European Organisation for Astronomical Research in the Southern Hemisphere

Programme: VLT

Project/WP: Science Data Quality group

EDPS workflow design tutorial

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

ESO – Karl-Schwarzschild-Str. 2 – 85748 Garching bei München – Germany
www.eso.org

This page was intentionally left blank

Change record

Issue/Rev.	Date	Section/Parag. affected	Reason/Initiation/Documents/Remarks
0.8	20/12/2024	All	First draft release

This page was intentionally left blank

Contents

I Introduction 9

1 Background 10

1.1 What is EDPS? 10

1.2 Scope 10

2 Installation and execution 11

3 Workflows 12

3.1 Overview 12

3.2 Example of a basic workflow 13

II Main components of a workflow 15

4 Tasks 17

4.1 Optional inputs 17

4.2 Targets and Metatargets 18

4.3 Filtering inputs and outputs 18

4.3.1 Filtering the inputs of a task 18

4.3.2 Filtering the outputs of a task 19

4.4 Convention on the order of task methods 20

5 Data Sources 21

5.1 Grouping and clustering of files 22

5.2 Minimum number of files in a group 23

5.3 How do I choose the name for a data source? 23

5.4 How do I write the data source file? 24

5.4.1 Import statements 24

5.4.2 Convention on its content 24

6 Classifications and associations **25**

6.1 Classification Rule 25

6.2 How to associate calibrations 26

6.2.1 Multiple association level and validity ranges 26

6.2.2 Raw calibrations or master calibrations? 27

6.2.3 Definition of validity ranges for associations 28

6.3 The rules file: complex classifications and associations 28

7 The parameters file **30**

7.1 "General" and "workflow" parameter files 30

7.2 Example 30

III Advanced features **35**

8 Tasks advanced features **37**

8.1 Conditional association 37

8.1.1 Conditional association based on static parameter 37

8.1.2 Conditional association based on dynamic parameter 38

8.2 Execute a task under certain conditions 38

8.3 How to specify alternative inputs 39

8.4 Setting the recipe parameters 39

8.5 Modifying the job properties 40

8.6 Modifying the input category tag 41

9 Running python scripts, looping on recipes **42**

10 Associations: advanced features **45**

10.1 Tasks requiring the same datasources but with different association rules 45

10.2 Optional and mandatory products of a task 45

11 Subworkflows **47**

11.1 General concepts 47

11.2 Example 47

IV Workflow examples 49

12 A complete simple workflow 51

13 Compacting the workflow: merged tasks and merged classification rules. 57

13.1 Pipeline description . . . 57

13.2 First workflow. . . . 59

13.3 Simplified workflow. . . . 63

V How to run and debug workflows 69

14 Tips for debugging. 71

15 EDPS integrated test suite: `json` tests. 72

15.1 Test suite JSON file . . . 72

15.2 Scenario definition . . . 72

15.2.1 Test-case metadata . . . 72

15.2.2 Input file definitions . . . 73

15.2.3 Result expectations . . . 74

15.2.4 Full example . . . 74

15.3 Default behaviour . . . 76

15.3.1 MJD-OBS . . . 76

15.3.2 TPL.START . . . 76

15.3.3 Default keywords . . . 77

15.4 Known limitations . . . 77

15.5 To ease Json implementation and verification . . . 78

Part I

Introduction

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 provides instructions and examples on how to write EDPS workflows for ESO pipelines. It is not intended for users who want to reduce data with the standard workflow for a supported pipeline. Supported pipelines are distributed with fully functional workflows, and a tutorial for using them is available at `https://www.eso.org/sci/software/edps.html`

The current document is mainly for developers of pipelines and for users who want to modify existing EDPS workflows. It assumes that the reader is familiar with the basic concepts behind EDPS workflows. EDPS workflows are written in Python. The EDPS libraries must be used to write a functional workflow. These libraries leave the developer with significant choices for the design of workflows. The official workflows are written using the same conventions for all instrument pipelines. These conventions are described in this document.

