SANGOMA: Stochastic Assimilation for the Next Generation Ocean Model Applications EU FP7 SPACE-2011-1 project 283580

Deliverable 1.3:
Specification of Data Model
Due date: 05/01/2012
Delivery date: 06/26/2012
Delivery type: Report, public

SANGOMA

Lars Nerger
Alfred-Wegener-Institute, GERMANY

Arnold Heemink Nils van Velzen
Martin Verlaan
Delft University of Technology, NETHERLANDS

Jean-Marie Beckers Alexander Barth
University of Liège, BELGIUM

Peter Jan Van Leeuwen
University of Reading, UK

Pierre Brasseur Jean-Michel Brankart
CNRS-LEGI, FRANCE

Pierre de Mey
CNRS-LEGOS, FRANCE

Laurent Bertino
NERSC, NORWAY

Contents

Chapter 1

Introduction

This document specifies the data model that will be used for the implementation of SANGOMA tools. The data model is initial specified as a logical data model. However, also some considerations of the physical realization are described in this document. The data model bases on the data models that are already present in the different assimilation systems used within the SANGOMA consortium. In addition, the data models of the MyOcean initiative are taken into account. The data model bears in mind that implementations in Fortran, Matlab/Octave as well as in C and Java must be possible, while for SANGOMA the coding will be in Fortan and Matlab/Octave. In addition, the interoperability of the tools implemented in Fortran with the other programming languages is ensured. The data model then aims at efficiency in terms of memory use and accessibility.

The physical realization will consider the realization of the data model in the different programming languages as well as in NetCDF, which is used as a standard file format within SANGOMA.

The data assimilation systems used by the members of the SANGOMA consortium, show clear variations in the data model. They range from an abstract object-oriented data model in which data is only indirectly accesses via functions ("methods"), over Fortran interfaces using derived data types and assumed shape arrays, to rather basic interfaces relying only on arrays and scalar variables. The discussions during the preparation of the SANGOMA data model showed that a data model using basic data types should be preferred for the tools developed within SANGOMA. In combination with the C-binding of Fortran2003, this data model ensures the interoperability with other programming languages like C. In addition, the Fortran programmers, who build the majority in SANGOMA can base on a typical implementation style of Fortran. An alternative would have been an abstract data model. This, however, would have been untypical for the Fortran codes. Also, it would have been more difficult to ensure efficiency of the memory usage and numerical calculations.

The data model was discussed within the consortium in a virtual meeting on May 8, 2012.

Chapter 2

Data models of data assimilation systems

2.1 PDAF

The Parallel Data Assimilation Framework – PDAF – is a computational framework for ensemble data assimilation. PDAF simplifies the implementation of the data assimilation system with existing numerical models. With this, users can obtain a data assimilation system with less work and can focus on applying data assimilation. PDAF aims at large-scale problems and is optimized for optimal compute performance and memory usage. PDAF is naturally parallel and is implemented with the Message Passing Interface (MPI) standard. Computation without MPI is possible using a stub-implementation of MPI. PDAF provides a selection of common filter algorithms that are optimized and parallelized.

The data model followed in PDAF is particularly simple to enable one the easy connectivity to numerical models. To extend a model with PDAF to a data assimilation system, only a few subroutine calls have to be added. In addition, the feature of parallel ensemble integrations is also provided for serial (i.e. non-parallelized) model.

The core part of PDAF contains the analysis step for the different filter algorithms implemented in PDAF. The core of PDAF is independent of the numerical model and also independent from the observations. The computations performed in the analysis step are mainly linear algebra operation. Accordingly, PDAF operates entirely on vectors and matrices, which are implemented as Fortran arrays.

2.1.1 Basic dimensions and arrays

The following dimensions are considered in PDAF. All are implemented as variables of type INTEGER:

1. size of state vector (`dim`)
2. size of observation vector (`dim_obs`)
3. size of ensemble (`dim_ens`)

The common arrays of the analysis step are:

1. state vector (`state(dim)`)
2. ensemble matrix (`ens(dim, dim_ens)`)
3. observation vector (`obs(dim_obs)`)

Next to these arrays, serveral additional arrays are temporarily allocated during the analysis step. These additional arrays are only used within the routines containing the implementation of the analysis step.

The parallelization of PDAF assumes domain-decomposition. That is, a state vector is distributed over several processes as sub-states. Analogously, the ensemble matrix is distributed, such that each process holds a full ensemble of sub-states. (An alternative mode-decomposed variant of PDAF exists, but does not contain all features of the released domain-decomposed variant.)

2.1.2 Connecting model and PDAF

PDAF implements the assimilation system by connecting PDAF and the model into a single executable. The data transfer between PDAF and the model is performed in memory using call-back routines. These are subroutines, that are implemented by the user and called by routines of PDAF. Required are 2 routines:

1. **get_state** – This routine initializes model fields from an ensemble state vector provided by PDAF. This routine is called before the forecast of the model state is conducted.
2. **put_state** – This routine initializes an ensemble state vector from PDAF by model fields at the end of a forecast.

The model integration (forecast) is controlled by an additional call-back routine:

1. **next_observation** – This routine determines the number of time steps for the integration of the ensemble of model states. In addition, the model time at the beginning of the forecast is set. Finally, a flag is initialized, which provides the information whether further integrations have to be performed, or the integration part of the data assimilation system should be exited.

Before and after the analysis step, a call-back routine is called in which PDAF provides the full ensemble of model states:

1. **prepoststep** – This routine allows the user to access the full ensemble. Commonly, it is used to compute the ensemble mean or variances. In addition, fields can be written into files.

2.1.3 Handling of observations

Obervations are also treated in user-supplied call-back routines. For global filters, the following routines are used:

1. **init_dim_obs** – This routine is called to initialize the number of available observations at the model time of the call

2. **init_obs** – This routine is called with PDAF providing the array for the vector of observations. The routine has to initialize the vector with the actual values of the observations.

3. **obs_ob** – This routine has to provide the implementation of the observation operator. The routine is provided with a state vector and has to return the observed part of it. As the observation operator is implemented in a functional way, any form of observation operator, including a non-linear one, is possible.

The handling of the observation error covariance matrix depends on the chosen filter algorithm:

1. SEIK/ETKF/SEEK/ESTKF
   prodRinvA – This routine is provided with some matrix **A** and has to return the product of the inverse of the observation error covariance matrix with the matrix **A**. This operation occurs in all of the ensemble square-root filters.

2. EnKF
   add_obs_error – For the EnKF, this routines has to add the observation error covariance matrix to some matrix. This operation only occurs in the EnKF.

The implementation of these routines is up to the user. As they are implemented in a problem-specific way, one can ensure optimal performance of the routines. For the filters involving localization, additional routines have to be implemented, that perform the localization of data for a local analysis domain.

2.2 OpenDA

OpenDA is an open interface standard for (and free implementation of) a set of tools to quickly implement data-assimilation and calibration for arbitrary numerical models. OpenDA wants to stimulate the use of data-assimilation and calibration by lowering the implementation costs and enhancing the exchange of software among researchers and endusers. As a generic toolbox for data assimilation, OpenDA provides a set of interfaces that define interactions between components. In addition, a library of data-assimilation algorithms is provided. The design goals of OpenDA are to provide shared tools to reduce implementation costs and to provide shared knowledge between different applications. Also it enables the development of algorithms, e.g. by universities. Since the algorithms are completely independent of the model, they are easier to test, which should result in fewer bugs. Applications with OpenDA are configurable without recompiling. OpenDA is portable to common platforms (Windows, MacOS, Linux) and aims at "good" performance.

