Model Interfacing

Jedi Academy IV, Monterey CA

26th February 2020

Outline

- Introduction
- Model space classes
- GetValues
- Building an application
- LinkedLists (not presented)

Introduction

OOPS provides the algorithms that combine generic building blocks into applications such as variational assimilation, forecast, EnKF, FSOI etc.

OOPS (by design) knows nothing about the actual implementation of the building blocks and carries no information about the underlying data. The classes that need to be implemented for a specific model are called interface classes.

Often models are written in Fortran so a mixed language approach is required and a binding between the languages is implemented.

Once the interface to a specific model is ready it can be used to create applications by passing traits and information about factories.

Models being interfaced to JEDI

MODEL	TYPE	INTERFACE	CENTER
FV3GFS	Atmosphere	fv3-jedi	NOAA-EMC
GEOS	Atmosphere	fv3-jedi	NASA-GMAO
FV3GFS GSDChem	Atmospheric chemistry	fv3-jedi	NOAA-ESRL
GEOS-AERO	Atmospheric aerosols	fv3-jedi	NASA-GMAO
MPAS	Atmosphere	mpas	NCAR
WRF	Atmosphere	wrf-jedi	NCAR
LFRic	Atmosphere	lfric	Met Office (UK)
MOM6	Ocean	soca	NOAA-EMC
SIS2	Sea ice	soca	NOAA-EMC
CICE6	Sea ice	soca-cice6	NOAA-EMC
NEPTUNE	Atmosphere	neptune	NRL
QG	Toy model	oops	ECMWF
Lorenz 95	Toy model	oops	ECMWF
ShallowWater	Toy model	shallow-water	NOAA-ESRL

Model space classes

Geometry Class: OOPS vs. FV3-JEDI

OOPS (GENERIC)

```
template <typename MODEL>
class Geometry  {
  typedef typename MODEL::Geometry Geometry_;

 public:

  Geometry(const eckit::Configuration &, const eckit::mpi::Comm &);
  Geometry(const Geometry &);
  explicit Geometry(boost::shared_ptr<const Geometry_>);
  ~Geometry();

  /// Interfacing
  const Geometry_ & geometry() const {return *geom_;}

 private:
  boost::shared_ptr<const Geometry_> geom_;
};
```

MODEL (SPECIFIC)

```
class Geometry : public util::Printable,
                 private util::ObjectCounter<Geometry> {
 public:
  static const std::string classname() {return "fv3jedi::Geometry";}

  explicit Geometry(const eckit::Configuration &, const eckit::mpi::Comm &);
  Geometry(const Geometry &);
  ~Geometry();

  GeometryIterator begin() const;
  GeometryIterator end() const;

  F90geom & toFortran() {return keyGeom_;}
  const F90geom & toFortran() const {return keyGeom_;}
  const eckit::mpi::Comm & getComm() const {return comm_;}

 private:
  Geometry & operator=(const Geometry &);
  void print(std::ostream &) const;
  F90geom keyGeom_;
  const eckit::mpi::Comm & comm_;
};
```

Geometry method

C++ Model, State, Increment etc.

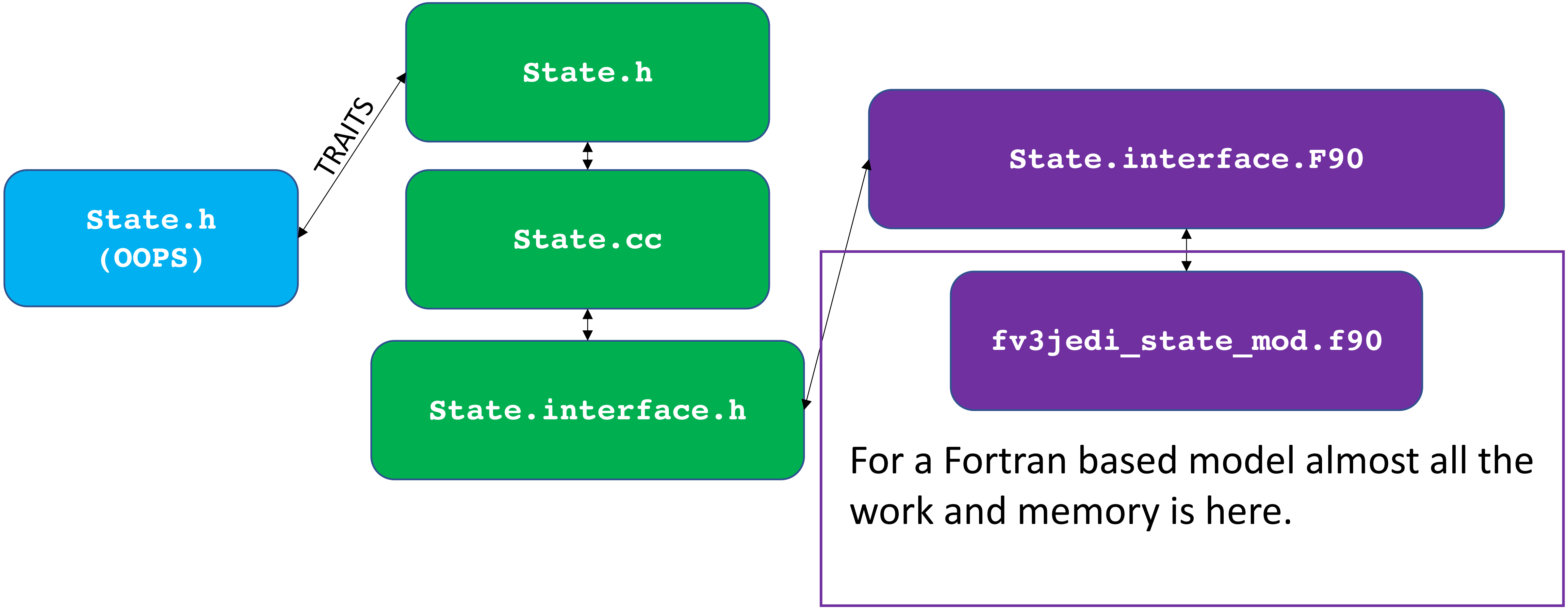
```
Geometry::Geometry(const eckit::Configuration & conf, const eckit::mpi::Comm & comm) : comm_(comm) {
  const eckit::Configuration * configc = &conf;

  dx_ = conf.getInt("dx")
  dy_ = conf.getInt("dy")

}
```

Fortran Model, State, Increment etc.

```
Geometry::Geometry(const eckit::Configuration & conf, const eckit::mpi::Comm & comm) : comm_(comm) {
  const eckit::Configuration * configc = &conf;

  interface(keyGeom_, &configc, &comm_);

}
```

