Description de l'interface fortran

1. Initialisation et finalisation de la bibliothèque

```
! Initialisation à partir d'un fichier
SUBROUTINE xml_parse_file(filename)
   CHARACTER(len = *), INTENT(IN) :: filename
END SUBROUTINE xml_Parse_File

! Initialisation à partir d'une chaîne de caractères
SUBROUTINE xml_parse_string(xmlcontent)
   CHARACTER(len = *), INTENT(IN) :: xmlcontent
END SUBROUTINE xml_Parse_String

! Finalisation
SUBROUTINE dtreatment_end(context_hdl)
   TYPE(XHandle), INTENT(IN), VALUE :: context_hdl
END SUBROUTINE dtreatment_end
```

L'initialisation de la bibliothèque XMLIOSERVER consiste à créer une arborescence d'objets dans la mémoire de l'application qui va permettre d'interpréter les souhaits des utilisateurs quant aux types de fichiers à sortir et à leur contenu en terme de champs de données et de grilles. Pour ce faire, il existe deux procédures:

SUBROUTINE xml_parse_file
Permet de parser un document xml depuis un fichier dont le chemin est indiqué en argument.

SUBROUTINE xml_parse_string
Permet de parser un document xml depuis une chaîne de caractères.

La finalisation est obtenue par l'appel du sous-programme ci-dessous pour chacun de contextes en cours d'utilisation. Les communications entre processus sont alors suspendues, les tampons sont progressivement vidés et les fichiers de données sont fermés.

SUBROUTINE dtreatment_end
Termine le traitement des données dans un contexte déterminé par le handle[1] fourni en entrée de la procédure.

2. Résolution des héritages et création des fichiers

```
! Lancement du traitement des données
SUBROUTINE dtreatment_start(context_hdl, filetype, comm_client_server)
   TYPE(XHandle), INTENT(IN)           :: context_hdl
   INTEGER      , INTENT(IN), OPTIONAL:: filetype, comm_client_server
END SUBROUTINE dtreatment_start
```

SUBROUTINE dtreatment_start
Débute le traitement des données par l'analyse de l'arborescence d'objet initialement créée, la résolution des héritages et la création des fichiers de données et de leurs entêtes. Dès lors que cette étape est terminée, l'utilisateur ne peut plus changer la configuration sans provoquer d'erreurs; l'écriture de données et la modification du calendrier sont les seules actions réalisables à partir du code Fortran par la suite. L'argument *context_hdl* est le handle sur le contexte à traiter, *filetype* est l'identifiant du type de fichier à sortir et *comm_client_server* est le communicateur MPI commun au client et au serveur.

1 Identifiant d'un objet en mémoire en vue d'une réutilisation ultérieure

3. Modification de l'arborescence d'objet

```
! enum XCalendarType
INTEGER, PARAMETER  :: D360   = 0, ALLLEAP   = 1, NOLEAP   = 2,
INTEGER, PARAMETER  :: JULIAN = 3, GREGORIAN = 4

! enum XDType
INTEGER, PARAMETER :: ECONTEXT = 4
INTEGER, PARAMETER :: EAXIS = 5 , EDOMAIN = 6 , EFIELD = 7
INTEGER, PARAMETER :: EFILE = 8 , EGRID = 9
INTEGER, PARAMETER :: GAXIS = 10, GDOMAIN = 11, GFIELD = 12
INTEGER, PARAMETER :: GFILE = 13, GGRID = 14

! struct XDate
TYPE XDate
   INTEGER :: year, month, day, hour, minute, second
END TYPE Xdate

! struct XDuration
TYPE XDuration
   REAL :: year, month, day, hour, minute, second
END TYPE Xduration

! Ajout d'élément à l'arborescence
SUBROUTINE xml_tree_add(parent_hdl, parent_type, child_hdl, child_type,
child_id)
   TYPE(XHandle)                , INTENT(IN) :: parent_hdl
   TYPE(XHandle)                , INTENT(OUT):: child_hdl
   INTEGER                      , INTENT(IN) :: child_type, parent_type
   CHARACTER(len = *), OPTIONAL, INTENT(IN) :: child_id
END SUBROUTINE xml_tree_add

! Création de contexte
SUBROUTINE context_create(context_hdl, context_id, calendar_type, init_date)
   TYPE(XHandle)        , INTENT(OUT) :: context_hdl
   CHARACTER(len = *)   , INTENT(IN)  :: context_id
   INTEGER              , INTENT(IN)  :: calendar_type
   TYPE(XDate), OPTIONAL, INTENT(IN)  :: init_date
END SUBROUTINE context_create

! Définition des attributs d'un élément de type <elem>
SUBROUTINE set_<elem>_attributes(<elem>_hdl, ftype, liste d'attributs ...)
   TYPE(XHandle)      , INTENT(IN) :: <elem>_hdl
   INTEGER            , INTENT(IN) :: ftype
   TYPE_ATT1, OPTIONAL, INTENT(IN) :: att1_
   TYPE_ATT2, OPTIONAL, INTENT(IN) :: att2_
   TYPE_ATT3, OPTIONAL, INTENT(IN) :: att3_
   TYPE_ATT4, OPTIONAL, INTENT(IN) :: att4_
   TYPE_ATT5, OPTIONAL, INTENT(IN) :: att5_
   ...
END SUBROUTINE  set_<elem>_attributes

! Obtention d'un attribut <attrib> de l'élément de type <elem>
SUBROUTINE get_<elem>_<attrib>(<elem>_hdl, ftype, <attrib>_)
   TYPE(XHandle), INTENT(IN)  :: <elem>_hdl
   INTEGER      , INTENT(IN)  :: ftype
   TYPE_ATT     , INTENT(OUT) :: <attrib>_
END SUBROUTINE get_<elem>_<attrib>
```

