European Organisation for Astronomical Research in the Southern Hemisphere

Programme: VLT

Project/WP: Science Data Quality group

EDPS workflow design tutorial Quick Start

Document Number: ESO-XXX

Document Version: 0.8

Document Type: Manual (MAN)

Released on: 2023-12-01

Document Classification: Public

– DRAFT –

Prepared by: L. Coccato, W. Freudling, S. Zampieri

Validated by:

Approved by: Name

This page was intentionally left blank

Change record

Issue/Rev.	Date	Section/Parag. affected	Reason/Initiation/Documents/Remarks
0.1	20/02/2024	All	First draft release

This page was intentionally left blank

Contents

1 Background

1.1 What is EDPS?

The ESO Data Processing System (EDPS) is a framework to run ESO's data processing pipelines. Each of ESO's data processing pipelines consist of a series of stand alone programs called "recipes". Each recipe is designed to process one type of input data. The processing of these input data typically needs a range of auxiliary files such as calibration files. EDPS is designed to select appropriate input data for the different recipes of a pipeline, and execute them in sequence. This is done by specifying for each pipeline the dependencies of the recipes on each other as well as the information to organise the data. From this description, the data organisation and processing workflows can be derived. The general principles of EDPS have been described by Freudling, Zampieri, Coccato et al. (2024, A&A, 681, A93). In the following, we refer to this description as a workflow. A workflow can the used to process a set of data fully automatically, either in batch or interactive mode.

1.2 Scope

This document describes the fundamental steps to write a basic EDPS workflow. The FORS instrument pipeline is used as example to design a workflow aimed at processing imaging data[1]. A more advanced tutorial with more examples and a complete list of EDPS functionalities is available at `https://www.eso.org/sci/software/edps.html`.

In this document, we assume the user has a working installation of EDPS on its computer. A comprehensive tutorial on how to install and execute EDPS a is available at `https://www.eso.org/sci/software/edps.html`.

1.3 Workflows overview

An EDPS workflow is fully defined by two main elements:

- `data_source`. A data source is a group of files to be processed, and the input to the task. Data sources use classification rules for file classification.
- `task`. A task represents a processing step in the reduction cascade, and it is typically associated to a pipeline recipe. Inputs to tasks are data sources, and other tasks.

A fully operational EDPS workflow capable of running a pipeline, contains other elements, i.e. all the components that allow to classify and group files together for processing, ensuring the correct sequence of recipes with appropriate input/output relations. For sake of simplicity, in this tutorial we adopt the convention to include all the needed elements of a workflow in a single file. However,

[1] For sake of simplicity, we limit the functionalities of this workflow to cover only certain steps of the FORS imaging reduction cascade. Therefore, the workflow is to be consider as a working example for the demo dataset available with this tutorial, and cannot be used to process any other FORS imaging data.

for more complex cases, it is advisable to split the various elements (list of tasks, definition of data sources, classification rules, auxiliary functions) into separate files, entwined by import statements. Therefore, a workflow “package” can consists of several Python files.

2 Setting up your system

We assume the user has already installed EDPS with its configuration saved in the file: `$HOME/.edps/application.properties`. We also assume the FORS instrument pipeline is installed, and the `esorex` command is in the system `$PATH`.

The first step is to configure EDPS so that it points to the directory where we develop our workflow. We recommend to use a Python IDE for developing.

1. Open the configuration file `$HOME/.edps/application.properties`
2. Set the variable `workflow_dir` to a location where you will be developing the workflow, e.g. `/home/user/edps/workflow_src`.
3. Create the workflow file `fors_demo_wkf.py` and save it inside `/home/user/edps/workflow_src/fors/`. At this stage, the file can be empty. The name could be anything, but it must start with the instrument name (i.e., `fors`), and end with `_wkf`. For example, `fors_my_example_wkf.py` could have been also a valid name.