C++ to Fortran Binding Files

All the model (and UFO) classes follow basically the same file structure for the mixed C++/Fortran languages:



Binding

```
Geometry::Geometry(const eckit::Configuration & conf, const eckit::mpi::Comm & comm) : comm_(comm) {
  const eckit::Configuration * configc = &conf;

  interface(keyGeom_, &configc, &comm_);

}
```

Integer locating this geometry object

Pointer to configuration object

Pointer to communicator object

Config binding

```fortran
subroutine c2f_binding(c_key_self, c_comm, c_conf, c_vars)

integer(c_int), intent(in) :: c_key_self
type(c_ptr),    intent(in) :: c_conf
type(c_ptr),    intent(in) :: c_comm
type(c_ptr),    intent(in) :: c_vars

type(interface)           :: f_self
type(fckit_configuration) :: f_conf
type(fckit_mpi_comm)      :: f_comm
type(oops_variables)      :: f_vars

call fv3jedi_geom_registry%get(c_key_self,f_self)

f_comm = fckit_mpi_comm(c_comm)
f_conf = fckit_configuration(c_conf)
f_vars = oops_variables(c_vars)

call f_self%implementation(f_comm, f_conf, f_vars)

end subroutine c2f_binding
```

```fortran
subroutine implementation(comm, conf, vars)

class(interface)         , intent(inout) :: self
type(fckit_configuration), intent(in)    :: conf
type(fckit_mpi_comm)     , intent(in)    :: comm
type(oops_variables)     , intent(in)    :: vars

...
call conf%get_or_die("yaml_input", self%data)


end subroutine implementation
```

The implementation is pure Fortran and is where the work is done

The interface is constructed once, potentially just by copying from some other model.

Example Fortran interfaces for Geometry

```fortran
module fv3jedi_geom_mod

implicit none
private
public :: fv3jedi_geom
public :: create, clone, delete, info
! ----------------------------------------
type :: fv3jedi_geom
 integer :: something
end type fv3jedi_geom
! ----------------------------------------
contains

subroutine create(self, c_conf)
implicit none
type(fv3jedi_geom), intent(inout) :: self
type(c_ptr),        intent(in)    :: c_conf
end subroutine create
```

```fortran
subroutine clone(self, other)
implicit none
type(fv3jedi_geom), intent(in   ) :: self
type(fv3jedi_geom), intent(inout) :: other
end subroutine clone

subroutine delete(self)
implicit none
type(fv3jedi_geom), intent(inout) :: self
end subroutine delete

end module fv3jedi_geom_mod
```

Geometry Class: Fortran Type

```
!> Fortran derived type to hold geometry data for the FV3JEDI model
type :: fv3jedi_geom
  integer :: isd, ied, jsd, jed                               !data domain
  integer :: isc, iec, jsc, jec                               !compute domain
  integer :: npx,npy,npz                                      !x/y/z-dir grid edge points per tile
  integer :: layout(2)                                        !Processor layout for computation
  integer :: io_layout(2)                                     !Processor layout for read/write
  integer :: halo                                             !Number of halo points, normally 3
  character(len=255) :: nml_file                              !FV3 nml file associated with this geom
  integer :: size_cubic_grid                                  !Size of cubed sphere grid (cell center)
  type(domain2D) :: domain                                    !MPP domain
  integer :: ntile                                            !Tile ID
  integer :: ntiles = 6                                       !Number of tiles, always 6
  integer :: stackmax                                         !Stackmax
  real(kind=kind_real), allocatable :: grid_lon(:,:)          !Longitude at cell center
  real(kind=kind_real), allocatable :: grid_lat(:,:)          !Latitude at cell center
  real(kind=kind_real), allocatable :: egrid_lon(:,:)          !Longitude at cell center
  real(kind=kind_real), allocatable :: egrid_lat(:,:)          !Latitude at cell center
  real(kind=kind_real), allocatable :: area(:,:)              !Grid area
  real(kind=kind_real), allocatable :: ak(:),bk(:)            !Model level coefficients
  real(kind=kind_real) :: ptop                                !Pressure at top of domain
end type fv3jedi_geom
```

Geometry Class: Fortran Methods

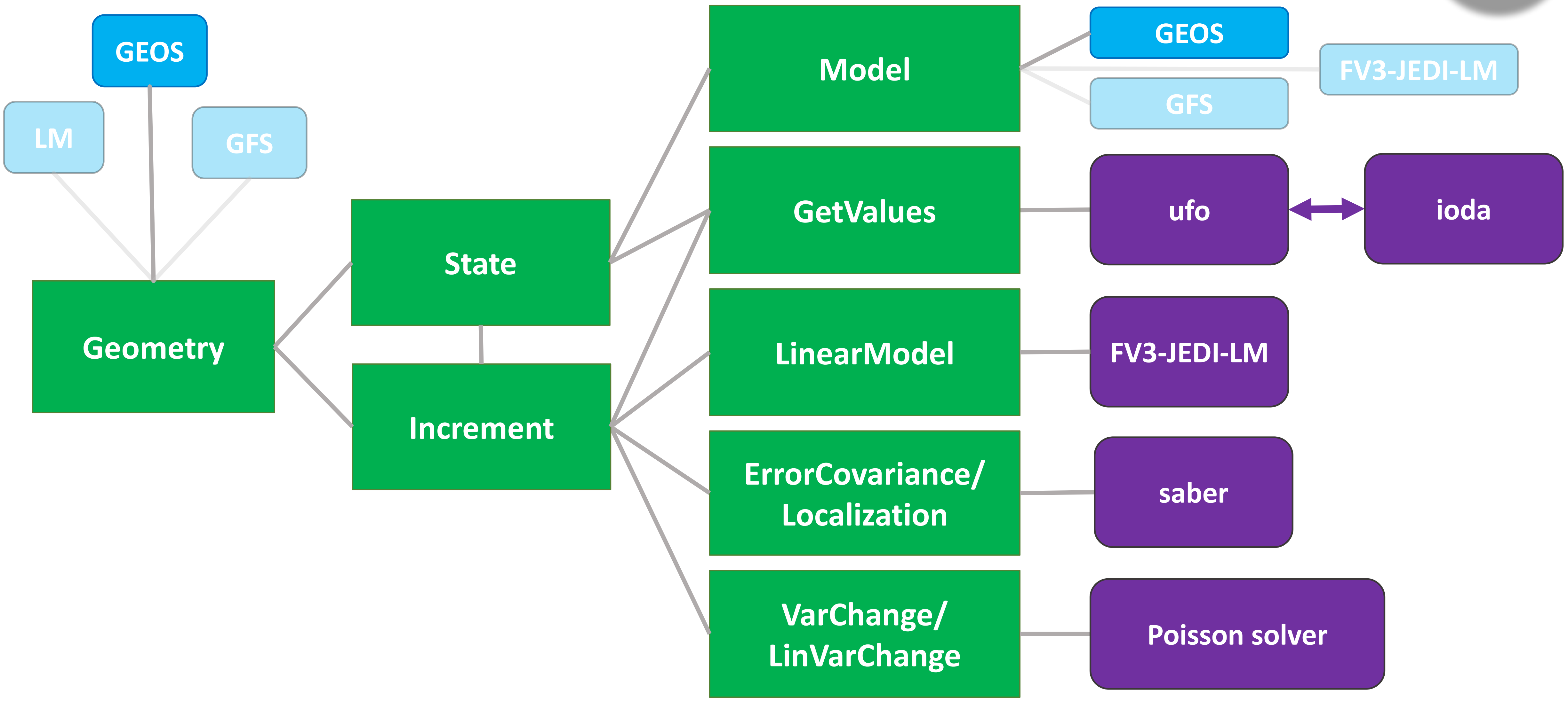
```
subroutine create(self, c_conf)

implicit none

!Arguments
type(fv3jedi_geom), intent(inout) :: self
type(c_ptr),        intent(in)    :: c_conf

!Locals
type(fv_atmos_type), allocatable      :: FV_Atm(:)

!Intialize using the model setup routine
call fv_init(FV_Atm, 300.0_kind_real, grids_on_this_pe, p_split)
deallocate(pelist_all)

self%isc = FV_Atm(1)%bd%isc
self%iec = FV_Atm(1)%bd%iec
self%jsc = FV_Atm(1)%bd%jsc
self%jec = FV_Atm(1)%bd%jec
self%ntile  = FV_Atm(1)%tile

self%npx = FV_Atm(1)%npx
self%npy = FV_Atm(1)%npy
self%npz = FV_Atm(1)%npz
self%layout(1) = FV_Atm(1)%layout(1)
self%layout(2) = FV_Atm(1)%layout(2)

!Lat,lon and area from
allocate ( self%area(self%isd:self%ied, self%jsd:self%jed) )
allocate ( self%grid_lon(self%isd:self%ied, self%jsd:self%jed) )
allocate ( self%grid_lat(self%isd:self%ied, self%jsd:self%jed) )

self%area = FV_Atm(1)%gridstruct%area_64
self%grid_lon = real(FV_Atm(1)%gridstruct%agrid_64(:,:,1),kind_real)
self%grid_lat = real(FV_Atm(1)%gridstruct%agrid_64(:,:,2),kind_real)

end subroutine create
```

