OOPS as a common framework for Research and Operations

Yannick Trémolet, Alfred Hofstadler and Willem Deconinck

ECMWF 14th Workshop on Meteorological Operational Systems

18-20 November 2013
Introduction

- Forecasting systems are becoming better but more and more complex:
  - Single analysis and single forecast,
  - Ensemble forecasts,
  - Flow dependent background errors from Ensemble Data Assimilation.

- Transition between Research and Operations is currently based on common SMS/ecFlow framework
  - Research suites are generated by PrepIFS as part of an experiment
  - Research experiments are re-engineered into (e-) suites for Operations
  - Transition is getting more complex and time consuming with increased complexity of suites

- Complexity will keep increasing in the future:
  - Long overlapping 4D-Var windows,
  - Hybrid data assimilation (EDA and DA coupled two-ways),
  - Coupled ocean-atmosphere models...
The OOPS Project

- The complexity of the IFS code is more and more difficult to manage.

- New scientific and technical (scalability) developments require a more flexible data assimilation system.

- We have started re-factoring the IFS into the Object-Oriented Prediction System (OOPS).

- The scripts and suite definitions will be affected:
  - The outer loop of 4D-Var will be moved inside the C++ layer,
  - The Fortran namalists will have to be replaced, at least partially, by more flexible technology (XML, JSON).

- The suite definitions and scripts define the application at the highest level.
  - We should think of them as part of the “system”.
Like the Fortran code, the suite definitions and scripts have become more and more difficult to maintain and develop.

Three levels are mixed together in the suite definitions and scripts:
- The model (IFS, NEMO...), although the top level of OOPS is generic,
- The “scientific” description of the cycling,
- The workflow “technical” specificity (SMS or ecflow).

The three levels could be, and should be, isolated from each other.
Example: Analysis and forecast cycling

```python
dassim = oops4dvar(userConfig)
Bmatrix = mars.retrieve(Bconfig)

for date in daterange(fcycle, lcycle, step):
    obs = mars.retrieve(date, obsConf)
    background = mars.retrieve(fc(date-step, step))

    an = dassim.run(obs, background, Bmatrix)

    fc = forecast.run(an)

    mars.archive(an)
    mars.archive(fc)
```

- The cycling is independent of the model.
The cycling is independent of the model.

\textbf{B} can be flow dependent.
# Initializations not shown...

```python
for date in daterange(fcycle, lcycle, step):
    edate = date-step
    for member in EDA:
        edaobs = perturb(obs)
        edabg = mars.retrieve( edafc[member](edate-step, step) )
        edafc[member] = dacycle.run(edaobs, edabg, Bmatrix, config)

    Bmatrix = Covariance.estimate( edafc )

    obs = mars.retrieve(date, obsConf)
    background = mars.retrieve( fc(date-step, step) )
    dacycle.run(obs, background, Bmatrix, daConfig)
```

- The cycling is independent of the model.
- $B$ can be flow dependent.
- $B$ can be computed on the fly by an EDA system.
Example: Analysis and forecast cycling

```python
dassim = oops4dvar(userConfig)
Bmatrix = mars.retrieve(Bconfig)

for date in daterange(fcycle, lcycle, step):
    obs = mars.retrieve(date, obsConf)
    background = mars.retrieve( fc(date-step, step) )

    an = dassim.run(obs, background, Bmatrix)

    fc = forecast.run(an)

    mars.archive(an)
    mars.archive(fc)
```

- The cycling is independent of the model.

- \( \mathbf{B} \) can be flow dependent.

- \( \mathbf{B} \) can be computed on the fly by an EDA system.

- On its own, the cycling algorithm is relatively easy to describe.
Abstracting the workflow

dassim = oops4dvar(userConfig)
Bmatrix = mars.retrieve(Bconfig)

for date in daterange(fcycle, lcycle, step):
    obs = mars.retrieve(date, obsConf)
    background = mars.retrieve(fc(date-step, step))

    an = dassim.run(obs, background, Bmatrix)

    fc = forecast.run(an)

    mars.archive(an)
mars.archive(fc)

- On its own, the cycling algorithm is relatively easy to describe.
- And there is enough information to generate all the triggers!
- Why are we writing them by hand?
  - We are duplicating information.
  - It is difficult to maintain and modify.
  - The risk of bugs is increased.
Prototype: PyOOPS

- A prototype has been implemented in python to test the approach.
- The system is organised around tasks whose input and outputs are metadata objects.
- The metadata objects are also used by the workflow to generate the triggers.

```python
class ForecastModel(Task):
    def constructor(self):
        self.add_input('init')
        self.add_output('fc')
        self.add_variable('length')
        self.add_variable('steps')

    def execute(self):
        analysis = self.input('init')
        forecast = MetaData(
            type = 'fc',
            date = analysis.valid_time,
            steps = self.variable('steps'),
            window_end = analysis.window_end
        )

        """ code here that configures and executes the model """
        self.set_output('fc', forecast)
```
Tasks are used as building blocks to compose complex structures.

```
class Analysis(CompositeTask):
    def constructor(self):
        self.add_input('window')
        self.add_output('an')

        self.fetch_obs = self.add_task(Retrieve('fetch_obs'))
        self.bgfc = self.add_task(GetBackground('bgfc'))
        self.an4dvar = self.add_task(Analysis4dvar('4dvar'))
        self.archive_bg = self.add_task(Archive('archive_bg'))
        self.archive_fb = self.add_task(Archive('archive_fb'))

    def compose(self):
        window = self.input('window')
        bg = self.bgfc(window=window)
        obs = self.fetchobs(window=window)
        (an, fb) = self.an4dvar(bg=bg, obs=obs, window=window)
        self.archive_bg(data=bg)
        self.archive_fb(data=fb)

        self.set_output('an', an)

...
```
Prototype with QG toy-model and ecFlow

```python
class Analysis(CompositeTask):
    def compose(self):
        window = self.input('window')
        bg = self.bgfc(window=window)
        obs = self.fetchobs(window=window)
        (an, fb) = self.an4dvar(bg=bg, obs=obs, window=window)
        self.archive_bg(data=bg)
        self.archive_fb(data=fb)
        self.set_output('an', an)
```

- Note that GetBackground is a composite task as well!
- The workflow (ecFlow) is abstracted from the suite definition.
  - Should we call it ezFlow?
Abstracting the workflow

- Scientists should think as if writing any algorithm.

- Executing the (python) code generates the suite (and scripts).
  - Each component can generate a single task or a family.
  - The workflow is chosen when running the python program.
  - A simple workflow can run the tasks on the fly (toy system on a laptop).

- The workflow can be specialized for Operations to control when the observations are retrieved and the analysis cycle started.

- Everything else is the same: More can be shared between RD and OD.
Future Framework

OOPS provides a common generator for both Research and Operations suites

**OPERATIONS**
- ecFlowview
- Operations suites

**RESEARCH**
- web interface
- Research suites
- ecFlowview

OOPS engine

OOPS provides a common generator for both Research and Operations suites
The OOPS prototype is working in research mode
- with toy models (Lorenz, QG),
- for (simple) forecast experiments with the IFS.

Next steps:
- port all suites to the new framework (the bulk of the work is in identifying all the inputs and outputs of each task in the current system),
- implement the OD mode.

Potential:
- for RD to express complex algorithms in a sustainable way,
- for OD to implement these algorithm faster and with less risk of errors.