribute 'units': for now, only `degrees_north` supported, see also section 4.3.2)
  - for the vertical dimension (element 'vertical_dimension'):
    - the domain minimum (element 'valid_min')
    - the domain maximum (element 'valid_max')
    - the units (attribute 'units': either `meters`, `bar`, `millibar`, `decibar`, `atmosphere`, `pascal`, `hPa`, `dimensionless`, , see also section 4.3.2)
    - the direction in which the coordinate values are increasing (attribute 'positive', either `up` or `down`)
    - *possibly a 'units_standard_name' attribute*
    - *possibly a 'long_name' attribute*
    - *possibly a 'formula_terms' attribute (for dimensionless vertical coordinates; see (5))*
- for the sampled domain covered by the grid (element 'sampled_space'):
  - whether or not the grid covers the pole (attribute 'pole_covered', either `true` or `false`)
  - the grid mesh structure type (attribute 'grid_type', which must match argument `grid_type` of `prism_def_grid` (see section 4.3.1), either `PRISM_reglonlatvrt`, `PRISM_irrlonlat_regvrt`, `PRISM_irrlonlat_sigmavrt`, `PRISM_reglonlat_sigmavrt`, `PRISM_unstructlonlat_regvrt`, `PRISM_unstructlonlat_sigmavrt`, `PRISM_unstuctlonlatvrt`, `PRISM_gridless`)
  - for each global index dimension (elements 'indexing_dimension'):
    - the index name (attribute 'local_name')

    * whether or not the number of indices in this dimension is time dependent (attribute 'time_dependency', either `true` or `false`)
    * whether or not the grid is periodic is this dimension (attribute 'periodic' either `true` or `false`)
    * the typical number of indices, i.e. the resolution, in this dimension (element 'extent')
    * the number of overlapping grid points in this index dimension, if any (element 'nbr_overlap')

- *the computational space covered by the grid (element 'compute_space'). In the `PSMILe` , a grid is defined by its volume elements which discretize the domain covered. In these volume elements, a number of sets of points, on which the variables are calculated, can be placed. For vectors, three sets of points can be placed so that the vector components need not to be at the same location. Subgrids can also be defined in the volume elements by giving the fraction atributed each of the subgrid classes. Element 'compute_space' gives the user a description of the sets of points, of vector sets of points, and on subgrids declared in the component code, but is currently not used in the `PSMILe` :*
  - *elements 'points': the sets of points defined on the grid, declared in the code with `PSMILe` call `prism_set_points` (see section 4.3.2); for each set:*
    * *a local name which must match 2nd argument in prism_set_point (attribute 'local_name')*
    * *a description of the set of points (attribute 'long_name')*
    * *whether or not the geographical position of this set of points is time dependent (attribute 'time_dependency', either `true` or `false`)*
  - *elements 'subgrid': the subgrids defined on the grid, declared in the code with `PSMILe` call `prism_set_subgrid` (see section 4.3.6); for each subgrid:*
    * *a local name which must match 2nd argument in prism_set_subgrid (attribute 'local_name')*
    * *a description of the subgrid (attribute 'long_name')*
    * *whether or not the definition of this subgrid is time dependent (attribute 'time_dependency' either `true` or `false`)*
    * *an associated set of points if needed for localisation of the subgrid in the volume element (attribute 'associated_points_local_name')*
  - *elements 'vector': the vector sets, declared in the code with `PSMILe` call `prism_set_vector` (see section 4.3.9), which associates, for a vector variable, the 3 pre-defined sets of points on which the vector components are located; for each vector set:*
    * *a local name which must match 2nd argument in `PSMILe` call `prism_set_vector` (attribute 'local_name')*
    * *a description of the vector (attribute 'long_ name')*
    * *the local name of the set of points on which the first component is located (attribute 'firstcomp_points_local_name')*
    * *the local name of the set of points on which the second component is located (attribute 'secondcomp_points_local_name')*
    * *the local name of the set of points on which the third component is located (attribute 'thirdcomp_points_local_name')*

5.4.3 Coupling/IO fi elds (transient variables)

The coupling/IO fields, also called transient variables, are scalar, vector or bundle variables which values evolve during the run as they are received/provided by the component model at a priori unknown times from/to its external environment (another model or a disk file) through `PSMILe` calls `prism_put` and `prism_get` (see sections 4.6.1 and 4.6.2).

Some of the coupling/IO field information is defined by the component developer in the PMIOD and should appear as is in the SMIOC, whereas some of this information may be modified and/or defined by the user depending on the coupling configuration he/she wants to assemble[5]. Elements or attributes that are defined by the developer in the PMIOD and that cannot changed by the user in the SMIOC are marked 'PMIOD defined' below.

The element 'transient', which must appear fo each coupling/IO field, has the following attributes and sub-elements:

- attribute 'local_name': it must match 2nd argument in the corresponding `PSMILe` call `prism_def_var` (see sections 4.4.1) (PMIOD defined);
- attribute 'long_name': gives a general description of the variable (PMIOD defined);
- element 'transient_standard_name': one or more PRISM standard names following the CF convention (if they exist). This uniquely identifies the nature of the coupling/IO field. In case of vector, three elements need to be specified (one for each vector component). 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. For bundle of vectors, three elements giving the generic names of the vector components (e.g `mass_flux_eastward`, `mass_flux_westward`, `mass_flux_up`), then one element per bundle species giving the species specific name (e.g. `nitrogen_oxide`, `hydrogen_peroxide`, `methane`, `carbon_monoxide`) need to be specified (PMIOD defined);
- element 'physics': a description of the coupling/IO field physical constraints; this information is only descriptive (PMIOD defined):
  - attribute 'transient_type': the coupling/IO field physical type (either 'single', 'vector', 'bundle', or 'bunvec' for bundle of vectors);
  - 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', which attribute 'datatype' gives the coupling/IO field numeric type (either `xs:real`, `xs:double`, or `xs:integer`); this information is only descriptive (PMIOD defined):
- element 'computation', which attributes and sub-elements give some information on the coupling/IO field computational characteristics; this information is only descriptive (PMIOD defined):
  - attribute 'maks', which tells whether or not a mask is associated to the coupling/IO field (either `true` or `false`).
  - attribute 'mask_time_dependency', which tells whether or not the mask evolves with time (either `true` or `false`).
  - attribute 'conditional_computation', which, if present, indicates under which condition the coupling/IO field is effectively sent and/or received.
  - attribute 'method_type', which, if present, indicates what the coupling/IO field value represents on the grid cell, either `mean`, `max`, `min`, `median`, `variance`, or something else described in a character string.
  - element 'associated_gridfamily', which attribute 'local_name' must be the same than the one of the grid family associated to the coupling/IO field.
  - element 'associated_compute_space', which attribute 'local_name' must be the same than the one of the computational space associated to the coupling/IO field (i.e. the attribute 'lo-

[5]On the longer term it is foreseen to keep all the descriptive information only in the PMIOD, and all the confi guring information only in the SMIOC.

cal_name' of either the associated set of points -element 'points'-, the associated subgrid -element 'subgrid'-, or the associated vector sets - element 'vector'.