Call to dynamical core

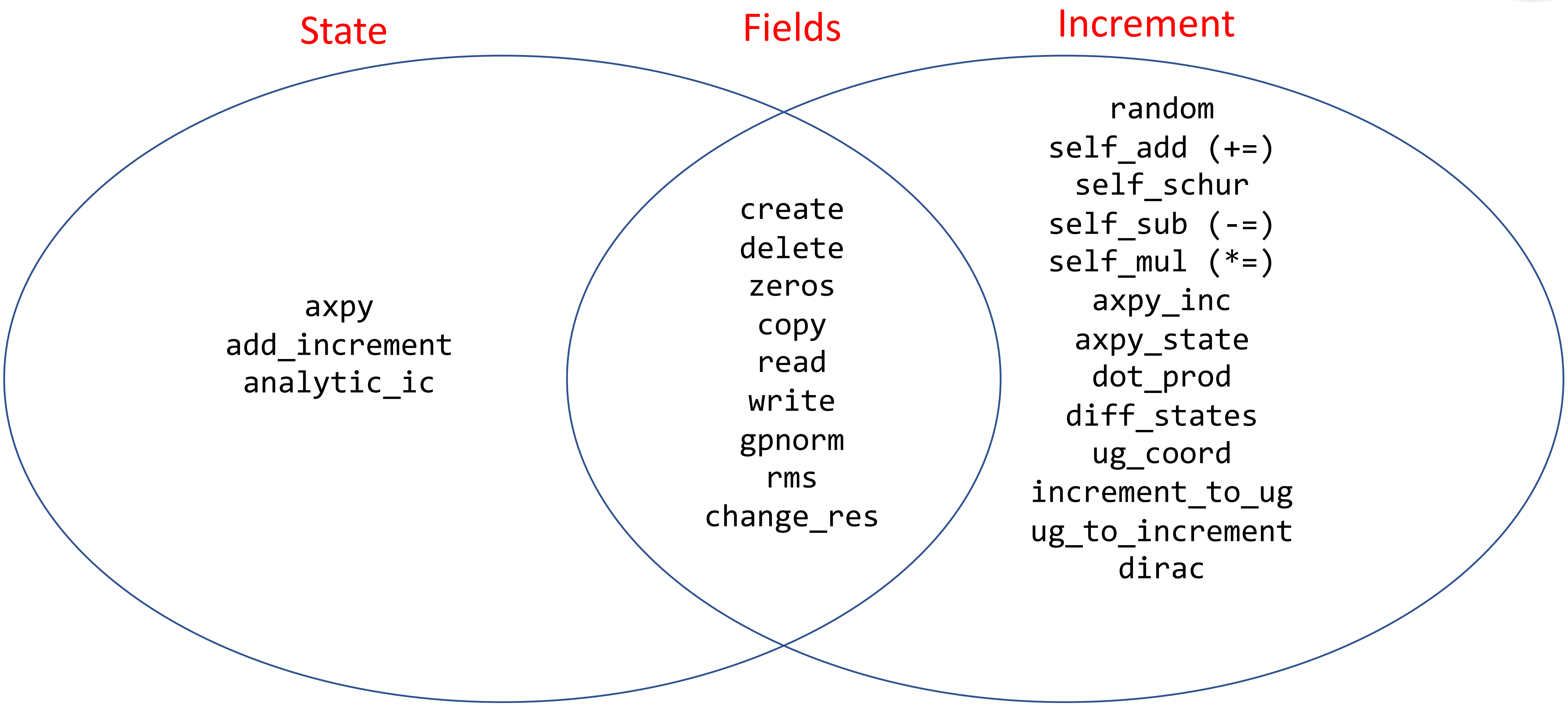
```
subroutine clone(self, other)

implicit none

type(fv3jedi_geom), intent(in    ) :: self
type(fv3jedi_geom), intent(inout) :: other

allocate(other%grid_lon(self%isd:self%ied, self%jsd:self%jed))
allocate(other%grid_lat(self%isd:self%ied, self%jsd:self%jed))

other%npx               = self%npx
other%npy               = self%npy
other%npz               = self%npz
other%layout            = self%layout
other%grid_lon          = self%
other%grid_lat          = self%

end subroutine clone
```

```
subroutine delete(self)

implicit none

type(fv3jedi_geom), intent(inout) :: self

! Deallocate
deallocate(self%grid_lon)
deallocate(self%grid_lat)
deallocate(self%egrid_lon)
deallocate(self%egrid_lat)
deallocate(self%area)
deallocate(self%ak)
deallocate(self%bk)

call mpp_deallocate_domain(self%domain)

end subroutine delete
```

Dependency structure



State and Increment: Methods



Example Fortran Field Class

The concept of fields is introduced in order to limit duplicate code across sate and increment.