OpenDA bases on object oriented concepts, e.g. there is no direct access to data like Fortran arrays. Instead functions are used for all kinds of operations. The main objects in OpenDA are:

1. treeVector

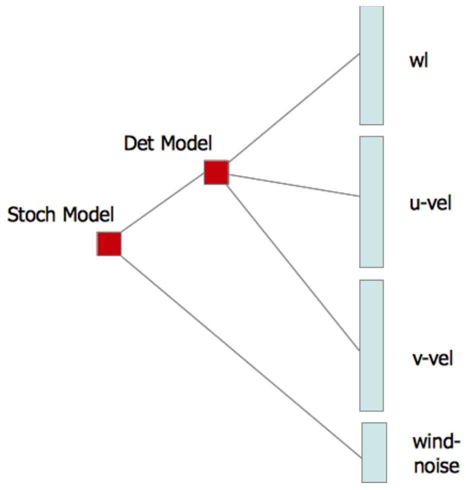

Figure 2.1: Graphical representation of a treeVector. The leafs of the treeVector contain the values. The treeVector of this example represents the state of a stochastic model. The state of this model is at the highest level a concatenation of a deterministic model with a (wind) noise model. The state of the deterministic model on its turn consists of three arrays containing the water level, and two velocity components. Note that the representation of the actual values can be completely different. In this example, the state of the deterministic model could stored in three Fortran arrays and the state of wind noise model in a java array.

2. stochasticModel
3. stochasticObservations
4. data assmilation/calibration method

The object oriented approach shields the exact data representation which is e.g. in-memory, files or scattered over various computers from the data assimilation. This is a fundamental design choice in OpenDA, which allows various kinds of models and observation sources to be use in combination with a single implementation of a data assimilation or calibration algorithm. The main objects of OpenDA will be briefly discussed in the following sections.

2.2.1 treeVector

The treeVector in OpenDA is a representation of a mathematical vector with some additional properties. A treeVector can be defined as a concatenation of other treeVectors called sub-treeVectors in this context. This allows us to group scattered values, possibly with a different representation in a single entity. These sub treeVectors have an unique name/tag that allows direct access /usage of sub treeVectors without any knowledge on length or composition of the other sub-treeVectors. A treeVector is graphically presented in Figure 2.1. The necessary mathematical operations on and between treeVectors like dot-producs, scaling, addition are performed by invocating methods (calling functions) no direct access to the values is necessary.

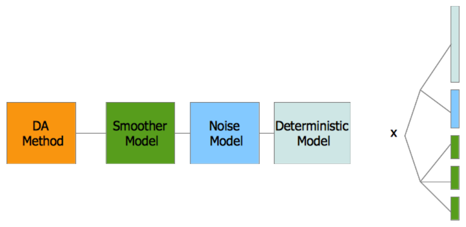

Figure 2.2: Graphical representation of a stochastic model including a smoother. The composite model is built from a number sub models. These models have no knowledge on each other's internals. The treeVector representing the state vector of this model is shown as well.

Additional meta information like units and grids can be added to treeVectors at all levels. This allows some basic automatic interpolation of values and the use of generic plotting/post processing utilities.

2.2.2 stochastic model

The formal form of a model in OpenDA is

$$\frac{dx(t)}{dt} = M\left(x(t), u(t), p, w(t)\right) \tag{2.1}$$

Where $x(t)$ denotes the model state at time t, $u(t)$ the time dependent forcings, p the time independent parameters and $w(t)$ the noise. x, u, p and w cannot be accessed directly. The values can be manipulated using a set of functions (the interface of the stochastic model).

Composite models (see Figure 2.2) can be build in a similar fashion as the treeVectors in OpenDA. The models that together form a composite model have no knowledge of each other's data representation. All interaction between the model is realized by using the methods from the model interface (calling functions).

2.2.3 stochastic observations

The stochasticObserver is the component in OpenDA that contains observations. In addition to the measured values, a stochasticObserver contains meta information on the observations such as location, quantity, interpolation kernel and the error model.

A stochasticObserver has some similarities with a relational database and information can be retrieved in the form of (tree)Vectors using enquire methods. It is possible to create a new stochasticObserver instance containing a sub-set of observations based on all kinds of properties like time, quantity location etc. The other way around, it is possible to combine multiple stochasticObservers into a single stochasticObserver (used by the data assimilation method). In this way we have a simple way to combine observations from different sources.

The optional addition of noise, according to the error model of the observation is performed by the stochasticObserver as well on request.

The interpolation of the model state to the observed values is considered to be very model dependent. Therefore it is not part of the stochasticObserver but part of the stochasticModel. For this purpose, it is possible to extract the necessary meta information on the observations. This meta information is then available in a component called ObservationDescriptions. This component is used as input to the state interpolation method of the stochasticModel.

2.2.4 data assimilation/calibration method

Data assimilation and model calibration methods are developed in a model independent way. In general, a data assimilation method is built using methods of the treeVector, stochasticModel and the stochasticObserver. Parallel computing is not explicitly programmed in the data assimilation method but is hidden in the implementations of the stochasticModel (and in the future treeVector).

2.3 Beluga/Sequoia

SEQUOIA (Sequential Optimization, Initialization and Analysis) is a modular assimilation system builder developed by LEGOS and disseminated via the SIROCCO French coastal ocean modeling national service. BELUGA is an analysis kernel of SEQUOIA that provides a 4D/local ensemble Kalman filter scheme. SEQUOIA has been used both in research and for operational forecasting. It has been used extensively within European projects (MERSEA, MFS, ECOOP, MyOcean, GOCEAN) and operational or industrial partnerships. In addition, as an implementation by today's standards (Fortran95, modularity, etc..), SEQUOIA adapts equally well to structured grids (finite difference) as to finite elements / finite volume via a system of generalized grid. Its standardized interface with the numerical model allows it to control virtually any model. Currently, SYMPHONIE, MOG2D/T-UGOm, and POLCOMS are interfaced, and other models are considered. The code also manages all of the simulation on cluster of PCs or standalone parallel machine.

The data model of SEQUOIA contains the following components:

2.3.1 Grid and state vector

Assimilation increment and analysis error variance

The global assimilation increment on the estimation grid, `g_dx`, is defined and allocated as follows:

```
integer, save ::  ak_mvs, ak_nodes
real, dimension(:,:), allocatable, save ::  g_dx
allocate ( g_dx(ak_mvs,ak_nodes) )
```

where `ak_mvs is` the number of variables on the vertical, and `ak_nodes` the number of nodes in the horizontal. Those notions are clarified below. The analysis error variance `g_ea` is defined in the same way as `g_dx`.

Multivariate Vertical Sequence (MVS) and variable types

If one takes the global grid, state variables on the vertical form the MVS. For example, an MVS size `ak_mvs` of 121 would result from the variables sea-surface height and 30 levels of each temperature, salinity, and of both components of horizontal velocities.

Not all variables may be present at one particular node, e.g. because of a shallower depth. An array is used store the number and indices of variables which are present at that particular node. This is

```
!  number of vertical variables at nodes
integer, dimension(:), allocatable, save ::  g_nmvs
allocate ( g_nmvs(ak_nodes) )
!  indices of variables at nodes
integer, dimension(:,:), allocatable, save ::  g_rmvs
allocate ( g_rmvs(ak_mvs,ak_nodes) )
```

State-space objects such as `g_dx` are defined in the “packed MVS” format defined by `g_nmvs` and `g_rmvs`, i.e. `g_dx(i,:)` refers to variable `g_rmvs(i,:)` of the global MVS.

The state variable types are stored in a separate array in packed MVS format:

```
!  state variable types
integer, dimension(:,:), allocatable, save ::  g_tmvs
allocate ( g_tmvs(ak_mvs,ak_nodes) )
```

The state variable types in `g_tmvs` and data (observation) types in `d%type` (defined further below) are of course expected to be consistent with each other, although of course the number of state variable types and data types can be different for one particular problem.

Usually the same numeric value to refers to one particular variable type. The state variable depths are defined in packed MVS format as:

```
real, dimension(:,:), allocatable, save ::  g_znod
allocate ( g_znod(ak_mvs,ak_nodes) )
```

In `g_znod`, depth is to be replaced by any appropriate coordinate for models not in z-coordinates.