- element 'intent', which contains in its sub-elements all coupling and I/O information (source and/or target, frequency, transformations, interpolation, etc.) for the coupling/IO field. In the component code, the coupling/IO field may be exported (with `PSMILe` `prism_put` call, see section 4.6.1) or imported (with `PSMILe` `prism_get` call, see section 4.6.2), or both. The sub-elements of 'intent' are:
  - element 'output': If the coupling/IO field is exported through a `prism_put` , it can be effectively be sent to none, one, or many targets, each target being described in one element 'ouput'. The element 'output' is described in more details in section 5.4.4.
  - element 'input': If the coupling/IO field is imported through a `prism_get` , this import is described in one element 'input'. *If an import comes from a combination of different sources, these sources must be described in different sub-elements 'origin'.* The element 'input' is described in more details in section 5.4.5.
- element 'transient_dependency': If the developer wants to indicate a dependency between the coupling/IO field and another coupling/IO field from the same component, he has to define an element 'transient_dependency' and to specify this dependency in the attribute 'dep_variable'. For example, coupling/IO field A is a transient_dependency of coupling/IO field B if coupling/IO field A is used in the calculation of variable B. This information may be needed to prevent deadlocks (PMIOD defined).

5.4.4 The 'output' element

If the coupling/IO field is exported through a `prism_put` in the component code, the developer must describe one default target file in an 'output' element in the PMIOD. In the SMIOC, the user is allowed to remove or modify this 'output' element, and/or add additional 'output' elements to describe additional targets (corresponding to the same prism_put). The only sub-elements of 'output' the user is not allowed to modify is 'minimal_period' and 'scattering' (see below). A more detailed description of element 'output', its attributes and sub-elements is given here.

1. attribute 'transi_out_name': a symbolic name defined by the user for that specific 'output' element.
2. element 'minimal_period': The period at which the `prism_put` is called in the code (PMIOD defined). 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'.
3. element 'exchange_date': The dates at which the coupling or I/O is effectively performed. To express these dates, the user has to specify one (and only one) of the following sub-element:
   - element 'period': The coupling or I/O is performed with a fixed period. To define this period, the user may specify a number of seconds, minutes, hours, days, months, and/or years, with respectively the sub-elements 'second', 'minute', 'hours', 'day', 'month', 'year'.
   - *element 'once': either* `beginning`, `end`, `beginning_and_end`, *i.e. the coupling or I/O is performed only at the beginning of the run, only at the end, or both (not implemented yet).*
   - *element 'precise_list': The coupling or I/O is performed on one or more precise dates (element 'date'), every 'year' years. To express a precise date, the user specifies a value for 'second', 'minute', 'hour', 'day', and/or 'month'. For example, the user may specify that the coupling will occur on January 1st and July 1st every second year by defining two elements 'date'; each 'date' will have the sub-elements 'month', 'day' and 'year' with the values* `1`, `1`, *and* `2`, *for the first one , and* `1`, `7`, `2` *for the second one (not implemented yet).*

- *element 'lastdayinmonth': The coupling or I/O is performed at a precise time, expressed by the sub-element 'second', 'minute', and 'hour' on the last day of each month (not implemented yet).*
- *element 'date_list': The coupling or I/O is performed on a list of different dates (elements 'date'). To express each 'date', the user may specify a value for 'second', 'minute', 'hour', 'day', 'month', and/or 'year' (not implemented yet).*

4. element 'corresp_transi_in_name': 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 section 5.4.5).
5. element 'file' or element 'component_name': The target file description (I/O) or the target component 'local_name' attribute (coupling). The 'file' element is described in more detail in section 5.4.7.
6. element 'lag': The number of `prism_put` periods[6] 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 4.6.4).
7. element 'source_transformation': The transformations performed on the output coupling/IO field in the source component `PSMILe` .
   - element 'source_time_operation': for each grid point, the output coupling/IO field is averaged (`taverage`) or accumulated (`accumul`) over the last coupling period, or its minimum (`tmin`) or maximum (`tmax`) value over the last coupling period is calculated below the `prism_put` by the source `PSMILe` and the result is transfered.
   - element 'statistics': different statistics (field minimum, field maximum, field integral) are calculated 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 'not-masked_points', and/or 'all_points' are specified. This is done below the `prism_put` by the source `PSMILe` (after the time operations described in element 'source_time_operation' if any). These statitistics are printed to the `PSMILe` log file for information only; they do not transform the output coupling/IO field.
   - element 'source_local_transformation': the following local transformations may performed on the output coupling/IO field by the source `PSMILe` :
     - element 'scattering': the 'scattering' is specified by the developer in the PMIOD and cannot 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 (PMIOD defined).
     - *element 'reduction': the user may specify one type of reduction (sub-element 'reduc_type') for each coupling/IO field dimension (sub-element 'reduc_dim') (i.e. possibly different reductions for the different dimensions) and the order in which they should be performed ('reduc_order'). The possible reductions, 'reduc_type', are:* `minimum` *(extraction of the minimum value along the dimension),* `maximum` *(extraction of the maximum value along the dimension),* `average` *(average of the values along the dimension),* `wgt_average` *(average of the values weighting them by the respective length of the cell along the dimension),* `subgrid_average` *(average of a subgrid output coupling/IO field along the*

[6]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.

subgrid dimension weighting by the respective subgrid fraction prism_set_subgrid). Bundle combination (which in fact is just an average along the bundle dimension dimension) is covered here.

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

8. element 'debug_mode': either `true` or `false`; if it is `true`, the output coupling/IO field is automatically also written to a file below the `prism_put` .

5.4.5 The 'input' element

If the coupling/IO field is imported through a `prism_get` in the component code, the developer must describe one default source file in an 'input' element in the PMIOD. *If an input comes from a combination of different sources, the user must describe these sources in different 'origin' sub-elements, and the combination must be identified in the sub-elements 'algebraic_combination' (not implemented yet).* A more detailed description of element 'input', its attributes and sub-elements is given here.