To test if your system is properly set up, do the following:

1. Activate the EDPS environment (see the EDPS quick start tutorial for instructions):

   ```
   . <path_to_environment>/bin/activate
   ```

2. Close the edps server that might be running. Note: any new workflow, or any modification to an existing one, become visible to edps only after a server restart.

   ```
   edps -shutdown
   ```

3. Check if your workflow is visible by typing:

   ```
   edps -lw
   ```

   If everything went well[2], you should see the list of visible workflows in the terminal, e.g.:

   ```
   ['fors.fors_demo_wkf']
   ```

[2] Sometimes it could be necessary to delete the `$HOME/EDPS_data/db.json` database, if the workflow directory has changed.

3 My first workflow

A workflow is fully defined by a data source and a task. In the following example, we define a workflow that processes bias frames. Edit the file `fors_demo_wkf.py` described in Section 2 as follows:

fors_demo_wkf.py

```
from edps import task, data_source

# --- Data sources ---
raw_bias = (data_source("BIAS")
            .build())

# --- Processing tasks ---
bias_task = (task("bias")
             .with_main_input(raw_bias)
             .build())
```

In the above workflow, we have defined a data source, named `"BIAS"` that is the main input of the processing task named `"bias"`.

In order to load the changes in `fors_demo_wkf.py`, the EDPS server must first be shut down:

```
edps -shutdown
```

To see the tasks available in the `fors_demo_wkf` workflow, type:

```
edps -w fors.fors_demo_wkf -lt
```

The output on terminal should be:

```
{
  "all": [
    "bias"
  ]
}
```

To create a png file with the graphic representation of the workflow (see Figure 5.1), type[3]:

```
edps -w fors.fors_demo_wkf -g | dot -Tpng > fors_demo.png
```

3.1 Classifying files

In order to classify files and make them available to the workflow, one has to create a classification rule and attach it to the datasource.

The general syntax for a classification rule is:

[3] See https://graphviz.org for further info on the `dot` command.

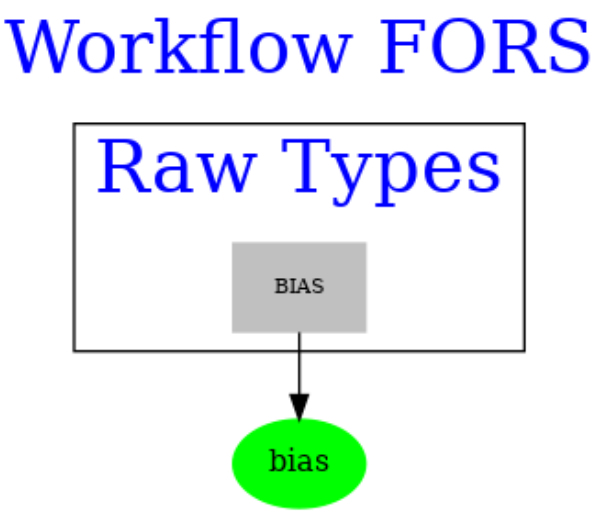

Figure 3.1: The basic `fors_demo_wkf` workflow.

```
class_rule_name = classification_rule("TAG", <dictionary>)}
```

where `"TAG"` is a string denoting the category of the file, as the processing recipe expects to find in the input sof. `<dictionary>` is a Python dictionary listing the header keywords and the values they should have to have a file classified as `"TAG"`. A function can be passed instead of a dictionary, if the classification requires more complicated expression. The rule has to be attached to the data source by adding the method: `.with_classification_rule(class_rule _name)` to the datasource. In our example, the classification rule for FORS raw biases is:

```
bias_class = classification_rule("BIAS",{"dpr.type": "BIAS"})
```

Our workflow now becomes:

```
from edps import data_source, task, classification_rule

#-Classification rukes
bias_class = classification_rule("BIAS",{"dpr.type": "BIAS"})

#-Datasources
raw_bias = (data_source("BIAS")
           .with_classification_rule(bias_class)
           .build())

#- Processing tasks
bias = (task("bias")
      .with_main_input(raw_bias)
      .build())
```

To see the file classification, type:

```
edps -shutdown
edps -w fors.fors_demo_wkf -i <path_to_data> -c
```

The <path_to_data> is the location of the input directory (a space separated list can be provided, directories are read recursively). A list with the file name followed by its classification is prompted on the terminal. At this stage, on;y bias exposures are classified with the tag BIAS. The other files in the input directory are tagged with NONE. The command edps -shutdown is needed if the EDPS server is open: server must be restarted in order to apply any workflow change.