Horizontal grid

The following variables are defined at each horizontal node:

```
!  unique node ID
integer, dimension(:), allocatable, save ::  g_node_id
allocate ( g_node_id(ak_nodes) )
!  geographic coordinates
real, dimension(:), allocatable, save ::  g_xnod, g_ynod
allocate ( g_xnod(ak_nodes) )
allocate ( g_ynod(ak_nodes) )
```

Usually, the grid is unstructured and made up of triangular elements (“cells”), whose vertices are the nodes mentioned above. When the input grid is structured (finite-difference models), it is broken up by the code into triangular cells while maximizing isotropy.

```
      integer, save ::  ak_cells
      !  pointers to node numbers for each cell
      integer, dimension(:,:), allocatable, save ::  g_icel_nodes
      allocate ( g_icel_nodes(ak_cells,3) )
```

Note: In the code, there is a difference between `ak_nodes`, the "declared" number of nodes, and `g_nnod`, the number of active nodes. This distinction is omitted here. The same distinction holds for `ak_cells` and `g_ncel`.

2.3.2 Observations

Internal representation

Individual observations, or observation-space objects such as "bogus" observations, are defined as a derived data type as follows:

```
   type ak_obs_struct
      sequence
      integer ::  type !  observation type
                       !  - same coding as for state-space variables
      real ::  x, y, z, t !  space-time coordinates of observation
      real ::  yo !  observed value
      real ::  eo_std !  observation error standard deviation
      integer ::  qcflag !  quality flag
      integer ::  status !  data status marker
      integer ::  set !  dataset number
      real ::  yopert !  perturbation on observed value
      integer, dimension(3) ::  vertex !  pointers to nodes
                                       !  surrounding observation
      real, dimension(3) ::  lambda !  corresponding barycentric coordinates
      real ::  yf !  prior (forecast) value, model proxy for the observation
      real ::  yp !  persisted value from last analysis
      real ::  dy !  innovation
   end type ak_obs_struct
```

Some of these fields are provided by the user `(type,x,y,z,t,yo,eo_std,qcflag,status)`, some by the data loader `(set)`, the others are calculated by the code.

The data status markers are one way to individually select observations to be "verification-only" (not used in the inversion) or to set bogus observations. Markers can also be defined globally per dataset. Currently, declarations are static in the code:

```
      integer, parameter ::  AK_OBS = 100000 !  max number of observations
      type(ak_obs_struct), dimension(AK_OBS), save ::  d !  the data vector
      integer, parameter ::  AK_TYPES = 10 !  max number of data types
      integer, dimension(AK_TYPES) ::  data_type
```

Observation operator

In SEQUOIA, there are two ways to handle observations: Proxies of observations `d%yf` are directly obtained from the model interface at the surrounding nodes `d%vertex`, then interpolated at the horizontal location of the observations by means of `d%lambda`. This is done at the time step when the observation occurs. There is no need for an observation operator for the proxies.

Some of the SEQUOIA kernels require explicitly an observation operator in the form of a tangent linear observation matrix (the H matrix) when forming, e.g., the Kalman gain. This is done by means of a library call (`ak_obsop()` user-provided routine). There is obviously one observation operator per data type.

In the BELUGA kernel no observation operator is needed for the covariances, which are calculated from the state-space samples and proxies provided by the members.

2.3.3 Samples library

In the BELUGA EnKF-kernel not the covariances themselves are shared between different ensemble members, but the samples library, which gathers the state-space and data-space samples needed for the covariance calculations. The samples library is defined as follows:

```
integer, save ::  mlist_size, p_glo
!  state-space samples
real, dimension(:,:,:), allocatable, save ::  xf_E
allocate ( xf_E(ak_mvs,ak_nodes,mlist_size) )
!  data-space samples
real, dimension(:,:), allocatable, save ::  yf_E
allocate ( yf_E(p_glo,mlist_size) )
```

where `mlist_size` is the size of the members list (the number of members), and `p_glo` is the currently selected number of observations over the integration time segment.

The following objects are also shared between members:

```
!  state-space ensemble mean
double precision, dimension(:,:), allocatable, save ::  xf_mean
allocate ( xf_mean(ak_mvs,ak_nodes) )
!  state-space ensemble variance
double precision, dimension(:,:), allocatable, save ::  xf_var
allocate ( xf_var(ak_mvs,ak_nodes) )
!  data-space ensemble mean
double precision, dimension(:), allocatable, save ::  yf_mean
allocate ( yf_mean(p_glo) )
!  data-space ensemble variance
double precision, dimension(:), allocatable, save ::  yf_var
allocate ( yf_var(p_glo) )
```

The analysis of the BELUGA-kernel is carried out locally at each grid node. Observations are selected within an “influence bubble” in space and time. Covariances are calculated for each node by each member from the member copy of the samples library, using localization. There is no global calculation of covariances.

2.4 SESAM

The SESAM code, developed in the MEOM group at LEGI, performs the various basic operations that are required in sequential data-assimilation systems. These operations include square root and ensemble observational updates (with global or local parametrization of the forecast error statistics), adaptive statistical parametrizations, anamorphosis transformations, or the computation of truncated Gaussian estimators. SESAM also provides diagnostic tools, to compute observation representers, EOF decompositions or regional RMS misfits, and various utilities for extracting observations, converting between file formats or performing simple algebraic operations.

SESAM uses a black-box coupling to a model through NetCDF files. Thus, separate programs are used to peform the model integrations and the tools provided by SESAM.

Basic variables

SESAM uses a simplistic data model. Nearly all variables are native arrays and no ancillary data is attached. In particular, the following quantities exist:

State vector:

```
BIGREAL, dimension(:), allocatable, save ::  vectx
```

Note: BIGREAL is a precompilation keyword allowing the user to choose the type of the data.

Observation vector:

```
BIGREAL, dimension(:), allocatable, save ::  vecto
```

Ensemble of state vectors:

```
BIGREAL, dimension(:,:), allocatable, save ::  basexr
```

Note: The last two characters in a variable name give the names of the dimensions to which the array is allocated. Thus: x=state variables, r=ensemble members or columns of the squared root covariance matrix.

Observed part of ensemble:

```
BIGREAL, dimension(:,:), allocatable, save ::  baseor
```

Observation error covariance matrix (assumed to be diagonal):

```
BIGREAL, dimension(:), allocatable, save ::  diago
```

Observation Operator

The observation operator uses a derived data type of the form:

```
TYPE(type_poscoef), dimension(:,:), allocatable ::  poscoefoj
```

This derived type is defined as

```
TYPE type_poscoef
   INTEGER ::  pos
   BIGREAL ::  coef
END TYPE type_poscoef
```

Assumed is a linear operator. For each observation it is:

`pos`: The indices of the states variables in the state vector to which it is related.

`coef`: Coefficient of the linear relation with the observations.

This scheme is specifically designed for linear interpolation, so that the second dimension of the array is typically 4 in 2D and 8 in 3D.

Localization

In case of localization, there is for each subsystem of dimension z:
Reduced dimension state vectors:

```
    BIGREAL, dimension(:,:), allocatable ::  vectrz
```

Reduced space covariance matrices:

```
    BIGREAL, dimension(:,:,:), allocatable ::  covrrz
```

Ancillary data (such as grid, mask,...) are only used for preprocessing to prepare these simple types from model files. In particular, the grid is required to prepare the observation operator. However, this is done in a preprocessing step and done in such a way that this is well separated from the generic methods themselves.

Ancillary data are also used in a preprocessing step to define the localization arrays. These are are defined by
(i) one state vector type array to identify the index of the subsystem to which each state variable belong, and
(ii) for each of the subsystem of dimension z, the list of state variables inside the corresponding influence bubble.

2.5 NERSC EnKF

NERSC uses a Fortran90 implementation of the EnKF for operational use in the TOPAZ system, which is used in the MyOcean project (there is also a Matlab research toolbox, which is not described here). Both the interface and parallelization are optimized for the HYCOM output file structure. However, they can be adapted by a skilled scientist to a different model. Classical diagnostics such as the innovations, ensemble spread, Degrees of Freedom of Signal (DFS) and Spread Reduction Factor (SRF) are produced in NetCDF format for easy monitoring of real-time forecasting and long reanalysis runs.

For the operational use of the EnKF, the NERSC EnKF system is optimized to handle large amounts of data. For each assimilation cycle, about 2.5 million observations of different quantities (satellite data of the sea level anomaly, surface temperature, ice concentration and drift, as well as in situ data of temperature and salinity) are assimilated. The EnKF analysis is linked to the HYCOM model through files.