1. attribute 'required_but_changeable': if the developer indicates in the PMIOD that this attribute is `true`, the user must keep at least one 'input' element in the SMIOC: it can be the developer-defined one or a modified one; if it is `false`, then an 'input' with no 'origin' sub-elements, or modified 'origin', or the developer-defined one may appear in the SMIOC. (In all cases, the user is not allowed to modify sub-elements 'minimal_period' and 'gathering' - see below.)
2. element 'minimal_period': The period at which the `prism_get` is called in the code (PMIOD defined). (See element 'minimal_period' of element 'output' in section 5.4.4.)
3. element 'exchange_date': The dates at which the coupling or I/O is effectively performed (see 'exchange_date' in 'output' in section 5.4.4).
4. element 'origin': An input coupling/IO field may come from one *or many (in the case of combination)* origin*s*, each one being described by an element 'origin' which contains the following attributes and sub-elements:
   - attribute 'transi_in_name': a symbolic name defined for that specific 'origin' element.
   - element 'corresp_transi_out_name': 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 section 5.4.4).
   - element 'file' or 'component_name': The source file description (I/O) or the source component 'local_name' attribute (coupling). The 'file' element is described in more detail in section 5.4.7.
   - *element 'cpl_rst_file': For a coupling field with corresponding output lag > 0, the description of the file from which it is read at the beginning of the run, and to which it is written at the end of the run. Its sub-element 'file' is described in more detail in section 5.4.7. (Not supported yet; only the default coupling restart file name is used, see section 4.6.4.)*
   - element 'middle_transformation': The transformations which link the source and the target.
     - element 'interpolation': 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 5.4.6.

  - *element 'conservation': the source* `PSMILe` *calculates the integral of the source field before interpolation and the target* `PSMILe` *calculates the integral of the field after interpolation; the difference is distributed over the target field to ensure global conservation. The only possible value of this element is* `global` *for now. (not implemented yet)*
  - *element 'algebraic_combination': if an input coupling/IO field comes from a combination of different sources, the 'origin' sub-elements of 'input' describing those sources, must all have an element 'algebraic_combination' (not implemented yet).*

5. element 'target_transformation': The transformations performed on the input coupling/IO field in the target component `PSMILe` .
   - element 'target_local_transformation': The local transformations performed on the input coupling/IO field.
     - element 'gathering': The 'gathering' is specified by the developer in the PMIOD and cannot be removed or added 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.
     - element 'add_scalar': The scalar specified in this element is added to each grid point coupling/IO field value.
     - element 'mult_scalar': Each grid point coupling/IO field value is multiplied by the scalar specified in this element.
   - element 'target_time_operation': Target time interpolation is supported below the `prism_get` only for IO data[7]. 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': see above in section 5.4.4.
6. element 'debug_mode': either `true` or `false`; if it is `true`, the input coupling/IO field is automatically written to a file below the `prism_get` .

5.4.6 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[8].

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 i.e. for source grid types `PRISM_reglonlatvrt`, `PRISM_irrlonlat_regvrt`, `PRISM_unstructlatlon_regvrt`. The mask may vary with depth. Its interest is to allow a linear interpolation in the vertical. Currently the combination of

[7]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.

[8]In the current OASIS4 version, interpolation is available only for coupling fi elds. In a future version, interpolation might also be possible for I/O fi elds read/written from/to a fi le.

2D and 1D interpolations that are supported are `bilinear` and `linear`, `bilinear` and `none`, `nneighbour2D` (nearest-neighbour) and `none` (see below).

- *'(interp1D_i, interp1D_j, interp1D_k)': The same 1D zonal interpolation, followed by a 1D meridional interpolation, followed by a 1D interpolation in the vertical ; this type of interpolation make sense only when the zonal, meridional and vertical grids are decoupled (i.e. for source grid type* `PRISM_reglonlatvrt`*) (not supported yet).*

The elements 'interp3D', 'interp2D', 'interp1D', *'interp1D_i', 'interp1D_j' and 'interp1D_k'* are separately described here after:

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 'para_search': *either* `global` *(an exact but more expensive parallel neighborhood search) or* `local` (a local but less expensive neighborhood search).
     - element 'nbr_neighbours': the number of neighbours.
     - element 'used_masked': either `true` (all points are considered in the `PSMILe` neighbourhood search and the Transformer detects masked points), or `false` (the nearest neighbours are chosen by the `PSMILe` among non-masked points only).
     - element 'gaussian_variance': the variance of the Gaussian function used to weight the neighbours, if any.
   - element 'trilinear': A trilinear algorithm; the parameters are:
     - element 'para_search': see element 'nneighbour3D' above.
     - *element 'if_masked': either* `novalue`*,* `tneighbour`*, or* `nneighbour`.
   - *element 'conservativ3D': A 3D conservative remapping.*
   - *element 'user3D': A user-defined 3D interpolation is chosen by the user. The sub-element 'file' (see section 5.4.7 describes the file in which the interpolation weights and addresses are available (not implemented yet).*
2. element 'interp2D': For 2D interpolation, the following methods can be chosen:
   - element 'nneighbour2D': A 2D nearest neighbour algorithm; the parameters are:
     - elements 'para_search', 'nbr_neighbours', 'used_masked', 'gaussian_variance': see element 'nneighbour3D' above.
   - element 'bilinear': A bilinear algorithm; for the parameters are:
     - element 'para_search': see element 'nneighbour3D' above.
     - *element 'if_masked': see element 'trilinear' above.*
   - *element 'bicubic': A bicubic algorithm.*
   - *element 'conservativ2D': A 2D conservative remapping.*
   - *element 'user2D': A user-defined 2D interpolation is chosen by the user. The sub-element 'file' (see section 5.4.7) describes the file in which the interpolation weights and addresses are available.*
3. element 'interp1D' *or 'interp1D_i' or 'interp1D_j' or 'interp1D_k'*: For 1D interpolations, the following methods can be chosen:
   - *element 'nneighbour1D': A 1D nearest neighbour algorithm.*
   - element 'linear': A linear algorithm; for the sub-elements, see above.
   - *element 'cubic': A cubic algorithm; for the sub-elements, see above.*
   - *element 'conservativ1D': A conservative remapping.*
   - *element 'user1D': A user-defined 1D interpolation; the sub-element 'file' (see section 5.4.7) describes the file in which the interpolation weights and addresses are available.*

- 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'.

5.4.7 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`. If 'suffix' is false (by default), the file name is composed only of element 'name'; if 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.
- element 'format': the format of the file; only NetCDF (`mpp_netcdf`) supported for now. *Other formats foreseen are ascii (*`mpp_ascii`*), IEEE (*`mpp_ieee32`*), native mpp_io (*`mpp_native`*).*
- 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. *The mode* `parallel` *means that each process writes its part of the field to one parallel file; this last option, that will rely on the use of parallel NetCDF based on MPI_IO, is not implemented yet.*
- 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.