3.2 Running a recipe

In order to run a recipe with EDPS, there are few steps that must be done.

1. **Pipeline.** The FORS instrument pipeline has to be installed in the system. It is recommended to have esorex in the system path; otherwise, EDPS has to point to the esorex executable associated with the pipeline to be executed. This can be achieved by setting the variable esorex_path in the application.properties configuration file to the correct path.
2. **Datasource association with classification rule**. The data source has to be linked to the appropriate classification rule. This can done by adding:

   ```
   .with_classification_rule(bias_class)
   ```

   to the raw_bias datasource.
3. **File grouping.** By default, all the files that fulfill the classification rule will be grouped and eventually processed together. What we want is to identify the groups of files (biases in this case) that are meant to be processed together, i.e. that enter the same recipe input .sof. This steps depends on the calibration plan and the instrument recipe. In our example, it is sufficient to separate the biases by start of the observing template and detector id, by adding:

   ```
   .with_grouping_keywords(["det.chip1.id","tpl.start"])
   ```

   to the raw_bias datasource. A long list of header keywords can be provided, as well as functions to support more complex rules and requirement for the minimum number of files to consider a group complete.
4. **Specifying the recipe.**The recipe has to be linked to the task that runs it, by adding:

   ```
   .with_recipe("fors_bias")
   ```

   to the bias processing task

Our workflow now becomes:

```
from edps import task, data_source, classification_rule

# --- Classification rules ---
bias_class = classification_rule("BIAS", {"dpr.type": "BIAS"})

```

```
6
7  # --- Data sources ---
8
9  raw_bias = (data_source("BIAS")
10             .with_classification_rule(bias_class)
11             .with_grouping_keywords(["det.chip1.id","tpl.start"])
12             .build())
13
14 # --- Processing tasks ---
15 bias_task = (task("bias")
16              .with_recipe("fors_bias")
17              .with_main_input(raw_bias)
18              .build())
```

To execute the workflow and run the recipe type:

```
edps -shutdown
edps -w fors.fors_demo_wkf -i <path_to_data> -t bias
```

The results will be saved in the directory specified in the application.properties configuration file, with the variable `base_dir`, whose default value is `$HOME/EDPS_data/`. Products are organized by `INSTRUMENT/task_name/job_id/`. For more information, consult the EDPS user manual.

3.2.1 Changing recipe parameters

In order to see all the available parameters in the recipe executed in the task bias and their default values, type:

```
edps fors.fors_demo_wkf -p bias
```

To process the biases with different recipe parameters, type:

```
edps -w dummy.dummy_wkf -i <input_directories>
   -rp <TASK> <PARAMETER> <VALUE>
```

Where `TASK` is the task name that runs the recipe (`bias`, in this example) we want to change the parameter for; `PARAMETER` is the parameter name, and VALUE is the value we want to use. The `PARAMETER` name must give the full parameter name, as displayed with the `-p bias` option.

For more information, consult Section 4 of the EDPS tutorial.

4 Adding a task to the workflow: flat fielding

In this Section we will add a new task to the reduction cascade. The task is named `flat_task`. It is designed to process raw flat fields and it is associated to the `fors_img_sky_flat` recipe of the FORS pipeline. The `flat_task` task needs the products of `bias_task` as input.

4.1 Adding the classification rules

We need to add the classification rules to classify the raw flats.

Flat fields are recognized by the header keyword `DPR.TYPE="FLAT,SKY"`, and they need to be flagged as `SKY_FLAT_IMG` in the recipe input sof.