2 Installation and execution

A comprehensive tutorial on how to install and execute EDPS a is available at https://www.eso.org/sci/software/edps.html

The default installation is designed to detect and execute workflows that come with instrument pipelines. From a workflow development point of view it is recommended to edit the EDPS configuration file `application.properties` that is located in the `$HOME/.edps` directory, so that any workflow under development (also those not associated to a pipeline) can be seen by EDPS. The fields to modify are:

- `workflow_dir` This should point to a directory containing all the workflow files. The directory should be structured so that it contains sub-directories named after the instrument, that contain the corresponding workflows. For example, a directory structure

  ```
  /home/user/edps_workflow/
              instr1/
              instr2/
  ```

  shall contain the workflows `inst1_wkf.py`, `inst2_wkf.py` in the corresponding directories (along with all the other relevant workflow files, see Section 3.1).

  The command[1]

  `edps -lw`

  should prompt the following list:

  `["instr1.instr1_wkf","instr2.instr2_wkf"]`

- `esorex` This should point to the esorex installation that is linked to the pipeline one has to prepare the workflow for.

[1]The EDPS environment should be activated from the terminal where the command is launched, see the EDPS tutorial for detailed instructions.

3 Workflows

3.1 Overview

An EDPS workflow contains all the components that allow to classify and group files together for processing, ensuring the correct sequence of recipes with appropriate input/output relations. We adopt the convention to store these different components into separate files, entwined by import statements. Therefore, a workflow “package” consists of several Python files:

- The list of tasks (a.k.a. main workflow). The convention for the main workflow file name is `instrument_wkf.py` (e.g., `muse_wkf.py`), where the suffix `_wkf` is mandatory. An instrument can have more than one workflows, their names must star with the instrument name and end with the `_wkf` suffix (e.g., fors_wkf.py, fors_spec_wkf.py, fors_imaging_wkf.py and so forth).
- The file with the datasources, that are the inputs of the various tasks. The convention for its name is `instrument_datasources.py` (e.g., `muse_datasources.py`). The list of datasources and the list of tasks entirely define the workflow. See Section 5.
- The file with the classification statements, that are used to define a datasource. It contains the `classification_rules` objects. The files are classified by these classification statements, which obey to some rules. The rules are either defined explicitly within the `classification_rules` object (for simple rules), or defined in the rule file (for more complex rules). The convention for the name of the classification file is `instrument_classification.py`. See Section 6.
- The file with the rules, functions that allows the classification and association of files. The convention for its name is `instrument_rules.py` (e.g., `muse_rules.py`). See Section 6.3.
- A file with the definition of header keywords. Classification and Association rules use header keywords to classify and associate files. Typically, rules uses strings or list of strings containing header keywords. It is convenient to define variables equal to those strings. In this way, the variable is defined once and it is easier to spot syntax issues when using Python development environments (e.g. Pycharm). The convention for the name of such file is `instrument_keywords.py` (e.g., `muse_keywords.py`). See Section 5.1.
- The file with auxiliary functions that are required by tasks (job-editing functions, functions for association conditions, dynamic parameters), if any. The convention for its name is `instrument_task_functions` (e.g., `kmos_task_functions.py`). See Section 8.
- A file with workflow and task parameters. This file is in .yaml format. The convention for its name is `instrument_parameters.yaml` (e.g., `muse_parameters.yaml`). See Section 8.
- The file(s) with subworkflows (if any). See Section 11.

3.2 Example of a basic workflow

A workflow is entirely defined by its data sources (i.e., inputs) and tasks (i.e., processing steps). The following is an example of a basic workflow, composed only by the tasks and data sources which are specified in two separate files.