SUBROUTINE xml_tree_add
Ajoute un élément à un autre élément de type différent, par exemple un domaine à un groupe de domaines ou encore, un champ à un fichier.
Les arguments à transmettre à ce sous-programme sont les suivants:

- *parent_hdl*: le handle sur l'élément parent;
- *parent_type*: le type de l'élément parent parmi les valeurs de l'énumération *XDType*;
- *child_hdl*: le handle sur l'élément enfant que l'on récupère en sortie;
- *child_type: le type de l'élément enfant (XDType);*
- *child_id (optionnel)*: l'identifiant de l'élément sous forme de chaîne de caractère.

SUBROUTINE context_create
Ajoute un contexte en spécifiant son identifiant, le type de calendrier qui lui est associé et une date initiale. Un handle sur le contexte est reçu en fin d'appel.
Les éléments de définition sont créés automatiquement pour le nouveau contexte.

SUBROUTINE set_<elem>_attributes
(<elem>: field, context, file, axis, domain, grid)
Modifie les attributs d'un élément en indiquant le handle et le type de celui-ci, suivis des valeurs souhaitées pour ses attributs.

SUBROUTINE get_<elem>_<attrib>
(<elem>: field, context, file, axis, domain, grid)
(<attrib>: nom de l'attribut)
Accède à la valeur de l'attribut d'un élément en indiquant le handle et le type de celui-ci, suivis de la variable où stocker la valeur.

4. Changements de contextes

```
! Changement de contexte
SUBROUTINE context_set_current(context, withswap)
   TYPE(XHandle)               , INTENT(IN) :: context
   LOGICAL (kind = 1), OPTIONAL, INTENT(IN) :: withswap
END SUBROUTINE context_set_current
```

SUBROUTINE context_set_current
Détermine le contexte de travail en spécifiant son handle.
L'argument *withswap* permet de maintenir l'identifiant du contexte dans une pile afin de le récupérer simplement par la suite.

5. Obtention de handle

```
! Récupération d'un handle
SUBROUTINE handle_create(hdl, dtype, id)
   TYPE(XHandle)     , INTENT(OUT) :: hdl
   INTEGER           , INTENT(IN)  :: dtype
   CHARACTER(len = *), INTENT(IN)  :: id
END SUBROUTINE handle_create
```

SUBROUTINE handle create
Accède au handle d'un élément dont le type et l'identifiant sous forme de chaîne de caractères sont spécifiés.

6. Envoi des données et mises à jour du calendrier

```
! Modification du pas de temps
SUBROUTINE set_timestep(timestep)
   TYPE(XDuration), INTENT(IN):: timestep
END SUBROUTINE set_timestep

! Mise à jour du pas de temps
SUBROUTINE update_calendar(step)
   INTEGER, INTENT(IN) :: step
END SUBROUTINE update_calendar

! Envoi de données
SUBROUTINE write_data (fieldid, data)
   CHARACTER(len = *), INTENT(IN) :: fieldid
   REAL, DIMENSION(*), INTENT(IN) :: data
END SUBROUTINE
```

SUBROUTINE set_timestep
Modifie la durée pour l'incrémentation du pas sur l'axe temporel.