Some model interfaces implement fields at the C++ level, e.g. qg, l95, soca. Some do so at the Fortran level, e.g. fv3-jedi, mpas, lfric.

```fortran
!Field type
type :: fv3jedi_field
 logical :: lalloc = .false.
 character(len=32) :: short_name = "null"   !Short name (to match file name)
 character(len=10) :: fv3jedi_name = "null" !Common name
 character(len=64) :: long_name = "null"    !More descriptive name
 character(len=32) :: units = "null"        !Units for the field
 logical :: tracer = .false.
 integer :: staggerloc   !Middle, corners, east, south, etc
 integer :: isc, iec, jsc, jec, npz
 real(kind=kind_real), allocatable :: array(:,:,:)
 contains
  procedure :: allocate_field
  procedure :: array_pointer
  procedure :: equals
  generic :: assignment(=) => equals
  procedure :: deallocate_field
endtype fv3jedi_field
```

State/Increment variables

```
type :: fv3jedi_state

  !Local copies of grid for convenience
  integer :: isc, iec, jsc, jec
  integer :: npx, npy, npz
  integer :: ntiles, ntile
  logical :: hydrostatic = .true.
  integer :: calendar_type, date_init(6)
  integer :: nf
  logical :: have_agrid
  logical :: have_dgrid

  type(fv3jedi_field), allocatable :: fields(:)

end type fv3jedi_state
```

State/Increment constructor

```fortran
subroutine create(self, geom, vars)

implicit none
type(fv3jedi_state), intent(inout) :: self
type(fv3jedi_geom),  intent(in)    :: geom
type(oops_vars),     intent(in)    :: vars

integer :: var, vcount

self%nf = vars%nv

allocate(self%fields(self%nf))

vcount = 0
do var = 1, vars%nv
   select case (trim(vars%fldnames(var)))
     case("ud","u","U")
       vcount=vcount+1;
       call self%fields(vcount)%allocate_field(geom%isc,geom%iec,geom%jsc,geom%jec,geom%npz, &
            short_name = vars%fldnames(var), long_name = 'eastward_wind_on_native_D-Grid', &
            fv3jedi_name = 'ud', units = 'm s-1', staggerloc = north, arraypointer = self%ud )
     case("...")
       ...
     case default
       call abor1_ftn("Create: unknown variable "//trim(vars%fldnames(var)))
   end select
enddo


end subroutine create
```

Incoming vars are decided by the user at run time.

Variables are pre-programmed but not hardwired

State/Increment method

```
subroutine self_schur(self,rhs)

implicit none
type(fv3jedi_increment), intent(inout) :: self
type(fv3jedi_increment), intent(in)    :: rhs

integer :: var

call checksame(self%fields,rhs%fields,"fv3jedi_increment_mod.self_schur")

do var = 1,self%nf
  self%fields(var)%array = self%fields(var)%array * rhs%fields(var)%array
enddo

end subroutine self_schur
```

Optionally check same list of fields in self and rhs

Loop through all allocated fields. Not dependent on variables chosen.

The Forecast Model

- Jedi is designed to work with the Model *in-core*. That is to say that JEDI will drive the model through the assimilation window exchanging states as it goes.

- This is often one of the hardest parts of interfacing JEDI to a particular forecast model.

- Forecast models have not necessarily been developed in a way that exposes a stepping method, the model states themselves and with an ability to 'rewind', as is needed for outer loops.

- Political issues can also present themselves.

Model class

```
// ------------------------------------------------------------------------------
static oops::ModelMaker<FV3JEDITraits, ModelGEOSFV3JEDI> makermodel_("GEOS");
// ------------------------------------------------------------------------------
ModelGEOSFV3JEDI::ModelGEOSFV3JEDI(const GeometryFV3JEDI & resol,
                            const eckit::Configuration & mconf)
  : keyConfig_(0), tstep_(0), geom_(resol), vars_(mconf)
{
  oops::Log::trace() << "ModelGEOSFV3JEDI::ModelGEOSFV3JEDI" << std::endl;
  tstep_ = util::Duration(mconf.getString("tstep"));
  const eckit::Configuration * configc = &mconf;

  // JEDI to GEOS directory
  getcwd(jedidir_, 10000);

  std::string sGEOSSCRDIR = mconf.getString("GEOSSCRDIR");
  strcpy(geosscrdir_, sGEOSSCRDIR.c_str());
  chdir(geosscrdir_);

  // Create the model
  fv3jedi_geos_create_f90(&configc, geom_.toFortran(), keyConfig_);

  // GEOS to JEDI directory
  chdir(jedidir_);

  oops::Log::trace() << "ModelGEOSFV3JEDI created" << std::endl;
}
```

Factory name

Model class

```
type :: geos_model
  type(MAPL_Cap) :: cap
  integer :: GEOSsubsteps
end type geos_model
```

GEOS

```
type :: nems_model
 type(ESMF_GridComp) :: esmComp !NUOPC driver
 type(ESMF_State)    :: importState, exportState
 type(ESMF_Clock)    :: clock
 integer :: dt
 character(len=20) :: startTime
end type nems_model
```