demo0_datasources.py

```
from edps import data_source

# --- Raw types datasources -------------------------------------------------------------
raw_bias = (data_source('BIAS')
            .build())

raw_flat = (data_source('FLAT')
            .build())

raw_science = (data_source('OBJECT')
               .build())

raw_sky = (data_source('SKY')
           .build())

# Catalogue of standard stars
static_catalog = (data_source("catalog")
                  .build())
```

demo0_wkf.py

```
from edps import task
from .demo0_datasources import *

#--- Processing tasks --------------------------------------------------------------------

#- Task for processing raw biases
bias_task = (task('bias')
             .with_main_input(raw_bias)
             .build())

#- Task for processing raw flats
flat_task = (task('flat')
             .with_main_input(raw_flat)
             .with_associated_input(bias_task)
             .build())

#- Task for processing science exposures
science_task = (task('object')
                .with_main_input(raw_science)
                .with_associated_input(raw_sky, min_ret=0) # sky is an optional input
                .with_associated_input(bias_task)
                .with_associated_input(flat_task)
                .with_associated_input(static_catalog)
                .build())
```

So far, the workflow contains only the link between the processing steps and the inputs. It does not

contain any instruction on how classify the files, nor how to associate them, nor which file belong to which data source, nor the recipe to be executed. It is however, a self-contained entity that defines the data reduction flow shown in Figure 3.1. To obtain the graphical representation of a workflow (e.g., demo0), type from a terminal where the EDPS environment is active:

```
edps -w demo.demo0_wkf -g | dot -Tpng > demo0.png
```

EDPS has other options for visualising a workflow which are described in detail in the EDPS tutorial.

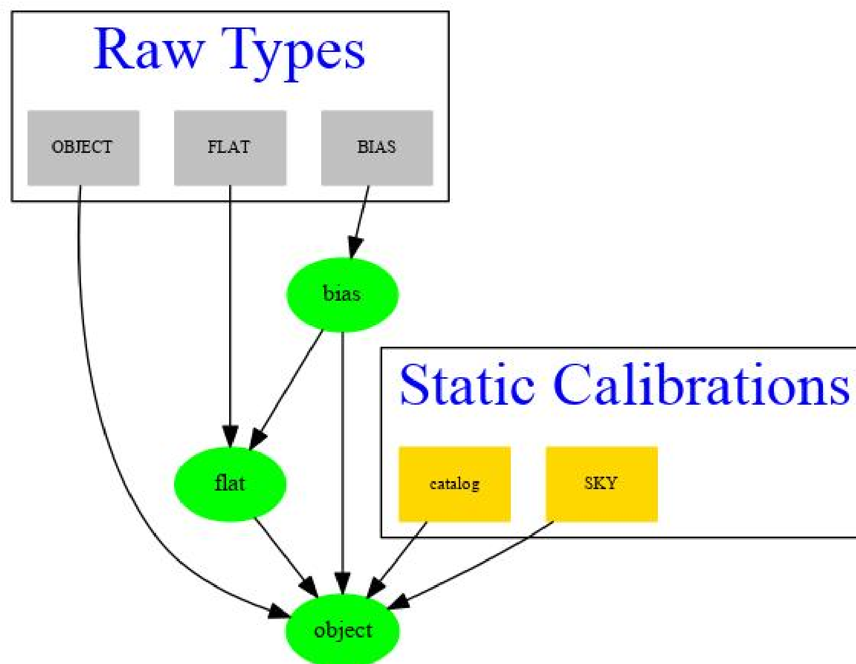

Figure 3.1: The basic "demo0_wkf" workflow.

Part II

Main components of a workflow

4 Tasks

Tasks are the processing units of the reduction process. Each task, if complete, generates a number of jobs, each of them running the same pipeline recipe (or Python function) on different groups of data. To be defined, a task needs just the main input. Other things such as associated inputs (e.g. calibrations), recipe to be executed, alternative inputs, conditions, can be added depending on the data reduction requirements.

The main input is passed with the method `.with_main_input()`. A tasks accepts only one main input, that can be either a datasource or another task.

Inputs other than the main input are passed with the method `.with_associated_input()`. Several associated inputs can be specified. An associated input can be either a datasource or a task.

An example of a task is:

```
1  science_task = (task("science")
2             .with_recipe("run_science")
3             .with_main_input(raw_science)
4             .with_associated_input(bias_task,[MASTERBIAS_class])
5             .build())
```

In the example above, the task gets its main inputs from the `raw_science` data source, and attaches bias calibrations (either the result of the `bias` task that processed raw bias calibrations, or master calibrations already present on disk). The task creates as many jobs as group of files to be processed and reduces them with the 'run_science' recipe.