The analysis part of the code is executed for the TOPAZ system in three steps using a different number of processes:

1. Prepare observation (computation of pivot point, superobing to reduce total number of observations, profile passed in hybrid coordinates).
2. Assimilate observation
3. Post-processing (assemble output and cut-off unrealistic values)

There are a several particular features of the analysis:

- Available are two analysis schemes: EnKF and the Deterministic EnKF (DEnKF)
- The observation error covariance matrix is assumed diagonal
- A moderation is used that inflates the observation error if the ensemble spread and observation error do not intercept.
- A local analysis step is performed separately for each horizontal field in each layer. The innovations are tapered dependent on distance.
- The computation of the transform matrix (X5-matrix) and the model update are fully parallel using MPI.
- A parameter estimation possibility is used for selected parameters.

Observations and observation operator

Observations are described by a derived type. It contains all information on the value and type of an observation as well as the location of an observation. The derived type is defined as:

```
type measurement
  real d !  Measurement value
  real var !  Error variance of measurement
  character(len=OBSTYPESTRLEN) id !  Type, can be one of those:
  !  'SST', 'SLA', 'ICEC', 'SAL', 'TEM', 'GSAL', 'GTEM', 'TSLA'
  real lon !  Longitude position
  real lat !  Latitude position
  real depth !  depths of position
  integer ipiv !  i-pivot point in grid
  integer jpiv !  j-pivot point in grid
  integer ns !  representativity in mod cells (meas.  support)
  !  ns=0 means:  point measurements
  real a1 !  bilinear coefficient (for ni=0)
  real a2 !  bilinear coefficient
  real a3 !  bilinear coefficient
  real a4 !  bilinear coefficient
  logical status !  active or not
  integer i_orig_grid !  orig.  grid index for ice drift processing
  integer j_orig_grid !  orig.  grid index
  real h !  layer thickness,
  integer date !  age of the data
  integer orig_id !  used in superobing
end type measurement
```

Analysis step

The analysis step of the NERSC EnKF uses basic data types. Dimensions are specified as integers:

```
integer ::  nobs !  number of observations
integer ::  nrens !  number of ensemble members
integer ::  idm, jmd !  Horizontal grid indices
```

Arrays are used to store the innovations (observation minus forecast ensemble state) and the ensemble observation anomalies:

```
real ::  d(nobs) !  Innovation
real ::  S(nobs, nrens) !  Ensemble observation anomalies
```

In order to reduce the required memory during the computations of the analysis step, local transformation matrices are pre-computed and stored in files. For the analysis step, they are then read and processed sequentially. Also the different fields to be analyzed are treated separately and distributed over several processors.

2.6 OAK

The Ocean Assimilation Kit (OAK) provides a modular toolbox for data assimilation mainly aimed at oceanographic applications. The definition of state variables is very flexible in OAK. By means of easy configuration files the user can select arbitrary variables from NetCDF files. Curvilinear grids are supported as well. In addition, OAK provides global and local versions of the analysis

The data model bases on Fortran arrays with the addition for derived types to describe the data structure in the model and the memory layout.

2.6.1 Memory layout description

The derived type `MemLayout` contains the following information

- Names of the individual variables
- Land-sea mask (as array pointer of type integer)
- start and end index in the state vector for each variable
- Permutation index (optional, useful for local assimilation)
- Distribution over nodes for parallel computing

The transition between model fields and state vector is performed by packing/unpacking functions. The assembly of the observation vector is performed by an analogous packing function as used for the state vector. For the state vector the packing function is as follows:

```
packVector(ML,x,temp,salt, uvel, vvel, ...)
```

where the arguments for the separate fields, e.g. `uvel` are optional in the interface. The memory Layout is given by the argument `ML`. The vector can also be loaded from a file by the function:

```
loadVector(path,filenames,ML,x).
```

2.6.2 Model grid description

The model grid is described in the derived type `grid`, which contains the following information

- The dimension of the grid
- Land-sea mask
- Coordinated for every grid point

The derived type only supports structured grids.

2.6.3 Observation operator

The observation operator is defined as a sparse matrix with a type including the following information:

- Number of non-zero entries
- Number of rows and number of columns in full matrix
- i, j indices of non-zero entries
- value of matrix element (usually an interpolation coefficient)

. This scheme only supports linear observation operators, but nonlinear observation operators can be used by augmenting the state vector by the observed variable. The sparse matrix contains only non-zero elements and implements an operator for multiplication with a matrix or a vector. Bi-linear interpolation coefficients in the observation operator are based on the coordinates of the observations and the model grid definition. For parallel computing, the observation operator can be distributed across nodes as a function call.

The derived data types are only used in high-level routines, which perform the loading and saving of the state vector, ensemble, and observations. The routines also compute assimilation diagnostics per variable

2.6.4 Variables in low-level routines

The low-level computational routines use only Fortran arrays. Used are the basic dimensions:

```
n !  Size of state vector
m !  Size of observation vector
r !  Ensemble size
```

The Fortran arrays used are

```
x(n)     !  State vector
Hx(m)    !  Observed part of state vector
yo(m)    !  Observation vector
S(n,r)   !  Ensemble perturbations in state space
HS(m,r)  !  ensemble perturbations in observation space
```

2.6.5 Local analysis

To perform local analysis updates, a partition vector `part(n)` of type integer is use. All elements with the same entry in the partition vector belong to the same local analysis zone. Frequently, the local partitioning uses vertical water columns. A permutation vector is applied to ensure that all variables belonging to the same zone are contiguous in memory.

Chapter 3

Aspects of the MyOcean data models

The MyOcean initiative uses different ocean models for operational forecasting. Among these models are the NEMO model and the TOPAZ system, which uses the HYCOM ocean model. Both models are also used by some of the members of the SANGOMA consortium.

The NEMO model, which will also be used for benchmarking in WP4 of SANGOMA uses CF-compliant NetCDF files. Internally, basic data types are used.

Derived data types are used in NEMO for diagnostic purposes and to describe forcing input files.

For the connection with SANGOMA assimilation tools, the CF-compliance of the output files is of relevance. Most connections between the NEMO model and assimilation systems used in SANGOMA will be performed using the information from these files.

Chapter 4

SANGOMA data model

4.1 General aspects

For the SANGOMA tools, we consider two types of interfaces:

In-memory: The in-memory interface has to be specified, if an assimilation tool is a callable function or subroutine that is called within the model code or the data assimilation system.

Files: This type of interface occurs, when the assimilation tools are in a separate executable program than the numerical model. Thus, a file standard has to be described to allow for compatibility of files with the reading and writing routines in the tools.

For the file interface, SANGOMA will base on the NetCDF file format, which is very popular for oceanographic and meteorological data. NetCDF is a binary file format that is self-describing by allowing to store meta data as attributes in the file header followed by the data in the body of the same file. SANGOMA will follow the CF-convention for the naming of variables and meta data inside the files. By following the CF-convention, the file-based interface of SANGOMA will also be compatible with output files of models used in MyOcean. The file interface of SANGOMA is described in section 4.2.1.

Regarding the in-memory interface, the assessment of the data models in the different data assimilation systems used by the members of the SANGOMA consortium showed a wide range of differences. The object oriented data model of OpenDA stands out from the other assimilation systems, which are implemented in Fortran. These systems use basic data types, i.e. Fortran arrays and variables at least in their computational routines. Some systems, like OAK and Sequoia/-Beluga use derived types for the handling of observation, e.g. the definition of the observation operator. These derived data types are distinct for the different assimilation systems. Within MyOcean, the models also use commonly basic data types. Sections 4.2.2 to 4.4 will described the logical data model for the in-memory interface as well as the aspects of a compatible binding to C code and some considerations for the interface description.

4.2 Logical data model

4.2.1 File interface

Introduction to NetCDF

NetCDF is a popular data format for oceanographic data. It uses an efficient binary data representation, that is portable across different operating systems. Access to the data is provided by a library with bindings to many programming languages including Fortran, C, java and matlab. The official NetCDF website at http://www.unidata.ucar.edu/software/netcdf/ provides a lot of useful information. In addition to the libraries, many useful tools exist, eg. to show or plot the contents of a NetCDF file.