NEMSfv3gfs

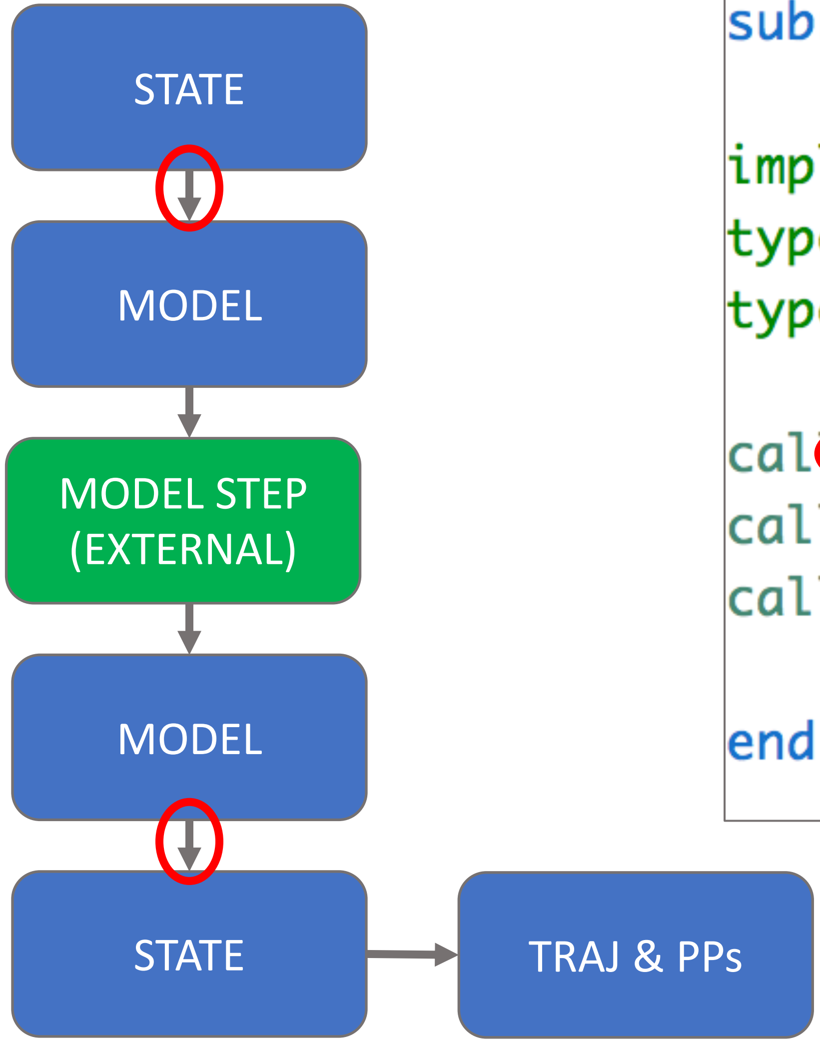
```
type :: fv3_model
  type(fv3jedi_lm_type)  :: fv3jedi_lm
end type fv3_model
```

Dynamical core only

```
type :: pseudo_model
  character(len=255)  :: pseudo_type !geos of gfs
  character(len=255)  :: pseudo_path
  character(len=255)  :: pseudo_file
end type pseudo_model
```

Pseudo model

Data flow



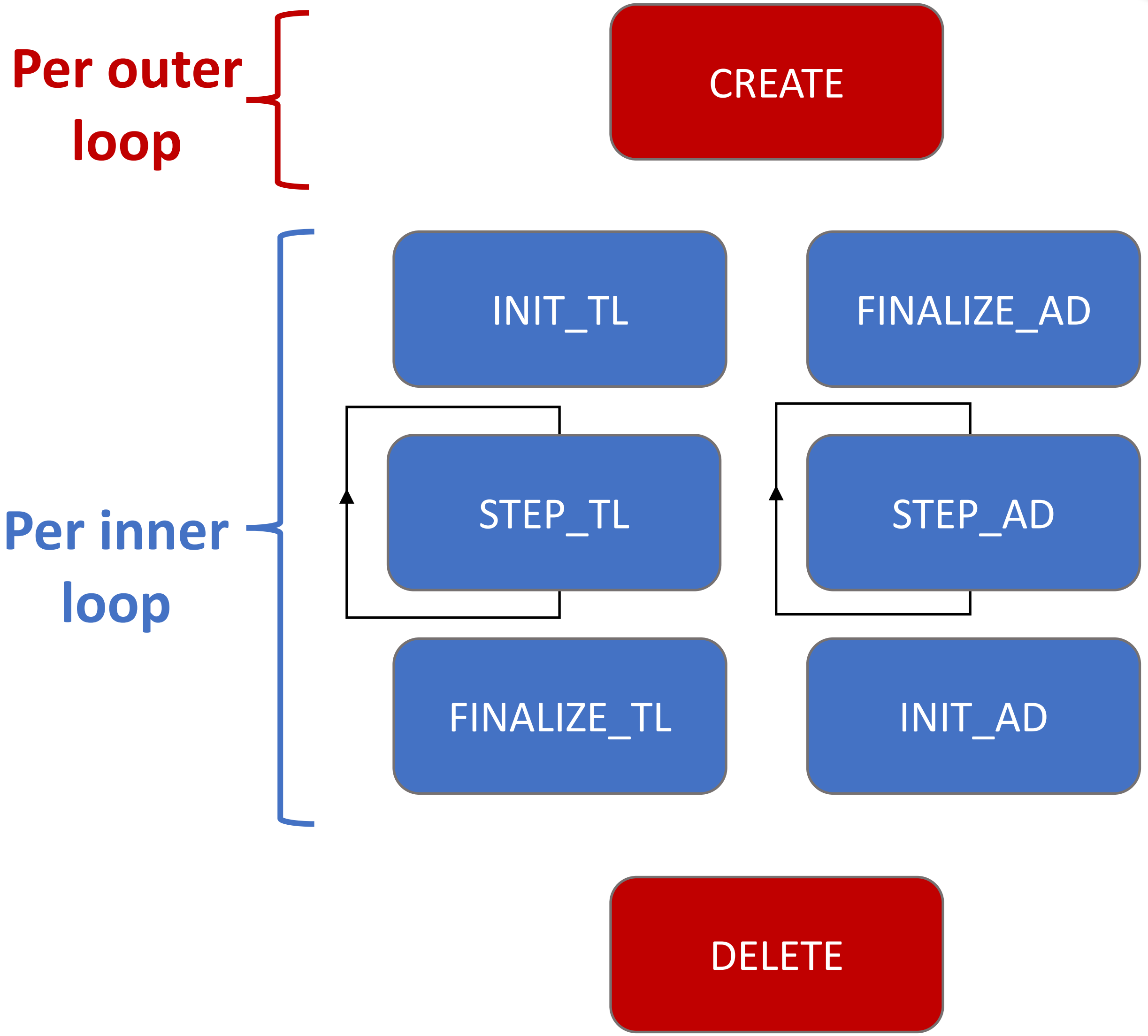

```fortran
subroutine model_step(self, state)

implicit none
type(fv3jedi_model), intent(inout) :: self
type(fv3jedi_state), intent(inout) :: state

call state_to_lm(state,self%fv3jedi_lm)
call self%fv3jedi_lm%step_nl()
call lm_to_state(self%fv3jedi_lm,state)

end subroutine model_step
```

Can be pointer, move or copy. Usually a copy to account for differences in precision

LinearModel class

```
type:: fv3jedi_tlm
  type(fv3jedi_lm_type) :: fv3jedi_lm
end type fv3jedi_tlm
```

```
public :: fv3jedi_tlm
public :: tlm_create
public :: tlm_delete
public :: tlm_initialize_tl
public :: tlm_initialize_ad
public :: tlm_step_tl
public :: tlm_step_ad
public :: tlm_finalize_tl
public :: tlm_finalize_ad
```