Calibrations (i.e., bias frames in this example) are associated to the raw_science following association rules attached to the main input of the bias task. See Section 6.2.

The task above generates as many jobs as science files. Incomplete jobs (e.g., science files that are missing the corresponding bias frames) are marked as incomplete and not executed.

4.1 Optional inputs

One can specify the minimum and maximum number of associated input of a certain type, using `min_ret` (default = 1) and `max_ret` (default = 1). Optional inputs of the task are specified by setting `min_ret=0`, e.g:

```
1  science_task = (task("object")
2            .with_main_input(raw_science)
3            .with_associated_input(bias_task, [MASTERBIAS_class], min_ret=0)
4            ....
```

The `min_ret` and `max_ret` specified in the task refers to the associated input: number of jobs to associate or number of master calibrations to associate, not to the number of raw files used in the associated job. These are indeed specified in the data source.

`min_ret` and `max_ret` apply to all master calibrations and jobs specified in the associated input. For example:

```
1   .with_associated_input(bias_task, [MASTERBIAS, BADPIXEL_MASK], min_ret=2)
```

asks for at least 2 bias jobs, or for at least 2 MASTERBIAS and 2 BADPIXEL_MASK frames. It is not possible to specify a different min_ret/max_ret for jobs and master calibrations.

Note: Associated inputs can be associated on the basis of a condition (a.k.a. conditional associations), that might depend on the properties of the data themselves or some parameters defined in the workflow (see Section 8.1).

4.2 Targets and Metatargets

One of the core features of EDPS is the concept of processing target(s), which can be used to restrict the processing to a subset of the workflow tasks. In this context "target" is synonym for "task" and can be referred to using the task name defined in the workflow. It is also possible to attach one or more "metatargets" to a task. A metatarget is a label that can be used to refer to a group of related tasks (for example all calibration tasks or all science tasks) instead of specifying them individually. Currently, EDPS has the following pre-defined metatargets:

- qc1_calib. Tasks used to create master calibration or instrument monitoring.
- qc0. Tasks that are meant to be run a quick look at the telescope (QC0 process).
- science. Tasks responsible for scientific reduction.
- calchecker. Tasks that require monitoring of the needed calibrations (typically instrument monitoring, processing scientific and standard star exposures).

Example:

```
1     bias = (task("bias")
2            .with_recipe("run_bias")
3            .with_main_input(raw_bias)
4            .with_meta_targets([qc1_calib, qc0])
5            .build())
```

Note: Despite the metarget is not mandatory to define a workflow, it is a mandatory element for the QCFlow and CalChecker applications.

4.3 Filtering inputs and outputs

4.3.1 Filtering the inputs of a task

By default all products of a associated task are passed to the recipe of the task.

It is possible to filter the inputs of a task using the method `.with_input_filter()`. The filter accepts a list of classification rules. The list can be a "white list" (only classification rules specified in the list are written into the recipe input set of files) or a "black list" (all inputs are written to the set of files, but those specified in the filter list). These modes can be specified by adding `mode=FilterMode.REJECT` and `mode=FilterMode.SELECT"` to the statement.

Note: the filter is applied at the moment of writing the recipe input set of files. The associations are still displayed in the job and they still determine whether a job is considered complete or not.

Example:

```
1  bias_task    = (task("bias")
2                  .with_recipe("run_bias")
3                  .with_main_input(raw_bias)
4                  .build())
5
6  science_task = (task("object")
7                  .with_recipe("run_science")
8                  .with_main_input(raw_science)
9                  .with_associated_input(bias_task, [MASTERBIAS_class])
10                 .with_input_filter(MASTERBIAS_class, mode=FilterMode.SELECT)
11                 .build())
```

In the example above, the task bias is associated to the task science. The task bias triggers the reduction of `raw_bias` frames with the recipe `run_bias`. All the products of the bias recipe are passed to the science task, but only those listed in `.with_input_filter` are written into the recipe input set of files. In the above example, assuming the bias recipe creates a masterbias and badpixelmask, the masterbias is passed while badpixelmask is not.