Therefore, the classification rules are:

```
flat_class = classification_rule("SKY_FLAT_IMG", {"dpr.type": "FLAT,SKY"})
```

4.2 Adding the data sources

We define now the data_source for the raw flats data type defined above. The data_source has to be associated with the corresponding classification rules and it must specify how the flats have to be grouped. Flats have to be grouped using `TPL START`, `DET CHIP1 ID` and `INS FILT1 NAME` header keywords.

The raw flat datasource can be defined as:

```
raw_flats = (data_source("FLAT")
   .with_classification_rule(flat_class)
   .with_grouping_keywords(["tpl.start", "det.chip1.id","ins.filt1.name"])
   .build()
```

4.3 Adding the association rules

Rules that specify how a data source has to be associated must be attached to the data source itself. This can be done by specifying a list of header keywords the file must match, or a more complicated function. For our example, biases should simply match the instrument name (header keyword `DET CHIP1 ID`)

Therefore, our goal can be achieved by adding

```
    .with_match_keywords(["det.chip1.id"])
```

to the bias task.

4.4 Adding the processing tasks.

In order to be able to process the flat fields, the corresponding processing task needs to be added to the workflow.

The task to process flat fields. It requires `raw_flat` as man input, and the outcome of the bias tasks as associated inputs; the correct bias will be associated following the association rules specified in the `raw_bias` datasource (see previous Section). It then needs to be connected to the appropriate recipe `fors_img_sky_flat` It looks like:

```
flat_task = (task("flat")
      .with_recipe("fors_img_sky_flat")
      .with_main_input(raw_flats)
      .with_associated_input(bias_task)
      .build())
```

4.5 The workflow with two tasks

The workflow with the bias and flat tasks is shown in Figure 4.1 and looks like :

fors_demo_wkf.py

```
1  from edps import task, data_source, classification_rule
2
3  # --- Classification rules ---
4  # - Raw files
5  bias_class = classification_rule("BIAS", {"dpr.type": "BIAS"})
6  flat_class = classification_rule("SKY_FLAT_IMG", {"dpr.type": "FLAT,SKY"})
7
8
9  # --- Data sources ---
10 # - Raw calibrations data sources
11
12 raw_bias = (data_source("BIAS")
13             .with_classification_rule(bias_class)
14             .with_grouping_keywords(["det.chip1.id","tpl.start"])
15             .with_match_keywords(["det.chip1.id"]).build())
16
17 raw_flats = (data_source("FLAT")
18              .with_classification_rule(flat_class)
19              .with_grouping_keywords(["tpl.start", "det.chip1.id","ins.filt1.name"])
20              .build())
21
22 # --- Processing tasks ---
23 # - Tasks to reduce raw calibrations
24 bias_task = (task("bias")
25              .with_recipe("fors_bias")
26              .with_main_input(raw_bias)
27              .build())
28
29 flat_task = (task("flat")
```

```
30              .with_recipe("fors_img_sky_flat")
31              .with_main_input(raw_flats)
32              .with_associated_input(bias_task)
33              .build())
```