Between version 3 and version 4 of NetCDF the default binary format has been changed. Version 4 now uses HDF as the binary format. This means that NetCDF files written with the version 4 library can also be read with the HDF library, but no longer with the version 3 library. Fortunately, it is still possible to read files written by the version 3 library with the new library. Because only part of the available tools have been updated to to version 4, it is not a trivial question which version should be used, but we will try to give some recommendations below.

The basic storage elements in NetCDF are a number of integer and floating point representations and multidimensional arrays. Each element has a name and can have attributes. Attributes help describe the contents to make the NetCDF file self-describing. Some useful attributes are a pointer to the documentation, units, coordinate descriptions.

Some useful features of NetCDF over binary files written directly from a Fortran program are:

- Existence of many tools to create, modify and display the contents
- Language bindings for many languages
- Possibility to create self describing files, with the use of attributes
- Possibility to read and write part of an array in slices, eg. 2D (x,y) slices into a 3D (x,y,t) array.
- Transparent transfer of data over internet with the OpenDAP protocol. Remote files can be accessed as if they were local and only the data that is actually requested is transfered, which is very useful for spatial and temporal selection of data.
- Possibility to transparently compress the data (version 4 only)

The dimensions of an array in a NetCDF file have names. These names can be used to link the dimensions between arrays. In the short example below, the `salinity` array has dimensions `(time, depth, lat, lon)`. Note that in Fortran the dimensions would be reversed, so this array corresponds to a Fortran declaration as `REAL(KIND=C_FLOAT) :: salinity(nlon,nlat,ndepth,ntime)`. The array `lat(lat)` has the same name for dimension as the array name, but only the dimension name links it to the `salinity` array.

```
dimensions:
   time = 2 ;
   depth = 24 ;
   lat = 166 ;
   lon = 91 ;
variables:
   float salinity(time, depth, lat, lon) ;
   float lon(lon) ;
   double time(time) ;
   float lat(lat) ;
   float depth(depth) ;
```

Meta data standardisation

To further standardise the use of attributes the CF-convention was invented. The CF-standard is still being extended. The latest version, version 1.6 at the time of writing this document, can be found at http://cf-pcmdi.llnl.gov/. Some important aspects of the CF-standard are:

- **standard_name** The list of standard names describes common variables for meteorology and oceanography (see http://cf-pcmdi.llnl.gov/documents/cf-standard-names) For example the standard_name for salinity is sea_water_salinity. Standard names are important to select matching variables.
- **units** Standard units are used to allow for conversion, eg 'hPa' or 'm/s'
- **spatial axis** NetCDF supports several options, but the of longitude and lattitide arrays is the simplest.
- **time axis** Definition of the offset and units for time eg 'hours since 1990-12-25 0:0:0'
- **ordering of dimensions** In CF the order of the spatial and temporal dimensions is fixed to `(time, depth, lat, lon)`. This means that time is the slowest running index, so that time slices will be stored as continuous blocks of data in the file. Note that for Fortran arrays the order is reversed, because the fastest running index comes first in Fortran.

The example below shows a part of the header of a NetCDF that contains a 4D array of salinity. The full header can be found in the appendix and more examples are easily generated with `ncdump -h <filename>` from examples available on internet, such as the excellent MyOcean site http://www.myocean.eu.org/.

```
dimensions:
   time = 2 ;
   depth = 24 ;
   lat = 166 ;
   lon = 91 ;
variables:
   short vosaline(time, depth, lat, lon) ;
```

```
            vosaline:_CoordinateAxes = "time depth lat lon " ;
            vosaline:_FillValue = -32768s ;
            vosaline:missing_value = -32768s ;
            vosaline:scale_factor = 0.001f ;
            vosaline:add_offset = 30.f ;
            vosaline:standard_name = "sea_water_salinity" ;
            vosaline:long_name = "Sea Water Salinity" ;
            vosaline:units = "1e-3" ;
        float lon(lon) ;
            lon:standard_name = "longitude" ;
            lon:units = "degrees_east" ;
            lon:long_name = "longitude" ;
        double time(time) ;
            time:standard_name = "time" ;
            time:units = "seconds since 2011-04-07 00:00:00" ;
        float lat(lat) ;
            lat:standard_name = "latitude" ;
            lat:units = "degrees_north" ;
            lat:long_name = "latitude" ;
        float depth(depth) ;
            depth:standard_name = "depth" ;
            depth:units = "m" ;
            depth:positive = "down" ;
            depth:long_name = "depth" ;
// global attributes:
            :title = "North West European Shelf from UK Met Office Model FOAM 7 km" ;
            :institution = "UK Met Office" ;
            :references = "http://www.ncof.co.uk" ;
            :source = "UK Met Office Operational Suite, FOAM 7 km run 2012-05-29" ;
            :Conventions = "CF-1.0" ;
```

Additional SANGOMA policies

Within the SANGOMA project, NetCDF files mainly serve to exchange data between models and the data-assimilation tools. Many ocean models can already produce NetCDF files, but standards and versions are different between models. In SANGOMA we strive for compatibility and simplicity. Since it is difficult to foresee all posibilities we provide guidelines rather than a specification/rules.

- Keep it simple. Since the NetCDF files are used to connect multiple models to multiple data-assimilation tools, more complex features are likely to break compatibility to some or may create a substantial amount of work for other partners of SANGOMA.

- SANGOMA tools should work for CF-compliant input files. This is probably hard to achieve to the full extent, but ocean models are not likely to use the more complex CF-features. We should probably start simple and adjust when needed.

- Be lenient to standards of model output. Educating developers of ocean models is not a task of SANGOMA. Different communities may be involved making it hard to change the CF-compliancy of model output. For many data-assimilation tools much of the meta-data can safely be ignored, making it unkind to throw errors for attributes that are not necessary. The output of data-assimilation tools is often of the same kind as the input files, e.g. for forecasted and analyzed state. The use of a template file derived from the input file will work even if the model output does not completely adhere to CF-standards. It also avoids writing code to copy all possible kinds of attributes to the output files, which a subsequent model run may need to read the analyzed state.

- Selection of arrays on which the tool should operate should preferably be configurable and not inferred from the CF-attributes. There are several reasons for this. The input file may contain more data than one wishes to use the tool for. Future users of the tool may have datatypes not foreseen at the time of writing the tool, which is more easily adapted through configuration than by inference. For example a global ensemble analysis update does not use coordinate values, so a tool written for and tested with a model with a structured grid can easily be adapted for an unstructured grid if the tool ignores the meta-data in the attributes.

- Tools can be compiled with NetCDF version 3 or 4, but features that break compatibility with version 3 should be avoided, unless they are necessary for the tool to function. Using specific version 4 features is likely to make it impossible to use the tool/code in some cases. If some NetCDF version 4 dramatically improves the functionality or performance one can also consider to provide this feature as an option.

- Some tools will work on ensembles. Though it is possible to store ensembles in a single NetCDF file, we recommend to use separate files for each member of the ensemble. This probably makes the interaction with the ocean models simpler, since ocean models are unlikely to recognize the additional dimension. Also, existing tools for displaying the data are less likely to work.

- It may be necessary or useful to read or write datatypes that are not common to the NetCDF community, eg and ensemble transform matrix or a probability density function. It is kind, but not required, to try to use a logical extension of the CF-standard for writing. For reading, it is probably best to be very lenient on standards in these cases.

4.2.2 In-memory interface

For the code-based or in-memory interface, the compatibility of the SANGOMA tools written in Fortran an other programming languages like C, Matlab, or Java has to be kept in mind. To ensure the interoperability with interfaces that are directly callable from Fortran and the other languages, the data model of SANGOMA relies on elementary data structures for both input and output variables, and avoids complex data types. In particular, the SANGOMA data model relies on vectors and matrices, which are elementary data entities in the linear algebra calculations of the analysis steps calculated by the different assimilation systems. In Fortran code, these data entities will be represented by arrays. By using basic data structures, the data model is in fact similar to that used by the PDAF assimilation system that was described in section 2.1.