Variable changes

Incremental hybrid-4DVar involves a number of linear and nonlinear variable transforms:

$$\frac{\partial J}{\partial \delta \mathbf{x}_0} = \mathbf{B}^{-1}\left(\delta \mathbf{x}_0 - \delta \mathbf{x}_b\right) - \sum_{k=0}^{K} K_m^\top \mathbf{M}_k^\top K_h^\top \mathbf{H}^\top \mathbf{R}_k^{-1}\left(\mathbf{d}_k - \mathbf{H} K_h \delta \mathbf{x}_k\right)$$

$$\delta \mathbf{x}_k = \mathbf{M}_{t_{k-1}\to t_k} \mathbf{M}_{t_{k-1}\to t_{k-2}} \ldots \mathbf{M}_{t_0 \to t_1} K_m \delta \mathbf{x}_0$$

$$\mathbf{d}_k = \mathbf{y}_k^o - h\left(k_h\left\{m_{t_0\to t_k}\left[k_m\left(\mathbf{x}_0\right)\right]\right\}\right)$$

$$B = K_b D C D K_b^\top$$

Variable changes

Sets of variables:

Background: the variables which you end up with an analysis of. Typically chosen to interact well with the forecast model being restarted.

Control increment: the variables of $\delta \mathbf{x}_0$, chosen based on various considerations.

Model: the variables that the model and linear model need, e.g. staggered winds.

B matrix variables: the variables used in the B matrix, e.g. unbalanced stream function and velocity potential.

VarChaC2MFV3JEDI

```
public :: fv3jedi_varcha_c2m
public :: create
public :: delete
public :: multiply
public :: multiplyadjoint
public :: multiplyinverse
public :: multiplyinverseadjoint
```

Increment containing control variables comes in, increment with model variables goes out. The base class handles the allocation and deallocation either side.

```
subroutine multiply(self,geom,xctl,xmod)

implicit none
type(fv3jedi_varcha_c2m),   intent(in)    :: self
type(fv3jedi_geom), target, intent(inout) :: geom
type(fv3jedi_increment),    intent(inout) :: xctl
type(fv3jedi_increment),    intent(inout) :: xmod

!Ps
xmod%ps = xctl%ps

!Tracers
xmod%qi = xctl%qic
xmod%ql = xctl%qlc
xmod%o3 = xctl%o3c

!Tangent linear of analysis (control) to model variables
call control_to_model_tlm(geom, xctl%psi, xctl%chi, xctl%tv, xctl%qc, &
                                xmod%ua , xmod%va , xmod%t , xmod%q , &
                                self%tvtraj,self%qtraj,self%qsattraj )

end subroutine multiply
```

Get Values

GetValues

In order to maintain the separation of concerns the observation operator is split into a model dependent parts and model agnostic part.

$$
\begin{aligned}
y^o &= h(x) \\
&= h_{obs}\left[h_{mod}(x)\right]
\end{aligned}
$$

The model dependent part might involve interpolation, field of view calculations and variable transforms.

The intermediate state after computing the model dependent part of the observation operator are known as GeoVaLs (Geophysical Values at observation Locations).

$$GeoVaLs = h_{mod}(x)$$

These are model states interpolated to observation locations and converted to the variables requested by the observation operator.

GetValues

Fortran class definitions

```
type getvalues
integer :: dummy
contains
 procedure :: create
 procedure :: delete
 procedure :: fill_geovals
end type getvalues
```

```
type getvalues_tlad
integer :: dummy
contains
 procedure :: create
 procedure :: delete
 procedure :: set_trajectory
 procedure :: fill_geovals_tl
 procedure :: fill_geovals_ad
end type getvalues_tlad
```

GetValues

```fortran
subroutine create(self, geom, vars, locs)

class(fv3jedi_getvalues), intent(inout) :: self
type(fv3jedi_geom),       intent(in)    :: geom
type(oops_variables),     intent(in)    :: vars
type(ufo_locs),           intent(in)    :: locs

end subroutine create

! ------------------------------------------------------------------------------

subroutine delete(self)

class(fv3jedi_getvalues), intent(inout) :: self

end subroutine delete
```

```fortran
subroutine fill_geovals(self, state, t1, t2, geovals)

class(fv3jedi_getvalues), intent(in)    :: self
type(fv3jedi_state),      intent(in)    :: state
type(datetime),           intent(in)    :: t1
type(datetime),           intent(in)    :: t2
type(ufo_geovals),        intent(inout) :: geovals

end subroutine fill_geovals
```

LinearGetValues

```
subroutine set_trajectory(self, state, t1, t2, geovals)

class(fv3jedi_getvalues_tlad), intent(in)    :: self
type(fv3jedi_state),           intent(in)    :: state
type(datetime),                intent(in)    :: t1
type(datetime),                intent(in)    :: t2
type(ufo_geovals),             intent(inout) :: geovals

end subroutine set_trajectory

! --------------------------------------------------------

subroutine fill_geovals_tl(self, inc, t1, t2, geovals)

class(fv3jedi_getvalues_tlad), intent(in)    :: self
type(fv3jedi_increment),       intent(in)    :: inc
type(datetime),                intent(in)    :: t1
type(datetime),                intent(in)    :: t2
type(ufo_geovals),             intent(inout) :: geovals

end subroutine fill_geovals_tl
```

```
type :: fv3jedi_getvalues_traj
 integer :: bumpid, ngrid
 logical :: noobs
 real(kind=kind_real), allocatable :: t(:,:,:)
 real(kind=kind_real), allocatable :: q(:,:,:)
 type(bump_type) :: bump
 logical :: lalloc = .false.
end type fv3jedi_getvalues_traj
```

GetValues: algorithm

```
Compute weights for interpolation


Loop over UFO variables

  Select case on variable

    Convert variable and prepare interpolation

    Loop over levels
      Interpolate to locations
    end (levels)

End (variables)
```

getValues: prepare state/increment variable

Loop over variables

Flag on whether to interpolate

Set number of levels for variable and interpolation flag.

Transform the variable if need be

Some variables use integration or aren't float.

ABORT

```
do jvar = 1, vars%nv

  do_interp = .false.

  select case (trim(vars%fldnames(jvar)))

  case ("eastward_wind")
    nvl = npz
    do_interp = .true.
    geoval => state%ua
  case ("virtual_temperature")
    nvl = npz
    do_interp = .true.
    call T_to_Tv(geom,state%t,state%q,geovalm)
    geoval => geovalm
  case ("Soil_Type")
    nvl = 1
    do_interp = .false.
    obs_state(:,1) = real(soil_type,kind_real)
  case default
    call abor1_ftn(trim(myname)//"unknown variable: "//trim(vars%fldnames(jvar)))
```

GeoVals

GeoVaLs are not part of the Model Space but currently have to be allocated by the model. This is because one of the dimensions is the number of vertical levels. The plan is to move this to the GeoVaLs constructor at some point and request this from the model geometry instead.