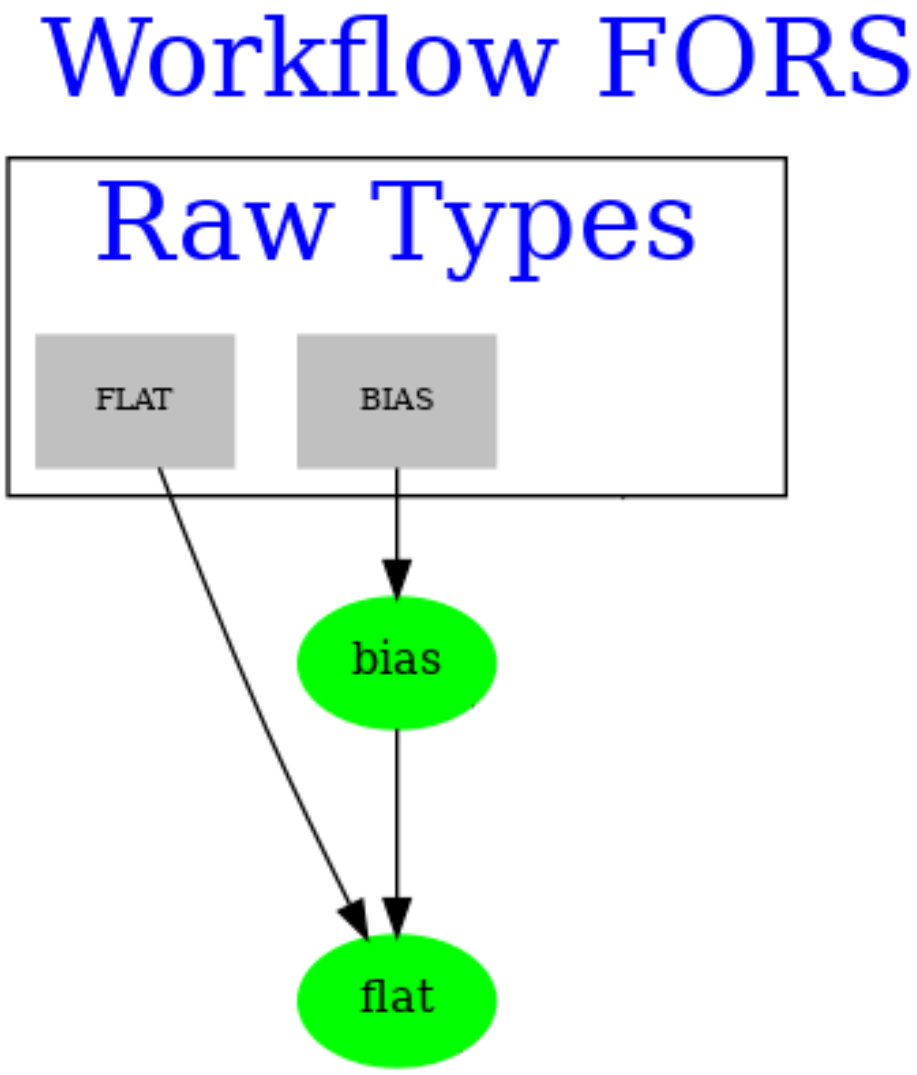

Figure 4.1: The "fors_demo_wkf" workflow containing the bias and flat tasks.

5 Completing the workflow: adding the science task

In this Section we will add the final task to complete the reduction cascade considered in our example. The task is named `science_task`. It is designed to process raw science files, and it is associated to the `fors_img_science` recipe of the FORS pipeline. The task needs the products of `bias_task` and `flat_task` as input. Moreover, it needs static calibrations (detector illumination and photometric tables).

Similarly to what done in Sections 4-4.3, we need to define the classification rules, the data sources, and the association rules.

5.1 Adding the classification rules

We need to add the classification rules to classify the science exposures and the static calibrations needed to process them.

Science fiels are recognized by `dpr.catg="SCIENCE"` and `dpr.tech="IMAGE"`, and they need to be flagged as `SCIENCE_IMG` in the recipe input sof.

The static calibrations are recognised by their `PRO.CATG` header keyword and need to be flagged as such.

Therefore, the classification rules for these data types:

```
science_class = classification_rule("SCIENCE_IMG", {"dpr.tech": "IMAGE",
"dpr.catg": "SCIENCE"})

photometric_tables_class = classification_rule("STATIC_PHOT_COEFF_TABLE",
{"pro.catg": "PHOT_TABLE"})

detector_illumination_class = classification_rule("DETECTOR_ILLUMINATED_REGION",
{"pro.catg": "DETECTOR_ILLUMINATED_REGION"})
```

5.2 Adding the datasources

The datasources for all the data types defined above have to be created. The datasources have to be associated with the corresponding classification rules, and they must contain the criteria with files have to be grouped. Note that science frames must be processed individually.