5.4.8 *Persistent variables*

The persistent variables are scalar variables which value is fixed in the SMIOC during the configuration process and does not evolves during the run. The treatment of persistent variables are not implemented yet; it needs to be more precisely defined.

5.4.9 *Dependency*

Not implemented yet; needs to be more precisely defined.

Chapter 6

Compiling and Running with OASIS4

6.1 Introduction

OASIS4 and its example toy models (see the description in Appendix A and sources in `PRISM_Cpl/examples`) were compiled and run on the following systems:

- AMD Athlon 2800 Cluster,
- AMD Opteron 848 Cluster,
- Intel Pentium 4 Workstation Cluster,
- NEC SX6,
- SGI O3000/2000 server with MIPS 4 processors and IRIX64,
- SGI IA64 Linux server Altix 3000,

with the following Fortran Compilers:

- Absoft Fortran Compiler Version 9.0 r2 32Bit,
- Intel Fortran Compiler Version 8.0 64Bit,
- Intel Fortran Compiler Version 8.0 32Bit,
- Pathscale Fortran Compiler Version 1.4.1 64Bit,
- Portland Group Compiler Version 4.1-2 32Bit
- SGI Fortran Compiler,
- NEC SX Fortan Compiler.

6.2 Compiling OASIS4 and its associated PSMIle library

To compile OASIS4[1], one has to follow these steps:

- The file that gives the options for compiling must be indicated through the file `PRISM_Cpl/make.inc`. Some configurations have already been set up in the `PRISM_Cpl/make_dir` directory. Note that the following librairies (not provided with the OASIS4 sources) are required:
  - Message Passing Interface, MPI1 (10) or MPI2 (6) (MPICH, SGI native MPI, NEC SX native MPI, and LAM-MPI were successfully tested)
  - NetCDF Version 3.4 or higher (5)
  - libxml Version 2.6.5 or higher (11)
- The CPP keys that can be activated (see `CPPDEF` in the`PRISM_Cpl/make_dir/make.xxx` files) are:

[1] OASIS4 is not adapted to the PRISM Standard Compiling Environment (SCE) yet

  - PSMILE_WITH_IO: to make use of the IO capability of PSMILe in the sources in directories xml, del and io. The sources in mpp_io need also to be compiled. This can be achieved by typing 'make with_io' in directory PRISM_Cpl instead of simple 'make'.
  - PRISM_WITH_MPI1: This options has to be chosen if the available MPI library supports MPI1 standard, like mpich1.2.* or does not support the full MPI2 standard.
  - PRISM_WITH_MPI2: When the available MPI2 library supports the complete MPI2 standard this option may be chosen instead.
  - 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 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: 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.
  - PRISM_ASSERTION: Mainly used by the developers; the code encapsulated by this cpp key will perform additional internal consistency checks and will provide additional information for debugging.

- If 'make' should be used instead of 'gmake', replace 'gmake' by 'make' in PRISM_Cpl/Makefile.
- The different source directories are compiled through the top Makefile. In directory PRISM_Cpl do 'make realclean' and 'make' to compile and generate the libraries and the Driver/Transformer executable. The option 'realclean' insures that the library and binary directories are regenerated. This option is recommanded to avoid compilation conflicts. It is possible to compile just one source directory but one has to be sure that there is no dependency between what was re-compiled and what was previously compiled (especially with the source/xml directory).
- The PSMILe library and the Driver/Transformer executable prismdrv_main should then be generated.

6.3 Compiling OASIS4 example toy coupled models

The toymodel examples, described in more detail in Appendix A, are built following the same structure. Any execution deals with the OASIS4 Driver/Transformer and two models, except for proto_ex for which there are 3 models. The compilation is done through the 'make' command in the PRISM_Cpl/examples sub-directories. To clean the directory, use 'make clean'.

6.4 Running an example toy coupled model with OASIS4

The way the applications are launched[2] depends on the platforms and the launch strategy used for it, set in the scc.xml file (i.e. not_spawn, or spawn, see also sections 3.1 and 6.5).

For running the examples simple-mg* on architectures on which the current release was tested, the script PRISM_Cpl/util/prismrun serves as an aid on how to start the examples. Note that to use this script, you have to define the environment variables PSMILE_HOME (complete path for PRISM_Cpl

[2]The example toy coupled models are not adapted to the PRISM Standard Running Environment (SRE) yet.

directory), MPIRUN (mpirun command with its complete path), and MPIEXEC if LAM-MPI is used (lam mpiexec command with its complete path) at the beginning of the script.

For not_spawn mode, the prismrun script uses an MPI configuration file applmg-<archi>.conf, where <archi> depends on the targeted architecture, either cosy, cosyi, linux, sx, irix64, lam, or mpich; the number of processors used by each application needs to be specified in this MPI configuration file used and must be coherent with the values indicated in the scc.xml file. For the spawn mode, only the number of processors for the Driver/Transformer needs to be specified in the prismrun option -drv_np. If LAM-MPI is used, the prismrun script uses also the lam.config file; the numbers of processors used need to be specified also in this file and must be coherent with the values indicated in the SCC XML file (see section 5.3).

To get help on how to use the script, use the -help option:

```
>prismrun -help
```

To get architecture specific help use

```
>prismrun <-cosy|-cosyi|-lam|-mpich|-linux|-sx|-irix64> -help
```

For example, for irix64, type in the PRISM_Cpl/examples/simple-mg directory:

```
>prismrun -irix64 -help
```

This will tell you how to start the example coupled experiment. For example, for irix64, type:

```
>prismrun -irix64 -f scc.xml -conf applmg-irix64.conf
```

After a successfull execution of the examples simple-mg*, the files atm.0 and ocn.O should be present and end with the sentence 'No errors detected'.

For running the example proto_ex, the executable prismdrv_main and the three model executables toyatm.exe, toyatm.exe, toylan.exe need to be started. As for the simple-mg* examples, the way the applications are launched depends on the platforms and on parameters specified in the SCC XML file (launching strategy, number of processors used by the coupler and each application, etc.). For example, on an SGI irix64 platform with the not_spawn starting mode (see section 3.1), one can type:

```
> mpirun -np 1 ../../bin/prismdrv_main :  -np 3 ./toyatm.exe :
-np 1 ./toyoce.exe :  -np 3 ./toylan.exe
```

where the -np x specify the number of processes for each application. Launching on other platforms are described in PRISM_Cpl/util/prismrun.