The data model excludes the use of Fortran derived types from the interfaces. This is motivated by the fact that Fortran derived types and C structs are only compatible if they are defined identically in both languages. In particular, the same data elements need to be included in the types and structs in both languages. As the survey of the data models of the different assimilation systems showed, the derived types in the different systems are distinct and often include a long list of information. If one would introduce derived types in the interface of the SANGOMA tools, one would need to define one standard derived type, which then needs to be defined identically in Fortran and C. The developers of the different systems would either need to adapt their derived types to this SANGOMA standard type, or to fill the SANGOMA derived type from their system-specific data type. To avoid the overhead to filling or adapting derived data types, call-back functions will be used within SANGOMA tools in cases where basic data types are insufficient. These call-back functions will be specified as an argument in the call to a tools (basically to represent a function pointer, if viewed from C) and will be called from within the tool routine. In the call-back function, a system-specific derived data type can be included, e.g. by using a Fortran module.

The decision to rely on basic data types, allows in particular the programmers that are used to Fortran to continue using typical implementation styles of the Fortran programming language. This is considered to be important, because also most large-scale ocean and atmospheric models, including those ocean model used within MyOcean are coded in Fortran. An alternative would have been to base on data abstraction as it is used in the object-oriented approach of the OpenDA assimilation system. However, discussions among the SANGOMA partners showed, that such strategy felt non-natural for the Fortran programmers. As it cannot be expected that the abstraction would lead to a better computational performance, it is more natural to rely on basic data types.

Analogous to PDAF, the SANGOMA data model assumes the existence of basic dimensions, which are just scalar integer numbers. In fact, on the computational level, a data assimilation system requires only a very limited number of dimensions (the names of the variables are suggestions):

- size of state vector (`nstate`)
- number of (scalar) observations (`nobs`)
- size of ensemble (`nens`)

There might be additional dimensions, like the number of observation of a particular field, if observations of different physical quantities are assimilated. However, for the handling of observations, which can become complex if one seeks for an abstract description, call back functions are recommended. These will also allow for more complex data structures, which can also be specific for a data assimilation system. Further, the parallelization might result in additional dimensions. These are usually sub-dimensions. That is, if a state vector of size `nstate` is distributed over several processes, each process holds a sub-vector of size `nstate_l(i)`.

The dimensions listed above specify the size of vectors and matrices that represent different quantities in the tools. For example:

- A state vector is logically a vector of size `nstate`. It will be represented by a one-dimensional array.
- An ensemble matrix is a matrix holding in each of its columns one state vector. As the ensemble size is `nens`, the size of the ensemble matrix is `nstate` $\times$ `nens`. (Here, the first dimension describes the number of rows in the matrix, while the second dimension described the number of columns.)
- The observation vector is logically a vector than contains the values of all available (scalar) observations. Hence the size of this vector is `nobs`.
- There might be further quantities in the assimilation systems, that are logically also vectors or matrices. One example is the ensemble projected onto the observation space, i.e. the ensemble of the model counterpart of the observational data. This ensemble is a matrix of size `nobs` $\times$ `nens`.

The review of the data models of the assimilation systems showed, that they use also more complex data structures, for example to describe the observation operator or the memory layout of the state vector. It can be useful to group the meta data describing, e.g., the observation operator. For this, the Fortran-based assimilation systems use derived data types. For the tools developed in SANGOMA, we will rely on call-back functions to make use of more complex data structures. These functions allow for a flexible specification of more complex data structures, while ensuring the interoperability of different languages.

An example for a call back function is the handling of the observation error covariance matrix. Many assimilation algorithms need to multiply the observation error covariance matrix or its inverse with some other matrix, which is used temporarily during the calculations of the analysis step. In order to allow for an abstract description of the observation error covariance matrix one can introduce a call-back function, which includes in its interface the matrix with which the product has to be computed. The call-back function has then to return the final product. A similar case might appear if one uses a covariance matrix to describe random noise to be added during the ensemble integrations in order to simulate model error. The description of the observation error covariance matrix is handled within the call-back function, such that the calling routine does not need to know how the observation error covariance matrix is described. There can be different cases of a covariance matrix, which can be described in different ways. In increasing complexity, some possibilities are:

- A diagonal matrix with constant variance. In this case, one only needs the information the covariance matrix is diagonal and the single value of the variance.

- A diagonal matrix with varying variances. If the variance is spatially dependent, then one needs the information that the matrix is diagonal, but in addition, one needs a vector of variances to describe the covariance matrix.
- A non-diagonal matrix with varying variances. This is the most general case. In some situations one might want to describe the matrix by the values of the diagonal, i.e. a vector of variances, plus a decorrelation length scale. The decorrelation length scale lets one compute the off-diagonal matrix entries. However, one might also want to directly supply the full observation error covariance matrix. Often the observation correlations are of short range, such that a significant part of off-diagonal entries are zero. In this case one might prefer to store the observation error covariance matrix in a sparse matrix format.

Another case where call-back functions can be used is for the handling of observation operators. Usually, one needs to compute the action of the observation operator onto a state vector. Thus, one can use a call-back function that is supplied with a state vector and which returns the vector resulting from the application of the observation operator onto the state vector. There are different observation operators dependent on the type of observation. For example:

- The simplest case are observations of a model field at grid point locations. In this case the observation operator is a matrix whose entries are either 1 (observation present) or 0 (no observation). One could store this observation operator in a sparse matrix format as, e.g., the OAK assimilation system does (see, section 2.6).
- If observations of a model field are present that are not located at grid point locations one needs to apply interpolation. In many cases one can precompute the interpolation coefficients, as is used, e.g., in the SESAM assimilation system (see section 2.4) or the OAK system.
- If observations are nonlinearly related to a model field, like the logarithm of a concentration, e.g. of an ecosystem model, there are different possibilities. One might either include these nonlinearly related quantities in the state vector and use the observation operator for grid point locations (used, e.g. in OAK, see section 2.6). If one wants to avoid the additional storage cause by extending the state vector, one might want to allow for the application of the nonlinear operator in the call-back routine (used e.g. in PDAF, see section 2.1). A nonlinear relationship, like the computation of the logarithm, could be directly implemented in the call-back function.

Overall, the use of call-back functions allows a very flexible handling of complex data structures. For example, they allow to implement the handling of the product of a matrix with a covariance matrix, such that it is compatible and efficient with a particular assimilation system. Also, the handling of the observation operator can be implemented in the most efficient way, given the frame of a chosen data assimilation system. This implies that, different assimilation systems will use their own call-back function. However, it will also give us the potential to unify more complex data structures, at least among the Fortran-based assimilation systems. As the call-back functions will be system-specific, they should be handled with care. Thus, the basic data model should be used as far as possible. Call-back functions should only be used in cases where the inclusion of the additional information in form of basic data types in a function interface would overly increase the length of the interface or where it would restrict the flexibility of the functionality too much.

The interfaces to call-back functions need to use the basic data structures to ensure the interoperability of different programming languages. The more complex data structures should be included in the call-back functions, e.g. by Fortran modules. As such the interfaces of call-back functions will be similar to the interfaces of the tool routines. In both cases only the required minimum of information should be included in the interfaces, like dimensions and the required arrays. For example, a function computing the product of a matrix with a covariance matrix, would need the dimensions of the input matrix as well as the matrix itself. The return matrix has also to be included, but no additional dimensions, because the return matrix will have the same size as the input matrix.

4.3 Interfaces with C-binding

In this section, we discuss some more technical aspects of the proposed in-memory interface of shared tools in SANGOMA. The tools we share for in-memory usage (functions and subroutines) will be denoted by "shared routines" in this section. Everything that is stated on the interface does only count for the shared routines. Not for routines that are called by the shared routines (except for call-back functions).

As mentioned in Section 4.2.2, we aim for usage and sharing of tools from various languages. Therefore

- we only allow basic data types, supported in all relevant languages, in the interfaces of the shared routines.
- we ensure a proper binding with C, because C routines can be called directly from many programming languages.

Many of the partners in SANGOMA use Fortran as their primary programming language. The Fortran 2003 standard contains C-binding functionality that allow Fortran code to be directly called from C and vice versa in a platform independent way. With these Fortran 2003 features, it is possible to realize the C binding of existing Fortran code with just a few additional keywords in the subroutine headers. We will give some basic tutorial information on the C binding functionality of Fortran 2003 in section 4.3.9, but much will already show up in the examples in earlier sections.