Our goal can be achieved by adding:

```
    .with_grouping_keyword(["mjd-obs"])
```

to the science data source. Note the use of a unique keyword such as mjd-obs to force the files to be processed individually. If no grouping keyword is provided, all the files of that datasource (e.g. fulfilling that classification rule) are grouped into a single group.

The science datasource is therefore:

```
raw_science = (data_source("OBJECT")
    .with_classification_rule(science_class)
    .with_grouping_keywords(["mjd-obs"])
    .build())
```

The data sources of the static calibrations are:

```
photometric_tables = (data_source("photometric_tables")
    .with_classification_rule(photometric_tables_class)
    .build())

detector_illumination = (data_source("detector_illumination")
    .with_classification_rule(detector_illumination_class)
    .build())
```

The datasources of the static calibration still miss the association rules, that specify how they should be associated to the science (see next Section).

5.3 Adding the association rules

Rules that specify how a data source has to be associated must be attached to the data source itself. This can be done by specifying a list of header keywords the file must match, or a more complicated function. For our example, flat fields must match the filter name and detector ID (header keyword: `INS.FILT1.NAME` and `DET.CHIP1.ID`), and static calibrations should simply match the `INSTRUME` keyword.

Therefore, our goal can be achieved by adding

```
    .with_match_keywords(["instrume"])
```

to the datasources of the static calibrations, and

```
    .with_match_keywords(["det.chip1.id","ins.filt1.name"])
```

to the datasources of of raw flats.

5.4 Adding the processing tasks.

One final task has to be added to the workflow to complete the reduction chain: the task to process the science exposures.

The last task process the science exposures. It requires as main input the `raw_science` and as associate inputs the outcome of the `flat_task` and `bias_task`, and some static calibrations. The correct files will be associated following the association rules specified in the various data sources. Note that the association rules defined in the raw bias data_source are valid both for associating biases to the flats, and for associating biases to the science.

It looks like:

```
science_task = (task("science")
      .with_recipe("fors_img_science")
      .with_main_input(raw_science)
      .with_associated_input(bias_task)
      .with_associated_input(flat_task)
      .with_associated_input(detector_illumination)
      .with_associated_input(photometric_tables)
      .build())
```

5.5 Running the bias and flat workflow on real data. T.B.D.

5.6 The complete workflow

The full workflow looks like:

fors_demo_wkf.py

```
from edps import task, data_source, classification_rule

# --- Classification rules ---
# - Raw files
bias_class = classification_rule("BIAS", {"dpr.type": "BIAS"})
flat_class = classification_rule("SKY_FLAT_IMG", {"dpr.type": "FLAT,SKY"})
science_class = classification_rule("SCIENCE_IMG", {"dpr.tech": "IMAGE",
                      "dpr.catg": "SCIENCE"})

# - Static calibrations
photometric_tables_class = classification_rule("STATIC_PHOT_COEFF_TABLE",
            {"pro.catg": "PHOT_TABLE"})
detector_illumination_class = classification_rule("DETECTOR_ILLUMINATED_REGION",
                  {"pro.catg": "DETECTOR_ILLUMINATED_REGION"})

# --- Data sources ---
# - Raw calibrations data sources

raw_bias = (data_source("BIAS")
             .with_classification_rule(bias_class)
             .with_grouping_keywords(["det.chip1.id","tpl.start"])
             .with_match_keywords(["det.chip1.id"]).build())

raw_flats = (data_source("FLAT")
              .with_classification_rule(flat_class)
```

```
            .with_grouping_keywords(["tpl.start", "det.chip1.id","ins.filt1.name"])
            .with_match_keywords(["det.chip1.id","ins.filt1.name"])
            .build())

# - Data sources for static calibrations
photometric_tables = (data_source("photometric_tables")
                      .with_classification_rule(photometric_tables_class)
                      .with_match_keywords(["instrume"])
                      .build())

detector_illumination = (data_source("detector_illumination")
                         .with_classification_rule(detector_illumination_class)
                         .with_match_keywords(["instrume"])
                         .build())


# - Science data source
raw_science = (data_source("OBJECT")
               .with_classification_rule(science_class)
               .with_grouping_keywords(["mjd-obs"])
               .build())


# --- Processing tasks ---
# - Tasks to reduce raw calibrations
bias_task = (task("bias")
             .with_recipe("fors_bias")
             .with_main_input(raw_bias)
             .build())

flat_task = (task("flat")
             .with_recipe("fors_img_sky_flat")
             .with_main_input(raw_flats)
             .with_associated_input(bias_task)
             .build())

# - Task to reduce the scientific exposures
science_task = (task("science")
                .with_recipe("fors_img_science")
                .with_main_input(raw_science)
                .with_associated_input(bias_task)
                .with_associated_input(flat_task)
                .with_associated_input(detector_illumination)
                .with_associated_input(photometric_tables)
                .build())
```