The model only sees GeoVaLs in GetValues so this is where the allocation occurs. E.g.:

```
if (.not.allocated(gom%geovals(jvar)%vals)) then
  gom%geovals(jvar)%nval = gvlev

  allocate(gom%geovals(jvar)%vals(gom%geovals(jvar)%nval,gom%geovals(jvar)%nobs))
  gom%geovals(jvar)%vals = 0.0_kind_real

  if (lastvar) gom%linit  = .true.
endif
```

Interpolation

Interpolation is needed in several places in the model interface. It's required in GetValues for interpolating to observation locations but also in State and Increment, for example to support data assimilation algorithms that support increments at varying resolution.

JEDI provides general unstructured interpolation options via BUMP (B Matrix Unstructured Mesh Package) and via a stand alone unstructured interpolation routine. In the future we also plan to support interpolation using Atlas. In addition each model can implement their own interpolation methods.

Generic interpolation: creating weights

Latitudes and longitudes for both BUMP and unstructured interpolation are unstructured, rank 1 vectors where order is not important. Input lats and lons do not have to be on the same processor as the output lats and lons.

BUMP interpolation from SABER

```
call bump%setup_online( f_comm, ngrid_in, 1, 1, 1, lon_in_us, lat_in_us, area, vunit, lmask, &
                        nobs=ngrid_ou, lonobs=lon_ou, latobs=lat_ou)
```

Unstructured interpolation from OOPS

```
call self%unsinterp%create( geom_in%f_comm, self%nnearest, trim(us_interp_type), &
                            geom_in%ngrid, rad2deg*geom_in%lat_us, rad2deg*geom_in%lon_us, &
                            geom_ou%ngrid, rad2deg*geom_ou%lat_us, rad2deg*geom_ou%lon_us )
```

Generic interpolation: apply

```
call self%bump%apply_obsop(field_in,field_ou)
```

```
call self%unsinterp%apply(field_in, field_ou)
```

Future of model interfacing

As development has evolved it has become clear that it should be possible to make some of the interfacing more generic and share code across models.

MAGIC (Model Agnostic Grid Interface Construct) by Rahul Mahajan explores the possibility of having some components be generic, or using a base class, to limit duplicate work across model interfaces.

One possibility is to leverage the capabilities in the Atlas. That way the Geometry, State and Increment can just be Atlas structures and have identical source code across interfaces.

This will also enable the possibility of a completely generic GetValues class.

Building an application

Building an application driver

Geometry

State

Increment

Model

…

ufo/ioda

If an oops branch with a bug is merged and no one is there to compile it, does the bug really exist?

Traits/ Factories → Driver fv3jediVar.cc → OOPS → fv3jedi_var.x

oops on its own is just headers requiring a template to be applied to the interface classes and be passed via traits and factories.

fv3jediVar.cc application driver

```
#include "FV3JEDITraits.h"
#include "instantiateLocalizationFactory.h"
#include "instantiateFV3JEDIVarChangeFactory.h"
#include "instantiateObsFilterFactory.h"
#include "oops/runs/Variational.h"
#include "RunFV3JEDI.h"

int main(int argc,  char ** argv) {
  fv3jedi::RunFV3JEDI run(argc, argv);
  fv3jedi::instantiateLocalizationFactory();
  fv3jedi::instantiateFV3JEDIVarChangeFactory();
  fv3jedi::instantiateObsFilterFactory();
  oops::Variational<fv3jedi::FV3JEDITraits> var;
  run.execute(var);
  return 0;
}
```

Include the model traits

Include the factories

Include the main application

Pass config (YAML)

Initialization step (FMS etc)

Instantiate factories

Create application object

Execute application

Run

```
class Run : public eckit::Main {
 public:
  Run(int argc, char** argv);
  virtual ~Run();
  void execute(const Application &);

 protected:
  const eckit::Configuration & config() const {return *config_;}

 private:
  boost::scoped_ptr<const eckit::YAMLConfiguration> config_;
  boost::scoped_ptr<util::Timer> timer_;
};
```

Inheritance from the base class Run

This can do generic initialization such as MPI init and prepare generic monitoring tools.

In turn inherits eckit::Main.

Execute runs the application and diagnostics.

Variational.h

```
template <typename MODEL> class Variational : public Application {
  typedef Geometry<MODEL>              Geometry_;
  typedef Model<MODEL>                 Model_;
  typedef State<MODEL>                 State_;

 public:
// -----------------------------------------------------------------------------
  Variational() {
    instantiateCostFactory<MODEL>();
    instantiateCovarFactory<MODEL>();
    instantiateMinFactory<MODEL>();
    instantiateObsErrorFactory<MODEL>();
    instantiateObsFilterFactory<MODEL>();
    instantiateTlmFactory<MODEL>();
    instantiateVariableChangeFactory<MODEL>();
  }
// -----------------------------------------------------------------------------
  virtual ~Variational() {}
// -----------------------------------------------------------------------------
  int execute(const eckit::Configuration & fullConfig) const {
//  Setup resolution
    const eckit::LocalConfiguration resolConfig(fullConfig, "resolution");
    const Geometry_ resol(resolConfig);

//  Setup Model
    const eckit::LocalConfiguration modelConfig(fullConfig, "model");
    const Model_ model(resol, modelConfig);
    Log::trace() << "Variational: model has been set up" << std::endl;
```

Run execute

```
void Run::execute(const Application & app) {
  int status = 1;
  Log::info() << "Run: Starting " << app << std::endl;
  try {
    status = app.execute(*config_);
  }
  catch(const eckit::Exception & e) {
    status = 1;
    Log::error() << e.what() << " caught in "  << Here() << std::endl;
    Log::error() << "Exception: " << app << " terminating..." << std::endl;
    eckit::Exception::exceptionStack(eckit::Log::error(), true);
  }
  catch(const std::exception & e) {
    status = 1;
    Log::error() << "Exception: " << e.what() << std::endl;
    Log::error() << "Exception: " << app << " terminating..." << std::endl;
  }
  catch(...) {
    status = 1;
    Log::error() << "Unknown exception: " << app << " terminating..." << std::endl;
  }
  Log::info() << "Run: Finishing " << app << std::endl;

// Performance diagnostics
  util::ObjectCountHelper::stop();
  util::TimerHelper::stop();

  Log::info() << "Run: Finishing " << app << " with status = " << status << std::endl;
  if (status) ::exit(status);
}
```