4.3.2 Filtering the outputs of a task

Similarly to the input filter, one can filter the outputs of a task. The method is called `.with_output_filter()` and accepts as input a list of classification rules. If one sets `mode=FilterMode.SELECT` (default) then, the listed products are passed to all the tasks that have this task as input. If one sets `mode=FilterMode.REJECT`, then the listed products are never passed to subsequent tasks (but they are saved on disk).

The filter works by checking the category of the products, as provided by the recipe in the metadata output file, and compares it with the category in the classification rule provided in the filter. It does not access the file on disk or read any header keywords.

Example:

```
1   bias     = (task("bias")
2              .with_recipe("run_bias")
3              .with_main_input(raw_bias)
4              .with_output_filter(OVERSCAN_class, mode=FilterMode.REJECT)
5              .build())
6
7   flat     = (task("flat")
8              .with_recipe("run_flat")
```

```
9               .with_main_input(raw_flats)
10              .with_associated_input(bias, [MASTERBIAS_class])
11              .build())
12
13   science = (task("object")
14              .with_recipe("run_science")
15              .with_main_input(raw_science)
16              .with_associated_input(bias, [MASTERBIAS_class, BPMASK_class])
17              .build())
```

In the example above, all the products of the task bias are passed to the tasks flat and object, except the one defined by the classification rule `OVERSCAN_class`.

Important note: if the association preference is set to associate master calibrations rather than tasks (see Section 6.2.2), then the task flats will get only the MASTERBIAS, the task science will get MASTERBIAS and BPMASKS calibrations.

4.4 Convention on the order of task methods

The adopted convention is to define the methods in the task with the following order:

- recipe
- call for "ADARI" reports
- main input
- associated inputs (follow the calibration cascade)
- execution condition
- dynamic parameters
- job functions
- input and output filters
- mapping categories
- meta targets

5 Data Sources

Data sources are inputs of a task. They specify:

- the files that have to be grouped together on the basis of some classification rules.
- the criteria which they have to be associated to the main input of a task.

An example of datasource is:

```
raw_science = (data_source("OBJECT")
               .with_classification_rule(science_extended_class)
               .with_classification_rule(science_pointlike_class)
               .with_grouping_keywords(["tpl.start","obs.targname"])
               .with_match_keywords(['instrume','ins.filt.name'])
               .with_match_function(rules.associate_science)
               .build())
```

The data source raw_science encompasses two different types of files: those that obey to the science_extended_class classification rule, and those that obey the science_pointlike_class classification rule. Files fulfilling at least one of the classification rules will be considered and grouped according to the list of header keywords specified by .with_grouping_keywords().

.with_match_keywords and .with_match_function specify the association rules that of this data source must obey to be associated to tasks (either a list of keywords to match or a more complicated function). See Section 6.2.

Datasources that are main inputs to a task must also have `.with_setup_keywords()` method declared. These list of keywords will be used by qcflow for the scoring.

The order of the datasources defined in the dasource file should follow the calibration cascade. Datasources based on static calibrations should be done at the end.

In the example above, we directly declared the header keywords to be used as strings (e.g., 'tpl.start', and 'obs.targname'). However, it is convenient to define these keywords in a appropriate file and use variables instead. The file has to be imported, i.e. via:

```
from . import instrume_keywords as kwd
```

Convention: It might be convenient to create lists of keywords that are used by many data sources. The list should include keywords of similar type and should be named in a way that "suggests" the type of keywords there are in. For example, one can put in the same list they keywords that refer to the detector (if they are used together, obviously) and in another list the keywords that refer to the spectrograph. For example:

```
    detector=[kwd.instrume, kwd.det_binx, kwd.det_biny, kwd.readoutnoise]
    spectr   = [kwd.opti_ins1, kwd.ipti_slit1_width]
```

and the match can be done via:

```
    .with_match_keywords(detector+spectr)
```

`kwd.instrume` and the other keywords are defined in the keywords file. It is up to the developer to come up with a strategy for naming and which keywords to include in a list; every instruments might require a different solution.