Other features that allows high flexibility in the workflow design are available, such as

- the possibility to specify conditions to associations or tasks executions;
- the possibilities to associate a data source following different association rules, depending on which file needs them;
- the possibility to modify the task dynamically depending on the properties of the main inputs;

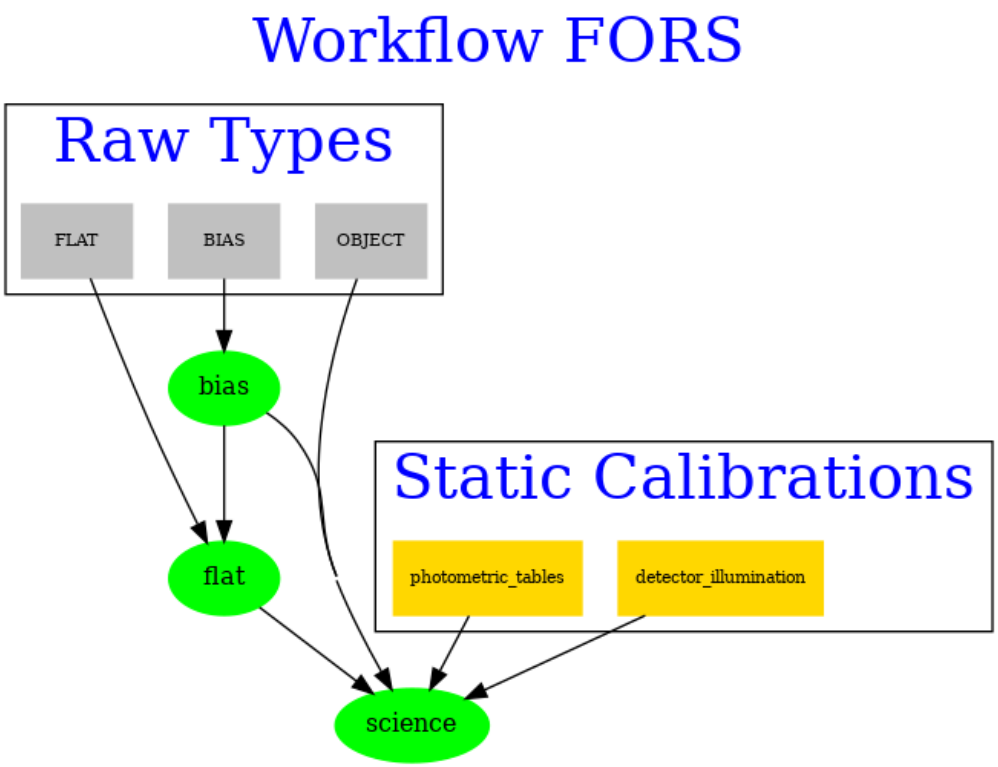

Figure 5.1: The complete "fors_demo_wkf" workflow.

- auxiliary functions,
- subworkflows;
- functions for classification and association;
- grouping and clustering rules;
- minimum number of associations.

and much more. Please consult the full EDPS workflow design manual for further information.

6 Running the complete workflow on real data. T.B.D.