Receives and application

Calls application execute

Checking of proper run

Output some diagnostics

FV3-JEDI Traits

OOPS level State.h interface

```
template <typename MODEL>
class State : public util::Printable,
              private util::ObjectCounter<State<MODEL> > {
  typedef typename MODEL::State      State_;

 public:
  State_ & state() {return *state_;}
  const State_ & state() const {return *state_;}

 private:
  boost::scoped_ptr<State_> state_;
};
```

FV3-JEDI Templates passed in through Traits. Basically just a list of implemented classes.

```
struct FV3JEDITraits {
  static std::string name() {return "FV3JEDI";}
  static std::string nameCovar() {return "FV3JEDIstatic";}

  typedef fv3jedi::GeometryFV3JEDI              Geometry;
  typedef fv3jedi::StateFV3JEDI                 State;
  typedef fv3jedi::IncrementFV3JEDI             Increment;
  typedef fv3jedi::ModelBiasFV3JEDI             ModelAuxControl;
  typedef fv3jedi::ModelBiasIncrementFV3JEDI    ModelAuxIncrement;
  typedef fv3jedi::ModelBiasCovarianceFV3JEDI   ModelAuxCovariance;
  typedef fv3jedi::GetValuesTrajFV3JEDI         InterpolatorTraj;

  typedef ufo::ObsOperator                      ObsOperator;
  typedef ufo::LinearObsOperator                LinearObsOperator;
  typedef ufo::ObsBias                          ObsAuxControl;
  typedef ufo::ObsBiasIncrement                 ObsAuxIncrement;
  typedef ufo::ObsBiasCovariance                ObsAuxCovariance;
  typedef ufo::GeoVaLs                          GeoVaLs;

  typedef ioda::ObsSpace                        ObsSpace;
  typedef ioda::ObsVector                       ObsVector;
  typedef ioda::Locations                       Locations;

};
```

Factory instantiation

Instantiate the change of variable designated by VarChaC2MFV3JEDI. In the YAML we need to call as “Control2Model”

Factory: the class VarChaC2MFV3JEDI is then implemented as normal.

```
void instantiateFV3JEDIVarChangeFactory() {
  static oops::LinearVariableChangeMaker<fv3jedi::FV3JEDITraits,
            oops::LinearVariableChange<fv3jedi::FV3JEDITraits,
                                       fv3jedi::VarChaC2MFV3JEDI> >
            makerVarChangeFV3JEDI_("Control2Model");
}
```

YAML: choose the subclass and the variables to be allocated.

```
- varchange: Control2Model
  inputVariables:
    variables: [ua, va, t, ps, q, qi, ql, o3]
  outputVariables:
    variables: [psi, chi, tv, ps, qc, qic, qlc, o3c]
```

C++/Fortran binding

Binding: C++ side

GeometryFV3JEDI.cc

```
GeometryFV3JEDI::GeometryFV3JEDI(const eckit::Configuration & conf) {
  const eckit::Configuration * configc = &conf;
  stageFv3Files(conf);
  fv3jedi_geo_setup_f90(keyGeom_, &configc);
  removeFv3Files();
}
```

GeometryFV3JEDIFortran.h

```
extern "C" {
  void fv3jedi_geo_setup_f90(F90geom &,
                             const eckit::Configuration * const *);
}
}
```

Binding: Fortran side

fv3jedi_geom_interface_mod.F90

```
subroutine c_fv3jedi_geo_setup(c_key_self, c_conf) bind(c,name='fv3jedi_geo_setup_f90')

implicit none
integer(c_int), intent(inout) :: c_key_self
type(c_ptr), intent(in)       :: c_conf

type(fv3jedi_geom), pointer :: self

call fv3jedi_geom_registry%init()
call fv3jedi_geom_registry%add(c_key_self)
call fv3jedi_geom_registry%get(c_key_self,self)

call create(self,c_conf)

end subroutine c_fv3jedi_geo_setup
```

Access to the object is through a linked list

Integer Key comes in, pointer to an object gets passed.

LinkedList inclusion

At the `interface_mod` level the Linked List is created for the Fortran version of the object.

`linkedList_i.f` contains the list of objects and `linkedList_c.f` contains the methods for manipulating and accessing the current object in the linked list.

```
module fv3jedi_geom_interface_mod

use fv3jedi_geom_mod

implicit none
private
public :: fv3jedi_geom_registry

#define LISTED_TYPE fv3jedi_geom
#include "linkedList_i.f"

type(registry_t) :: fv3jedi_geom_registry

contains

#include "linkedList_c.f"
```

linkedList_i.f

Linked list node is where an object is actually stored in memory. It also contains a pointer to the next element.

```
!> Node of a linked list
type :: node_t
 integer                 :: key
 type(LISTED_TYPE) :: element
 type(node_t), pointer   :: next => NULL()
end type
```

Class containing pointer to the head node. Methods for accessing that object.

```
!> Registry type
type :: registry_t
 logical                :: l_init = .false.
 integer                :: count  = 0
 type(node_t), pointer :: head   => NULL()
 contains
  procedure :: init => init_
  procedure :: finalize => finalize_
  procedure :: add => add_
  procedure :: get => get_
  procedure :: remove => remove_
end type
```

linkedList_c.f: initilaize

If linked list not initialized associate the head node and set the flags.

```
subroutine init_(self)
 class(registry_t) :: self

 !set count to zero and allocate the head of the list
 if(.not.self%l_init.or..not.associated(self%head)) then
  self%count = 0
  allocate(self%head)
  nullify(self%head%next)
  self%l_init=.true.
 endif

end subroutine
```

linkedList_c.f: add

Key comes in from OOPS. Adding an object to the linked list so ‘up the counter’ and set the key.

Then allocate the next element. This is the actual allocation of memory for the object.

Associate a pointer to the next element in the linked list.

```
subroutine add_(self,key)

 class(registry_t) :: self
 integer           :: key

 type(node_t), pointer :: next

 self%count = self%count+1
 key = self%count

 allocate(next)
 next%key = key

 next%next => self%head%next
 self%head%next => next

end subroutine
```

linkedList_c.f: get

Pointer comes in which needs to be associated with the object in the position in the linked list associated with the key.

Do while loop sweeps the linked list until the key matches the point in the linked list.

```
subroutine get_(self,key,ptr)
 class(registry_t)                :: self
 integer                          :: key
 type (LISTED_TYPE), pointer :: ptr

 type(node_t), pointer :: next

 next => self%head
 ptr => NULL()

 do while(associated(next))
  next=>next%next
  if(key.eq.next%key) then
   ptr => next%element
   exit
  endif
 enddo

end subroutine
```

Questions?