5.1 Grouping and clustering of files

A data source can represent different groups of files. The files that fulfill at least one of the classification rules are grouped together following the list of keywords in the method .with_grouping_keywords(<list>). The list of keywords applies to both classification rules, files from two different classification rules can be grouped together.

To process files individually, one has to use a unique grouping keyword (e.g. `arcfile` or `mjd.obs`.

To process the files all together do not specify grouping keywords.

If a keyword starts with the symbol $, it means that it has to be read from a workflow parameter. This is useful, for example, if the files can be reduced following different data reduction strategies (e.g. grouping files of the same night, or files from the same `tpl.start`, or individually).

Together with the grouping method, there is also the .with_cluster method(), that allows to cluster files together by closeness of a parameter. One can specify the minimum and maximum threshold of the cluster method. This method is particularly useful to collect exposures of a specific time range (e.g., a night, regardless of their tpl.start) or group files on sky coordinates.

Grouping and clustering can be both specified in the data source. Their order reflects the order over which files grouped or clustered. Examples:

To group data by `tpl.start` and `ins.filt.name`:

```
1    science=(data_souce('OBJECT')
2             .with_classification_rule(science_class)
3             .with_grouping_keywords([kwd.instrume,kwd.ins_filt_name])
4             .build())
```

To group data by `tpl.start` and `ins.filt.name`, and cluster them by position on the sky:

```
1    science=(data_souce('OBJECT')
2             .with_classification_rule(science_class)
3             .with_grouping_keywords([kwd.tpl_start,kwd.ins_filt_name])
4             .with_cluster('SKY.POSITION',0.001, 0.05)
5             .build())
```

A file belong to a cluster if the minimum distance from the elements of the cluster is below 0.001 degrees (threshold) and if the maximum from the elements in the cluster is below 0.05 degrees (maximum cluster size)

Note: in the example above, we used variables instead of explicitly stating the the header keywords as strings. These variables are defined in the instrument_keywords.py file this way:

```
1    tpl_start="tpl.start"
2    ins_filt_name="ins.filt.name"
3    instrume="instrume"
```

that is imported with the following statement at the beginning of the datasource file:

```
import instrument_keywords as kwd
```

5.2 Minimum number of files in a group

The method `.with_min_group_size()` specifies the minimum number of files that has to be present in a group to be considered by the workflow. Example:

```
raw_bias = (data_source("BIAS")
    .with_classification_rule(bias_class)
    .with_group_keywords(["tpl.start"])
    .with_min_group_size(3)
    [...]
    .build())
```

Few important notes:

- The method `.with_min_group_size()` has effect only for datasources that are main inputs of tasks. If the datasource is an associated input to a task, then use the keywords `min_ret` and `max_ret` in the `.with_associated_input()` method in the task itself. `min_ret`'s and `max_ret`'s default value is 1.
- there is not a `.with_max_group_size()` equivalent method. By default, all data fulfilling the grouping conditions are included in the group. To process files individually, use unique header keywords such as "mjd.obs" or "arcfile".

5.3 How do I choose the name for a data source?

Datasources have two types of names: a "variable name" used within the workflow and a "label name" used in the graphic representation and instrument monitor. If the "label name" is not provided, EDPS automaically uses the label defined by the classification rule attached to the data source. Because a datasource can have more than one classification rule, it is recommended to assign a "label name" if multiple classification rules are used.

Example:

```
variable_name = (data_source("LABEL_NAME")
            .with_classification_rule(rule1)
            .with_classification_rule(rule2)
            [...]
            .build())
```

It is recommended to use consistent "variable" and "label" names, lower case for variables, upper case for labels.

In the case of data sources coming from raw calibration files, the convention is to start the variable name with the prefix "raw_" (e.g., `raw_bias`).

Names of datasources should be chosen to reach a compromise between:

- understanding what the file is about
- match of the DPR.TYPE keyword that defines the file
- match of the TAG expected by the recipe aimed at processing it.