SUBROUTINE update_calendar
Permet de mettre à jour le calendrier du contexte courant.

SUBROUTINE write_data
Envoie les données d'un champ en spécifiant son identifiant et les données sous la forme d'un tableau de réels de 1 à 3 trois dimensions en simple ou double précision.

7. Accès aux variables dynamiques

```
! Changement de répertoire
SUBROUTINE change_directory (path)
   CHARACTER(len = *), INTENT(IN)  :: path
END SUBROUTINE change_directory

! Récupération du répertoire courant
SUBROUTINE change_directory (path)
   CHARACTER(len = *), INTENT(OUT) :: path
END SUBROUTINE change_directory

! Récupération de valeur de variable
SUBROUTINE get_variable (id, value)
   CHARACTER(len = *), INTENT(IN)  :: id
   TYPE_VAR          , INTENT(OUT) :: value
END SUBROUTINE get_variable
```

SUBROUTINE change_directory
Change le répertoire où récupérer les variables dans le contexte courant.

SUBROUTINE get_directory
Accède au chemin du répertoire courant dans l'arborescence de définition des variables.

SUBROUTINE get_variable
Accède à la valeur d'une variable dans le contexte et le répertoire actifs en indiquant son identifiant.

8. Exemples simples d'utilisation

ETAPE DE CONFIGURATION

```
1.  SUBROUTINE FAKE_ENTRY(comm_client, comm_client_grp, comm_client_server) BIND (C)
2.     INTEGER(kind = C_INT), INTENT(IN), VALUE ::        &
3.           comm_client, comm_client_grp, comm_client_server
4.     TYPE(XDate)            :: init_date  = XDate(1985, 03, 15, 17, 35, 00)
5.     TYPE(XHandle)          :: style_ctxt = NULLHANDLE
6.     TYPE(XHandle)          :: temp_mod   = NULLHANDLE,  &
7.                               temp_mod_  = NULLHANDLE,  &
8.                               temp_mod__ = NULLHANDLE
9.
10. ! Parsing du document xml de définition à  partir d'une chaîne de caractères.
11. CALL xml_parse_string ("<? xml version=1.0 ?><simulation></simulation>")
12.
13. ! On crée un nouveau context et on lui associe un handle.
14. CALL context_create(context_hdl   = style_ctxt, context_id  = "mon_context", &
15.                     calendar_type = GREGORIAN,  init_date = init_date)
16.
17. ! Récupération d'un handle sur le groupe de définition des champs.
18. CALL handle_create(temp_mod, GFIELD,  "field_definition")
19.
20. ! Ajout d'un groupe de champs anomyme
21. CALL xml_tree_add(parent_hdl = temp_mod,  parent_type = GFIELD, &
22.                   child_hdl  = temp_mod_, child_type  = GFIELD)
23.
24. ! Définition des attributs du groupe de champs
25. CALL set_field_attributes(field_hdl = temp_mod_, ftype = GFIELD, &
26.                           unit_  = "SI",  prec_  = 8)
27.
28. ! Ajout d'un groupe de champ identifié
29. CALL xml_tree_add(parent_hdl = temp_mod_, parent_type = GFIELD, &
30.                   child_hdl  = temp_mod,  child_type  = GFIELD, &
31.                   child_id   = "field_enabled")
32.
33. ! Définition des attributs du groupe de champs
34. CALL set_field_attributes(field_hdl  = temp_mod,  &
35.                           ftype      = GFIELD,    &
36.                           operation_ = "instant", &
37.                           enabled_   = .TRUE._1,  &
38.                           freq_op_   = "1h")
39.
40. ! Ajout d'un champ nommé «champ_2D_k8_inst»
41. CALL xml_tree_add(parent_hdl  = temp_mod,   parent_type = GFIELD, &
42.                   child_hdl   = temp_mod__, child_type  = EFIELD, &
43.                   child_id    = "champ_2D_k8_inst")
44.
45. ! Définition des attributs du champ
46. CALL set_field_attributes(field_hdl      = temp_mod__,   &
47.                           ftype          = EFIELD,       &
48.                           name_          = "champ1",     &
49.                           standard_name_ = "lechamp1",   &
50.                           long_name_     = "le champ 1", &
51.                           domain_ref_    = "simple_domaine0")
52.
53. ! ... Début du traitement ...
```