After a successfull execution of the example proto_ex, all files containing standard output from the different components should end with lines like

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

 ### Model: ... : No errors detected
```

6.5 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 PRISM_Cpl/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 by PRISM to place the MPI processes of an application into a MPI_COMM_WORLD which is different for each application. 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 3.1 and 6.4) 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.

Intel Fortran Compiler

To sucessfully compile OASIS4 Intel Fortran Compiler version 8.0 or higher is required.

Chapter 7

Scalability with OASIS4

One of the major enhancements of OASIS4 compared to OASIS3 is the full parallelization of the PSMILe (see section 4.5.1) and the Transformer (see section 3.2).

The simple-mg toy model in directory PRISM_Cpl/examples (see also section A.1), with a T106 resolution for the atmosphere model was selected for scalability tests.

Selected platforms were NEC SX-6, SGI Altix and Origin, AMD-Athlon PC Cluster, and AMD-Opteron PC Cluster. Table 7.1 summarizes the characteristics of the tested systems and used software.

Model/Feature	CPU specs	Main Memory (per CPU)	Compiler	MPI-library
NEC SX-6	0,5 GHz (*16)	8 GB	F90: Rev. 285 C++: Rev.063	NEC-MPI LC310039
SGI ALTIX	Madison 1,5 Ghz 6 MB L3-cache	2 GB	Intel ifort 8.050 Intel icc 8.069	SGI MPT 1.12
SGI-Origin	R14000 0,7 Ghz 8 MB L2-cache	2 GB	MIPSPro 7.4.1	SGI MPT 1.12
AMD-Athlon PC	2,8 GHz, 32 bit	4 GB	Absoft 32bit F95 9.0 r2	MPICH- Myrinet
AMD-Opteron PC	2,2 GHz, 64 bit	4 GB	Pathscale 1.4.1	LAM 7.1.1

Table 7.1: Characteristics of the tested systems and used software for scalability tests

Simulation with up to 24 CPUs were carried out, starting with one process for each component model and the Transformer (1-1-1) and ending with 8 processes per component model and the Transformer (8-8-8). The notation in the result tables below is X-Y-Z where X, Y, and Z are respectively the number of processes for the atmosphere toy model, for the Transformer, and for the ocean toy model. For example: 4-1-4 means 4 processes for each component model and 1 processes for the Transformer.

Two measures of the scalability is taken in each simple-mg toy component model:

- the time in seconds until the prism_enddef is reached. This measure is reported in the columns 'enddef ATM' and 'enddef OCE' for respectively the atmosphere and the ocean component model in the tables below. The subroutine prism_enddef (see section 4.5.1) finishes the definition phase and includes the parallel neighborhood search for the interpolation done in parallel by the PSMILe linked to the models. Increasing the number of processes for the component models should therefore reduce this time. This is a measure of the PSMILe scalability.
- the required time in seconds for a ping-pong exchange of data with the other component. This measure is reported in the columns 'ping-pong ATM' and 'ping-pong OCE' for respectively the atmosphere and the ocean component model in the tables below. As the data are transfered in parallel by the PSMILe and interpolated by the Transformer, increasing the number of processes

for the Transformer and for the component models should reduce this time. This is a measure of the Transformer and of the PSMILe scalability.

SX6	enddef ATM	enddef OCE	ping-pong ATM	ping-pong OCE
1-1-1 (1 node)	0.6	0.6	0.9	1.0
2-2-2 (1 node)	0.4	0.3	0.5	0.5
4-4-4 (2 nodes)	0.5	0.5	2.4	2.6

Table 7.2: Scalability results for simple-mg on NEC SX-6.

SGI	enddef ATM	enddef OCE	ping-pong ATM	ping-pong OCE
1-1-1	3.5	1.8	1.1	1.9
2-2-2	1.6	1.3	0.6	1.0
4-4-4	0.7	0.6	0.3	0.5
4-1-4	1.0	0.9	0.8	0.8

Table 7.3: Scalability results for simple-mg on SGI-Altix.

SGI	enddef ATM	enddef OCE	ping-pong ATM	ping-pong OCE
1-1-1	9.6	5.5	2.6	5.4
2-2-2	6.0	3.6	1.4	3.0
4-4-4	1.3	2.9	0.8	1.8
8-8-8	0.7	0.6	0.3	0.7
8-1-8	1.0	1.0	1.6	2.5

Table 7.4: Scalability results for simple-mg on SGI-ORIGIN.

AMD	enddef ATM	enddef OCE	ping-pong ATM	ping-pong OCE
1-1-1	4.6	4.6	1.9	4.1
2-2-2	2.3	2.5	0.9	2.0
4-4-4	1.0	3.2	0.9	1.2
4-1-4	1.4	1.1	2.4	2.8

Table 7.5: Scalability results on AMD Athlon-Cluster.

AMD	enddef ATM	enddef OCE	ping-pong ATM	ping-pong OCE