It is advisable to follow the naming convention of the pipeline `DRS.TYPE` as close as possible, and at the same time, to use common sense to find a balance with a self-explanatory name. For example, the datasource that corresponds to data with `DPR.TYPE = LMP_FMT_CHECK` can be simply `format_check`.

It is also recommended to use the same label name for data sources of different workflows that refer to the same type of files (for example, use consistently names such as "BIAS", "SKYFLAT", "LAMP_FLAT", "ARC", "STANDARD_STAR", "OBJECT" across different workflows). In this way, the instrument monitor will display consistent nomenclature across instruments.

5.4 How do I write the data source file?

5.4.1 Import statements

Create a file named instrument_datasources.py (e.g., muse_datasources.py), with the following import statements:

```
1 from edps import data_source, match_rules
2 from edps.generator.time_range import *
3 from .muse_classification import *
4 from . import muse_keywords as kwd
```

Files in the workflow package that need the data source file must have the following import statement:

```
1     from .muse_datasource import *
```

Note: if one feels uncomfortable in using "*" for importing, an alias can be used (as done, for example, for the keyword and rules files).

Each data source should have a comment explaining what it is for. Commented symbols should separate data sources from raw data, from static calibrations, and of other types (e.g. user-provided calibrations).

5.4.2 Convention on its content

The datasource.py file should contain the datasources and the alternative matching rules (if applicable) that will override those embedded in the data source itself. The alternative matching rule(s) should be written right after the data source it refers to.

Examples are given in Section 10.1

6 Classifications and associations

6.1 Classification Rule

Classification rules are the backbone of the file classification process. Data sources contain files that satisfy one or more classification rule. The default classification rule contains two elements:

- A string with the the tag the recipe processing the file expects.
- A dictionary listing the keywords and their values the file must have to fulfill that rule; or the name of a function that contains the conditions the file must fulfill.[2] Dictionaries are expected for "simple" classification rules, where the keywords must simply match some values. Functions are needed for more complicated relations.

Example of "simple" classification rules (taken from the CRIRES workflow).

```
1  from edps import classification_rule
2  from . import matisse_keywords as kwd
3
4  #Dictionaries containing the values of header keywords that define
5  #calibrations and science data
6  matisse = {kwd.instrume: "MATISSE"}
7  calib_kwd = {**matisse, kwd.dpr_catg: "CALIB"}
8  calib_spectrum_kwd = {**calib_kwd, kwd.dpr_tech: "SPECTRUM", }
9  flat_off_class = classification_rule("OBSDARK",  {**calib_spectrum_kwd,
10                  kwd.dpr_type: ["DARK", "FLAT,OFF"]})
11 flat_on_class = classification_rule("OBSFLAT", {**calib_spectrum_kwd,
12                 kwd.dpr_tech: "INTERFEROMETRY",
13                 kwd.dpr_type: ["FLAT", "FLAT,BLACKBODY"]})
```

where `instrume`, `dpr_type`, `dpr_tech`, and `dpr_catg` are defined in the keywords file. It is recommended to use nested dictionaries and keep the condition of DPR.TYPE at the end.

Example of "complex" classification rule

```
1  from edps import classification_rule
2  from . import crires_rules as rules
3 CAL_WAVE_GAS_CELL = classification_rule('CAL_WAVE_TW', rules.is_gas_cell)
```

where `is_gas_cell` is a function expressing the conditions the file must fulfil to be classified as `"CAL_WAVE_TW"`. It will be described in Section 6.3.

Note: Other two functions `ClassificationRule`, and `ProductClassificationRule` are available, but their use-cases have been incorporated in the more generic `classification_rule` function.

[2] If this second item is not provided, then the tag is expected to be also the PRO.CATG of the file itself.

6.2 How to associate calibrations

Data sources or tasks are associated to a task by the so-called association rules. In the case of data sources, the association rules are defined in the data source In the case of tasks, the task inherit the association rule of the first main input up the reduction cascade, task by task, until the first data source is reached.