4.3.1 Argument types of shared routines

To ensure that shared routines implemented in various programming languages can be shared we only allow basic data types for the routine arguments.

The C-declaration of shared routines only have arguments of the following types:

- scalars by value: integers, real values and logicals (e.g. int, long, float, double, bool).
- pointers to scalars
- pointers to arrays of scalars
- strings (special story, see section 4.3.5)
- pointers to functions conforming to the SANGOMA interface standard (callback functions)

Note: various precisions are allowed for integers and reals.

The Fortran declaration of the shared routine arguments allows the use of:

- scalars of the type integer, real (with appropriate specification of `kind`, see section 4.3.4) and logical,
- n-dimensional arrays of the allowed scalars (fixed size),
- strings and
- functions conforming to the SANGOMA interface standard (callback functions)

Notice, that Fortran supports N-dimensional arrays and the C interface only supports arrays. N-dimensional (fixed size) arrays are stored in memory as a contiguous block of memory (column oriented) and are therefore compatible with the arrays in C.

Optional arguments cannot be ported between languages and are therefore not allowed.

The use of global variables is not allowed in shared routines. The only place where global variables can be used are in the callback routines.

4.3.2 Fixed size, assumed size vs assumed shape in Fortran

In Fortran there are different ways in which arrays are represented

- `real x(m,n)`: "fixed size" array (legal in Fortran 1.0)
- `real x(m,*)`: "assumed size" (legal in Fortran77)
- `real x(:,:)`: "assumed shape" (new in Fortran90)

Both fixed size and assumed size are contiguous blocks of memory, passed to routines by a memory pointer. The assumed shape arrays do not need to be contiguous blocks of memory and cary some meta information. Note that under normal conditions the memory is contiguous, which means that fixed/assumed size arguments of a subroutine can be called without copying of data (implemented by the compiler) with assumed shape arrays.

The interfaces of the shared routines do not allow assumed shape arguments but will only use fixed size arrays. However a passed fixed size argument can be transformed into an assumed shape variable without the need to copy the content using the c_f_pointer function of Fortran2003.

4.3.3 Arguments by value and reference

All variables are handled in Fortran by reference (a pointer to the address is passed). In C this is not the case. Scalar arguments, that do not change (equivalent to `intent(in)`) are passed by value (a copy of the value of the argument is passed). The Fortran 2003 C-binding features allow one to pass arguments by value as well in Fortran. The programmer only needs to add a single attribute "`value`" to the declaration of the argument and this does not have any implication when the routine is used from Fortran. We encourage the declaration of scalar arguments by value when possible to simplify calling these routines from C (and other languages).

4.3.4 Precision of values

In order to be able to call shared routines from different languages we need to ensure the proper precision of the data. The intrinsic `ISO_C_BINDING` module supports special Fortran kinds corresponds to the precisions used in C. See Table 4.1

Named constant from ISO_C_BINDING	C type	Equivalent Fortran type
C_SHORT	short int	INTEGER(KIND=2)
C_INT	int	INTEGER(KIND=4)
C_LONG	long int	INTEGER (KIND=4 or 8)
C_LONG_LONG	long long int	INTEGER(KIND=8)
C_SIGNED_CHAR	signed char, unsigned char	INTEGER(KIND=1)
C_SIZE_T	size_t	INTEGER(KIND=4 or 8)
C_INT8_T	int8_t	INTEGER(KIND=1)
C_INT16_T	int16_t	INTEGER(KIND=2)
C_INT32_T	int32_t	INTEGER(KIND=4)
C_INT64_T	int64_t	INTEGER(KIND=8)
C_FLOAT	float	REAL(KIND=4)
C_DOUBLE	double	REAL(KIND=8)
C_LONG_DOUBLE	long double	REAL(KIND=8 or 16)
C_FLOAT_COMPLEX	float _Complex	COMPLEX(KIND=4)
C_DOUBLE_COMPLEX	double _Complex	COMPLEX(KIND=8)
C_LONG_DOUBLE_COMPLEX	long double _Complex	COMPLEX(KIND=8 or 16)
C_BOOL	_Bool	LOGICAL(KIND=1)
C_CHAR	char	CHARACTER(LEN=1)

Table 4.1: Table of named constants in ISO_C_BINDING and their equivalent typed in C and Fortran

When shared routines are compiled for various precisions, the programmer can use parameters to denote the precision of the arguments in the interface. These parameters can then be set to the appropriate values from ISO_C_BINDING at one central location. In the example codes that will follow we have set these parameters in the module `sangoma_base`.

```
module sangoma_base
   use, intrinsic :: ISO_C_BINDING
```

```
    implicit none

    integer, parameter :: REALPREC=C_DOUBLE
    integer, parameter :: INTPREC=C_INT

end module sangoma_base
```

4.3.5 Strings

Strings are handled different in Fortran than in C. The difference lies in the way the length of a string is handled. Both in C and fortran, a string is a consecutive block of memory containing characters. The length of this memory block (string length) is undetermined in C. A null character will mark the end of the string. In Fortran, the length of a string is fixed. Unused elements are marked with blanks. Strings are passed in Fortran to routines as two arguments, a pointer to the string and a (hidden) argument denoting the length, where strings are passed in C by only the pointer.

The handing of string arguments in routines will involve conversion between C and Fortran strings. Handing of C strings in the interface is fully supported by the ISO_C_BINDING of Fortran 2003. Handling of Fortran strings is not, which means we have to solve it in a compiler dependent way.

The dilemma is that limiting to C-strings in the interface is portable but very inconvenient for Fortran systems because it results in a double conversion from Fortran to C and back for each call. Therefore we propose that shared routines, having string arguments contain a wrapper routine as well for the C-strings. For callback routines this is not possible. In that case we propose to only allow C-strings.

4.3.6 Logicals/booleans

Logicals can be passed in shared routines. However most C code will use integers for logicals and not the _Bool type (which is part of the C9X extension to the C language). We propose to allow logicals to be part of shared routines but advice to use integers instead (0 to denotes .false.; 1 to denotes .true.).

4.3.7 Header files

The C binding does unfortunately not generate C-header files. These header files allow the C-compiler to check correct calling to the shared routines. Writing header files for shared routines is not obligatory but it would be very nice when the programmers of a shared routine provide them.

4.3.8 Call-back functions

Call-back functions are allowed in the interface. Call-back functions can be used for operations on complex data which cannot be passed easy trough the interface. The following example code shows the use of call-back functions.

The shown module contains a shared routine that uses a callback function:

```
module sangoma_callback

    use, intrinsic :: ISO_C_BINDING
    use sangoma_base, only:REALPREC, INTPREC
    implicit none

contains

   subroutine some_operation(x, n, f_callback) &
              bind(C,name="callback_some_operation")

      use, intrinsic :: ISO_C_BINDING
      implicit none

      integer(INTPREC), value, intent(in) :: n
      real(REALPREC),          intent(in) :: x(n)

      interface
         subroutine f_callback(x,n) bind(C)
            use, intrinsic :: ISO_C_BINDING
            use sangoma_base, only:REALPREC, INTPREC

            integer(INTPREC), value, intent(in) :: n
            real(REALPREC),          intent(in) :: x(n)
         end subroutine
      end interface

      call f_callback(x,n)
   end subroutine

end module
```

This shared routine is used both from C and from Fortran in the following two example codes.

```
subroutine my_f_callback(x,n) bind(C)

   use, intrinsic :: ISO_C_BINDING
   use sangoma_base, only: REALPREC, INTPREC
   implicit none

   integer(INTPREC), value, intent(in) :: n
   real(REALPREC),          intent(in) :: x(n)

   print *,'Welcome in my F90 callback function'
   print *,'n=',n,'x(3)=',x(3)

end subroutine
```

```
program my_callback

   use sangoma_base, only: REALPREC, INTPREC
   use sangoma_callback, only: some_operation

   implicit none
   external my_f_callback
   real(REALPREC)   :: x(10)
   integer(INTPREC) :: n=10

   call some_operation(x,n,my_f_callback)

end program my_callback
```

Or a C version of a similar program

```
#include<stdio.h>

void my_callback(float *x, int n){
  printf("Welcome in my C-callback function\n");
  printf("n=%d\n",n);
  printf("x[2]=%f\n",x[2]);
}

int main(){
   float x[10];
   int n=10;

   x[2]=123.0;
   callback_some_operation(x,n, &my_callback);
}
```

4.3.9 Mini tutorial on C binding in fortran

In this section we will give a brief overview of the C-binding functionality in Fortran 2003. It is not our intention to fully explain all details. The purpose of this section is to give some idea of what needs to be done in the Fortran code to make the C-binding work.