1-1-1	1.5	1.6	0.6	1.1
4-4-4	1.1	1.0	0.2	0.3
8-8-8	0.4	0.5	0.1	0.2
8-1-8	0.7	0.6	0.6	0.7

Table 7.6: Scalability results on AMD Opteron-Cluster.

In general the elapsed times are in the order of seconds for the simple-mg. Nevertheless scalability of OASIS4 PSMILe and Transformer can be demonstrated by comparing the 'enddef' or 'ping-pong' times for configurations 1-1-1, 2-2-2, 4-4-4, and 8-8-8 (when available). This time decreases on all platforms with the number of processes used (the only exceptions are the 2-2-2 and 4-4-4 cases on the SX-6 which can be easily explained as the 2-2-2 run is done on one node only whereas the 4-4-4 run involves 2 nodes, and the 'enddef OCE' time for the AMD Athlon PC Cluster for 4-4-4 on table 7.5).

At the end of each table, the numbers for the 4-1-4 or 8-1-8 configuration are also given. This number illustrated the necessity of having a parallel Tranformer; in fact, the ping-pong tests realised with only 1 process for the Transformer (4-1-4 or 8-1-8 configuration) show an elapse time which is up to 3 times larger than the ping-pong tests realised with 4 or 8 processes for the Transformer (4-4-4 or 8-8-8 configuration).

The parallelization of the OASIS4 gives therefore big advantages in case of expensive interpolations between component fields exchanged between highly parallel component models. The parallel neighbourhood search in the PSMILe library as well as the parallel Transformer reduce interpolation time as well as communication time.

Appendix A

Toy coupled models with OASIS4

A.1 General description

Different toy models were created to test OASIS4 functionality. A toy coupled model is based on component models with no real physics but reproducing realistic coupling and I/O exchanges (parallel decomposition, size and number of coupling fields, interpolation and other operations, frequencies of coupling or I/O, etc.).

The sources of the different OASIS4 toy coupled models can be found in `PRISM_Cpl/examples` subdirectories. Chapter 6 briefly describes how to compile and run these toy coupled models. For each toy coupled model, the coupling and I/O configuration is defined in the SCC and SMIOC XML files in the related directory (see also chapter 5). The following toy models are available:

- `proto_ex`:

  In this example, three toy applications anticipating an atmosphere model (toyatm), an ocean model (toyoce), and a land model (toylan) are coupled. The proto_ex toy coupled model, as well as its descriptive and configuration XML files, are described in more detail in section A.2 below.

- `simple-mg`:

  The example couples two toy applications anticipating an atmosphere model (appl-atm.F90) and an ocean model (appl-ocn.F90). The atmosphere model works on a Gaussian grid while the ocean model is using an isotropic grid. Different resolutions are possible using CPP flags for the ATM (T21, T42, T63, T106 and T255). The default is T63. The resolution of the ocean model can be modified by changing the variable dy_lat in appl-ocn.F90. The default value is 4.0. The exchanged data are scalar fields. A trilinear interpolation is performed between the two different grids.

- `simple-mg-io-wrt`:

  The example is derived from the example `simple.mg` and features how to write data into a NetDF file using `prism_put` . The atmosphere model declares and sends one additional field; in its SMIOC file `atm_atm_smioc.xml` it is specified that this field should be written to a file `atm_wind.nc`.

- `simple-mg-io-wrt2`:

  The examples is like `simple-mg`, but tests the transfer of a field to two destinations. The atmosphere model declares the field `atm_wind` and sends it through one `prism_put` call. As is specified in its SMIOC file `atm_atm_smioc.xml`, the field is transfered both to the ocean model (via the Transformer) and to the file `atm_wind.nc`. Moreover, for the field `ocn_wind2` sent by the ocean model, the debug mode is turned on; this means that at run-time, the field is automatically written to the file 'ocn_wind_getg_out.1994-01-01T00:00:00.nc' below the `prism_put` .

- `simple-mg-io-rd`:

The example is similar to simple-mg but now each model loops over 12 time steps. In the atmosphere model, one additional field atm_wind_input is read from input files; this example tests the capability of the PSMILe to read data which are spread over a sequence of files comprising the different time stamps needed by a coupled experiment. Here the sequence consists of three files atm_wind_in.1993-09-01T00:00:00.nc, atm_wind_in.1994-01-01T00:00:00.nc and atm_wind_in.1994-05-01T00:00:00.nc. According to the job start date, the PSMILe find automatically the best match as file atm_wind_in.1994-01-01T00:00:00.nc.

Note that resolutions of the atmosphere and ocean models must not be changed in this directory without changing the input files. The example simple-mg-io-forrd can be run to create the input files (see below).

- simple-mg-io-forrd:

  This example is the counterpart of simple-mg-io-rd. The atmosphere model writes 12 time stamps of a transient into a file atm_wind_out.1994-01-01T00:00:00.nc. That file can be renamed to atm_wind_in.1994-01-01T00:00:00.nc and can be used for the example simple-mg-io-rd. Moreover, the example shows how the PSMILe performs a file name extension if a file basename is given and the suffix extension is turned on (see section 5.4.7).

A.2 The proto_ex toy coupled model

A.2.1 The proto_ex toy coupled model general description

The proto_ex toy coupled model (see sources in PRISMCpl/examples/proto_ex) illustrates the coupling and I/O of three applications toyatm.F90, toylan.F90, and toyoce.F90. The coupling and file I/O managed by OASIS4 Driver, Transformer and PSMILe library linked to the 3 component models is illustrated on figure A.1.

Both toyatm and toylan 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, toyoce, is not parallel and uses a a real ocean model cartesian, stretched and rotated grid of 182X149 grid points.

All coupling and I/O fields are scalar fields. The model toyatm declares 1 input field SISUTESU, and 4 output field CONSFTOT, COSENHFL, COWATFLU, ATWINSTS as is listed in its PMIOD file toyatm_atmos_pmiod.xml (see section A.2.6). The model toylan declares 2 input field LAWATFLX and SOSENHFL, and 1 output field LARUNOFF as is listed in its PMIOD file toylan_land_pmiod.xml (see section A.2.7). The model toyoce declares 4 input field SONSHLDO, SOWAFLDO, SORUNOFF and OCWINSTS, and 1 output field SOSSTSST as is listed in its PMIOD file toyoce_ocean_pmiod.xml (see section A.2.8).

Chapter 6 briefly describes how to compile and run the proto_ex toy coupled model. In the not_spawn mode, each application (i.e. prismdrv_main, toyatm.exe, toyoce.exe, and toylan.exe) must be started with the number of processes specified in the scc.xml file (see section A.2.5). In the spawn mode, only the prismdrv_main needs to be started and will automatically spawn the other applications on the number of processes specifies in the scc.xml.

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

In the toyatm SMIOC file toyatm_atmos_smioc.xml (see section A.2.9), it is specified that ATWINSTS will be sent to toyoce, COSENHFL to toylan, COWATFLU both to toyoce and toylan, while CONSFTOT is not sent at all; it is also specified that SISUTESU will come from toyoce. The toylan SMIOC file toylan_land_smioc.xml (see section A.2.10) specifies that LARUNOFF will both go to toyoce and be written to a file LARUNOFF.nc and that LAWATFLX and SOSENHFL will be received from toyatm. Finally, in the toyoce SMIOC file toyoce_ocean_smioc.xml (see section A.2.11), it is specified that

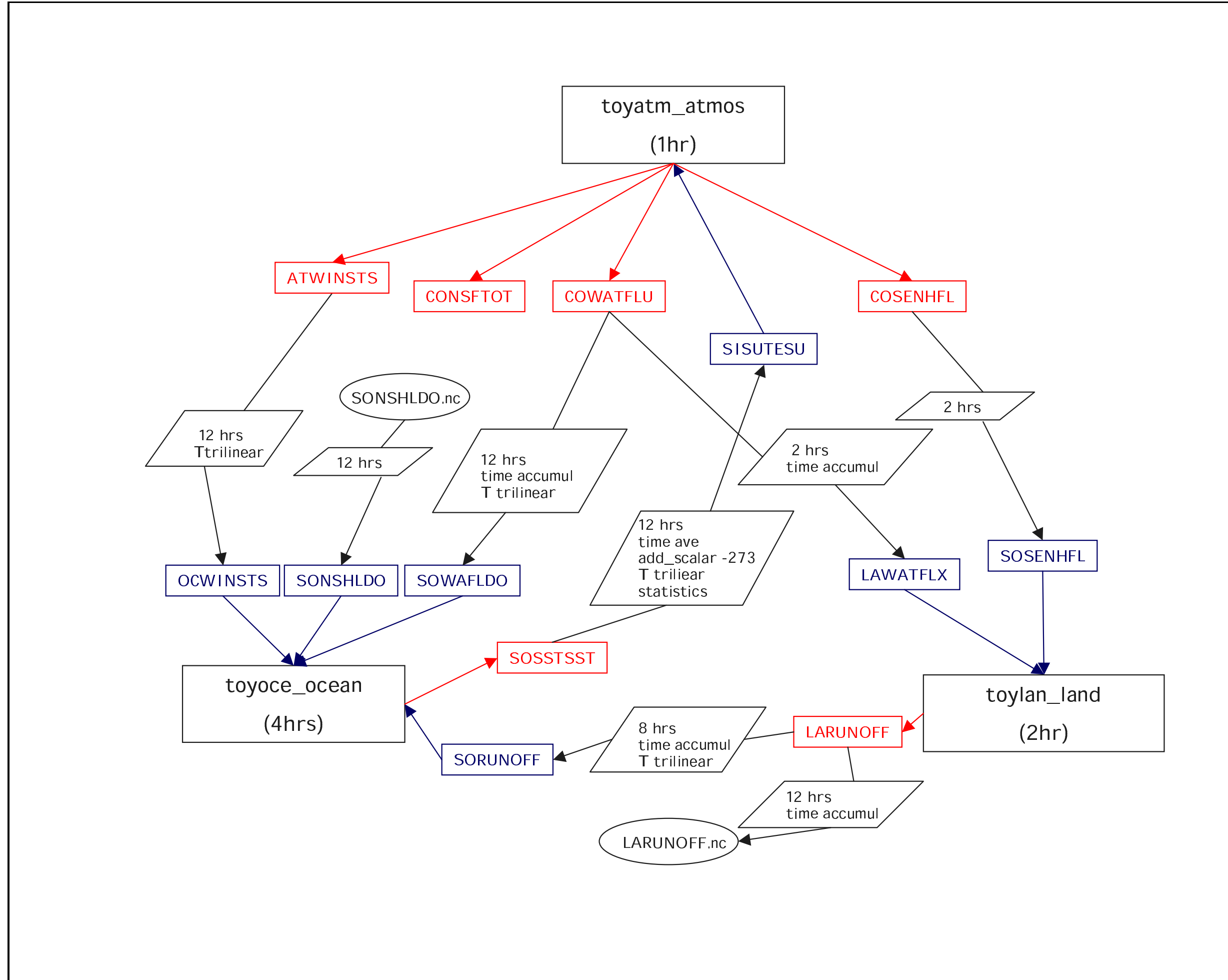

Figure A.1: The `proto_ex` toy coupled model coupling and I/O confi guration

OCWINSTS and SOWAFLDO will be received from toyatm, SORUNOFF from toylan, while SONSHLDO will be read from a file SONSHLDO.nc; SOSSTSST will be sent to toyatm.

Different operations are performed by the PSMILe 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 toyatm and toylan (and vice-versa) are direct, involving possibly some repartitioning if their parallel partitioning are different. As toyatm and toyoce do not have the same grid, their exchanges of coupling fields go through the Transformer (not illustrated on figure A.1) where a linear interpolation is performed. The different coupling and I/O periods are also specified in the different SMIOC files.

The example proto_ex features also the restart of a coupled experiment in which a coupling field is defined with a lag (see sections 4.6.4 and 5.4.4 for more details).

To create a restart at the first run, type:

```
prismrun -<system> -f scc.xml.start -conf <myfile.conf>
```

After the first 3-day run, the file COSENHFL_atmos_toyatm_rst.2000-01-04T00:00:00.nc is created and can be used to restart a second run of 3 days. To do this you need to change the COSENHFL lag defined in toyatm_atmos_smioc.xml from zero to one and use the scc.xml which is prepared for the second 3-day run:

```
prismrun -<system> -f scc.xml.restart -conf <myfile.conf>
```

The COSENHFL field with a positive lag of one will automatically be read from its coupling restart file at the beginning of the run below the first prism_get and will automatically be written out to a file COSENHFL_atmos_toyatm_rst.2000-01-07T00:00:00.nc below the last prism_put of the run. This can again be used for a subsequent run.

A.2.2 The proto_ex toyatm AD XML file

The AD XML file of proto_ex toyatm can be found at http://www.cerfacs.fr/PRISM/XML/toyatm_ad.xml

A.2.3 The proto_ex toylan AD XML file

The AD XML file of proto_ex toylan can be found at http://www.cerfacs.fr/PRISM/XML/toylan_ad.xml

A.2.4 The proto_ex toyoce AD XML file

The AD XML file of proto_ex toyoce can be found at http://www.cerfacs.fr/PRISM/XML/toyoce_ad.xml

A.2.5 The proto_ex toy coupled model SCC XML file

The SCC XML file of proto_ex can be found at http://www.cerfacs.fr/PRISM/XML/scc.xml

A.2.6 The proto_ex toyatm PMIOD XML file

The PMIOD XML file of proto_ex toyatm can be found at http://www.cerfacs.fr/PRISM/XML/toyatm_atmos_pmiod.xml

A.2.7 The proto_ex toylan PMIOD XML file

The PMIOD XML file of proto_ex toylan can be found at http://www.cerfacs.fr/PRISM/XML/toylan_land_pmiod.xml

A.2.8 The proto_ex toyoce PMIOD XML file

The PMIOD XML file of proto_ex toyoce can be found at http://www.cerfacs.fr/PRISM/XML/toyoce_ocean_pmiod.xml

A.2.9 The proto_ex toyatm SMIOC XML file

The SMIOC XML file of proto_ex toyatm can be found at http://www.cerfacs.fr/PRISM/XML/toyatm_atmos_smioc.xml

A.2.10 The proto_ex toylan SMIOC XML file

The SMIOC XML file of proto_ex toyatm can be found at http://www.cerfacs.fr/PRISM/XML/toylan_land_smioc.xml

A.2.11 The proto_ex toyoce SMIOC XML file

The SMIOC XML file of proto_ex toyoce can be found at http://www.cerfacs.fr/PRISM/XML/toyoce_ocean_smioc.xml

Appendix B

DTDs and Schemas of OASIS4 description and configuration files

B.1 AD DTD

The AD DTD can be found at http://www.cerfacs.fr/PRISM/XML/ad.dtd

B.2 SCC DTD

The SCC DTD can be found at http://www.cerfacs.fr/PRISM/XML/scc.dtd

B.3 PMIOD DTD and Schema

B.3.1 PMIOD DTD

The PMIOD DTD can be found at http://www.cerfacs.fr/PRISM/XML/pmiod.dtd

B.3.2 PMIOD Schema

The PMIOD Schema can be found at http://www.cerfacs.fr/PRISM/XML/pmiod.xsd

B.4 SMIOC DTD and Schema

B.4.1 SMIOC DTD

The SMIOC DTD can be found at http://www.cerfacs.fr/PRISM/XML/smioc.dtd

B.4.2 SMIOC Schema

The SMIOC DTD can be found at http://www.cerfacs.fr/PRISM/XML/smioc.xsd

Bibliography

[1] Ahrem, R. et al., 2003: MpCCI Mesh-based parallel Code Coupling Interface, Specification Version 2, Fraunhofer Institute for Algorithms and Scientific Computing. Sankt Augustin, Germany.

[2] Balaji, 2001: Parallel Numerical Kernels for Climate Models, ECMWF TeraComputing Workshop 2001, World Scientific Press, Reading.

[3] Buja, L. and T. Craig, 2002: Community Climate System Model CCSM 2.0.1 User Guide, National Center of Atmospheric Research, Boulder, CO.

[4] Carril, A., R. Budich, S. Valcke, 2004: The TOYCLIM demonstration run report, PRISM Report Series No 13.

[5] Eaton, B., Gregory, J., Drach, B., Taylor, K., and Hankin, S.PMEL, 2003: NetCDF Climate and Forecast (CF) Metadata Conventions, http://www.cgd.ucar.edu/cms/eaton/cf-metadata/index.html

[6] Gropp, W., S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir, 1998: MPI – The Complete Reference, Vol. 2 The MPI Extensions, MIT Press.

[7] Legutke, S. and V.Gayler, 2004: The PRISM Standard Compilation Environment, PRISM Report Series No 4.

[8] Li, J., W. Liao, A. Choudhary, R. Ross, R. Thakur, R. Latham, A. Siegel, B. Gallagher, M. Zingale, 2003: Parallel NetCDF: A High-performance Scientific IO Interface, Proceedings of the SC'03, Nov 15-21, Phoenix, Arizona, USA. http://www-unix.mcs.anl.gov/parallel-netcdf

[9] Rew, R. K. and G. P. Davis, 1997: Unidata's netCDF Interface for Data Access: Status and Plans, Thirteenth International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology, Anaheim, California, American Meteorology Society.

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

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

[12] http://www.gfdl.noaa.gov/~fms/pubrel/j/mom4/doc/mom4_manual.html

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

Index

prismrun, 48
32 Bit architectures, 50
64 Bit architectures, 50

AD, 32, **33**
  argument, 33
  coupling_mode, 33
  local_name, 33
  long_name, 33
  nbr_procs, 33
  start_mode, 33
Averaging, 7

Calendar, 29
Collective calls, 8, 23, 28
Compiling, **47**

Driver, 8
DTD, 32

EPIO, 23
EPIOS, 23
EPIOT, 23

Gathering, 7
Global index space, 18
Grid types, 13, 14, 37

Id, 7
Identifi cators, 7
Initialisation, 8
Internal communication, 9

Lag, 24, 27, 34, 41, 58
LAM-MPI, 49
Local index space, 18

MPI, 7–9, 28
  MPI_Comm_Spawn, 50
  MPI_Comm_Spawn_Multiple, 49
  MPI_Finalize, 8, 28
  MPI_Init, 8, 28
  LAM-MPI, 49
  MPICH, 50
MPI communicator, 9
MPI-IO, 24
MPICH, 50
mpp_io, 24

Neighborhood search, **23**

Partition, 18
PMIOD, 32
PRISM API
  prism_abort, **28**
  prism_calc_newdate, **29**
  prism_def_grid, **12**
  prism_def_partition, **18**
  prism_def_var, **21**
  prism_enddef, **23**
  prism_error, **29**
  prism_get, **26**
  prism_get_calendartype, **29**
  prism_get_local_comm, **9**
  prism_get_nb_ranklists, **10**
  prism_get_ranklists, 10, **10**
  prism_get_real_kind_type, **30**
  prism_init, 8, **8**
  prism_init_comp, 8, **8**, 9, 10
  prism_initialized, **9**
  prism_put, **25**
  prism_put_inquire, **26**
  prism_put_restart, **27**
  prism_remove_mask, **30**
  prism_set_angle, **20**
  prism_set_corners, **14**
  prism_set_mask, **16**
  prism_set_points, 7, **19**
  prism_set_scalefactors, **15**
  prism_set_subgrid, **17**
  prism_set_vector, **20**
  prism_set_vectormask, **16**
  prism_terminate, **28**
  prism_terminate, 8
  prism_terminated, **28**
  prism_version, **30**
PRISM derived data type
  PRISM_Time_Struct, 29
PRISM derived data types, 8
PRISM Parameter
  PRISM_gridless, 13, 14, 37
  PRISM_irrlonlat_sigmavrt, 13, 14, 37
  PRISM_irrlonlatvrt, 13, 14, 37
  PRISM_reglonlat_sigmavrt, 13, 14, 37
  PRISM_reglonlatvrt, 13, 14, 37
  PRISM_unstructlonlat_regvrt, 13, 14, 37
  PRISM_unstructlonlat_sigmavrt, 13, 14, 37
  PRISM_unstructlonlatvrt, 13, 14, 37

Proleptic Gregorian Calendar, 29

Restart, 27, 58
Running, **47**

Scattering, 7
SCC, 32
- application, 34
- component, 34
- coupling_mode, 34
- driver, 34
- end_date, 34
- executable_name, 34
- host, 34
- local_name, 34
- long_name, 34
- nbr_procs, 34
- ranks, 34
- redirect, 34
- run, 34
- start_date, 34
- start_mode, 34

SCC XML, 8
SMIOC, 32
- code, 36
- compute_space, 38
- contact, 37
- documentation, 37
- fi le, 45
- fi le adding, 45
- fi le fi ll value, 45
- fi le format, 45
- fi le io_mode, 45
- fi le name, 45
- fi le packing, 45
- fi le scaling, 45
- fi le suffi x, 45
- Fortran_units, 37
- grid, 37
- grid formula_terms, 37
- grid grid_name, 37
- grid latitude_dimension, 37
- grid local_name, 37
- grid long_name, 37
- grid longitude_dimension, 37
- grid physical_space, 37
- grid pole_covered, 37
- grid positive, 37
- grid sampled_space, 37
- grid units, 37
- grid units_standard_name, 37
- grid valid_max, 37
- grid valid_min, 37
- grid vertical_dimension, 37
- grid_type, 37
- indexing_dimension, 37
- input, 42
- intent, 40
- interpolation, 42, 43
- laboratory, 36
- lag, 41, 58
- local_name, 36
- long_name, 36
- output, 40
- simulated, 36
- transient, 39

SMIOC XML, 8

Time lag, 24

XML, **32**
- Attribute, 32
- Element, 32
- Tag, 32