Let us start with an example of a Fortran routine that makes use of the C-binding.

```
module sangoma_pod

   use, intrinsic :: ISO_C_BINDING
   use sangoma_base, only: REALPREC, INTPREC
   implicit none
   contains

   subroutine eigvals(ens,n,m,eig) bind(C,name="sangoma_pod_eigvals")

      integer(INTPREC), value, intent(in)  :: m, n
      real(REALPREC),          intent(in)  :: ens(n,m)
      real(REALPREC),          intent(out) :: eig(m)

      !  Some code here

   end subroutine eigvals

   ! some more routines

end module sangoma_pod
```

The code is programmed normally like any other fortran module. There are a few differences:

- `use, intrinsic :: ISO_C_BINDING`: The module containing the C-binding utilities
- `bind(C,name="sangoma_pod_eigvals")`: This will tell the compiler that the routine must be C-callable. `name` denotes the C-name of the routine.
- `value` attribute: This indicates that this argument is passed by value from C.
- `integer(INTPREC)` and `real(REALPREC)`: These type specifications make sure that the used data type in Fortran corresponds to the data types in C. The precision parameters are define in the module `sangoma_base`.

Fortran users do not notice much of the extensions when using a shared routine as shown from the following example because all interfacing information is automatically handled by the module file generated by the compiler.

```
program call_shared_tool

   use sangoma_pod, only eigvals
   implicit none

   integer, parameter :: m=20
   integer, parameter :: n=1000
   real :: ens(n,m)
   real :: eig(m)

   call eigvals(end,n,m,eig)

end program call_shared_tool
```

4.4 Standard for subroutine interface description

In order to simplify the use of the tools implemented in SANGOMA, the interface arguments should be well documented already in the source code itself. This implies that the (intent) attribute is used, that that a comment describes each argument. In addition, the function or subroutine header should include a description of the functionality and use of a tool. Also, each file should include a line showing the revision number of the file in the Subversion repository.

As an example, consider a routine that is supplied with the ensemble array and might be doing 'some' work with the array (e.g. it might compute the estimate variance of the ensemble). Without C-binding a very simple form of the routine header might look as follows:

```
!$id: $
subroutine work_on_ens(nstate, nens, ens)

   ! This routine operates on the ensemble array.
   ! It can do whatever it likes...

   use sangoma_base, only: REALPREC, INTPREC

   implicit none

   integer(INTPREC), intent(in)  :: nstate ! State dimension
   integer(INTPREC), intent(in)  :: nens   ! Size of ensemble
   real(REALPREC), intent(inout) :: ens(nstate,nens) ! Ensemble array

   ... Code ...

end subroutine work_on_ens
```

This exapample demonstrated the following:

- "`!$id: $`" is the first line of code. In this line, Subversion will add the revision number when the file is committed to the repository.
- The example shows Fortran statements in lowercase characters. However, for SANGOMA we should be flexible and allow for both lower and upper case.
- All arguments are given with `intent` and are followed by a comment describing the argument
- The list of arguments is followed by the description of the routine.
- "`implicit none`" should always be used.
- The lines should be indented to show the logical structure of the program.
- Use association (`use some module`) should always be used with "`only:`".
- Free form source code should be used, but is not mandatory.

- The precision of integer and floating point variables is specified by `INTPREC` and `REALPREC`. These parameters are specified in the module `sangoma_base` (see section 4.3.4). Using these parameters gives the flexibility to support for single and double precision codes.

For the C-binding, the header has to be modified to

```
!$id: $
subroutine work_on_ens(nstate, nens, ens)  bind(C,name="work\_on\_ens")

   ! This routine operates on the ensemble array.
   ! It can do whatever it likes...

   use, intrinsic :: ISO_C_BINDING
   use sangoma_base, only: REALPREC, INTPREC

   implicit none

   integer(INTPREC), intent(in), value :: nstate ! State dimension
   integer(INTPREC), intent(in), value :: nens   ! Size of ensemble
   real(REALPREC), intent(inout)       :: ens(nstate,nens) ! Ensemble array

   ... Code ...

end subroutine work_on_ens
```

Chapter 5

Appendix

5.1 Example of a NetCDF header

This appendix shows the header of a NetCDF file produced by the UK Met Office within MyOcean (see http://www.myocean.eu.org/). The standard tool `ncdump -h <filename>` can be used to produce an ASCII dump of the header as shown here. For compactness, the header is reduced to a single physical field. Also the string length in global attributes has been shortened to let the example fit into the page.

```
netcdf MetO-NWS-PHYS-dm-Agg_1338362551845 {
dimensions:
   time = 2 ;
   depth = 24 ;
   lat = 166 ;
   lon = 91 ;
variables:
   float lon(lon) ;
      lon:standard_name = "longitude" ;
      lon:units = "degrees_east" ;
      lon:long_name = "longitude" ;
      lon:nav_model = "Default grid" ;
      lon:axis = "X" ;
      lon:_CoordinateAxisType = "Lon" ;
      lon:valid_min = -1.000214f ;
      lon:valid_max = 8.999739f ;
   double time(time) ;
      time:standard_name = "time" ;
      time:units = "seconds since 2011-04-07 00:00:00" ;
      time:calendar = "Gregorian" ;
      time:long_name = "Validity time" ;
      time:data_time = 86400.f ;
      time:axis = "T" ;
      time:_CoordinateAxisType = "Time" ;
      time:valid_min = 36244800. ;
      time:valid_max = 36331200. ;
   short vosaline(time, depth, lat, lon) ;
```

```
        vosaline:_CoordinateAxes = "time depth lat lon " ;
        vosaline:_FillValue = -32768s ;
        vosaline:missing_value = -32768s ;
        vosaline:scale_factor = 0.001f ;
        vosaline:add_offset = 30.f ;
        vosaline:standard_name = "sea_water_salinity" ;
        vosaline:long_name = "Sea Water Salinity" ;
        vosaline:units = "1e-3" ;
    float lat(lat) ;
        lat:standard_name = "latitude" ;
        lat:units = "degrees_north" ;
        lat:long_name = "latitude" ;
        lat:nav_model = "Default grid" ;
        lat:axis = "Y" ;
        lat:_CoordinateAxisType = "Lat" ;
        lat:valid_min = 49.00001f ;
        lat:valid_max = 60.00001f ;
    float depth(depth) ;
        depth:axis = "Z" ;
        depth:standard_name = "depth" ;
        depth:units = "m" ;
        depth:positive = "down" ;
        depth:long_name = "depth" ;
        depth:_CoordinateAxisType = "Height" ;
        depth:_CoordinateZisPositive = "down" ;
        depth:valid_min = 0.f ;
        depth:valid_max = 5000.f ;

// global attributes:
        :title = "North West European Shelf from UK Met Office Model FOAM 7 km" ;
        :institution = "UK Met Office" ;
        :references = "http://www.ncof.co.uk" ;
        :source = "UK Met Office Operational Suite, FOAM 7 km run 2012-05-29" ;
        :Conventions = "CF-1.0" ;
        :history = "Data extracted from dataset http://data.ncof.co.uk/..." ;
        :time_min = 36244800. ;
        :time_max = 36331200. ;
        :julian_day_unit = "seconds since 2011-04-07 00:00:00" ;
        :z_min = 0. ;
        :z_max = 5000. ;
        :latitude_min = 49.0000076293945 ;
        :latitude_max = 60.0000114440918 ;
        :longitude_min = -1.00021362304688 ;
        :longitude_max = 8.9997386932373 ;
}
```