The rules attached to the data source can be overridden by rules attached to the .with_associated_input() method in the task (see Section 10.1).

Association can be done in two ways:

- Using the `.with_match_keywords()` method. Here, one provides a list of keywords that the files must match ("simple" association).
- Using the `.with_match_function()` method. Here, one provides a function the files must obey ("complex" association).

Below, we show an example of "simple" association with the `.with_match_keywords()` method.

```
from . import instrument_keywords as kwd

    raw_standard = (data_source('RAW_STANDARD')
                .with_classification_rule(standard_class)
                .with_group_keywords([kwd.tpl_start])
                .with_match_keywords([kwd.det_id])
                .build())

    standard  = (task('STANDARD')
                .with_recipe('run_standard')
                .with_main_input(raw_standard)
                .build())

    science  = (task('SCIENCE')
                .with_recipe('run_science')
                .with_main_input(raw_science)
                .with_associated_input(standard, ['RESPONSE'])

                .build())
```

Where `kwd.tpl_start` and `kwd.det_id` are variables imported from the keyword file instrument_keywords.py ; their values are `"tpl.start"` and `"det.id"`, respectively.

In the case the association is done by other operations than a pure match, then the method `.with_match_function()` ("complex" associations) has to be used (see Section 6.3).

6.2.1 Multiple association level and validity ranges

Associations can have multiple quality levels, depending on several factors, the most common of which is the time range during which a calibration ensures a product of a certain quality.

Therefore, data sources can have several `.with_match_keywords` and `.with_match_functions` statements.

The convention is to assign to each statement a different level (number) with the following meaning:

```
# Convention for Data sources Association rule levels:
# Each data source can have several match function which correspond to
# different  quality levels for the selected data.
# The level is specified as a number that follows this convention:
#   level < 0: more restrictive than the calibration plan
#   level = 0: follows the calibration plan
#   level = 1: quality sufficient for science reduction
#   level = 2: probably still acceptable quality
#   level = 3: significant risk of bad quality results
```

Therefore, when writing the matching rules, it is fundamental to set it to the appropriate quality level, so that one knows what is the expected quality of the products. It is not mandatory to define a match rule for each level. Different levels can have different rules and/or different validity ranges.

6.2.2 Raw calibrations or master calibrations?

The order by which the calibration is searched and its type (raw or master) depends on the order of the matching rules within the data source and on the preference expressed in the parameter `association_preference` of the `application.properties` configuration file.

For example, let's consider the following example of data source:

```
    match_kwd=['instrume','ins.mode']
    calibration = (data_source("CALIBRATION")
         .with_classification_rule(calibration_class)
         .with_match_keywords(match_kwd,time_range=SAME_NIGHT,level=-1)
         .with_match_keywords(match_kwd,time_range=TWO_DAYS,level=0)
         .with_match_keywords(match_kwd,time_range=ONE_DAY,level=1)
         .with_match_keywords(match_kwd,time_range=UNLIMITED,level=3)
```

The definition of time ranges is given to Section 6.2.3

If the `application.properties` configuration file has

- `association_preference = RAW`. First, the system will check if there are raw calibrations matching the first association rule. If found, they are associated with quality level flag = -1. If not found, raw calibrations matching the second association level level are searched (level = 0). If not found, the next level is searched until the last. If no raw calibrations are found for none of the levels, then master calibrations matching the first rule. If none are found, the second level is searched, and so forth. If no calibrations are found, the association is not done.

- `association_preference = MASTER`. Same as `RAW`, but first master calibrations are looked for all the association levels. Then, if master calibrations are not found, the system looks for raw calibrations.

- `association_preference = RAW_PER_LEVEL`. First, the system will check if there are raw calibrations matching the first association rule. If not found, MASTER calibrations matching the first rule are searched. If not found, RAW calibrations matching the second rule are searched, if not found MASTER calibrations matching the second rule are searched. The sequence goes on until the last rule.
- `association_preference = MASTER_PER_LEVEL`. First, the system will check if there are MASTER calibrations matching the first association rule. If not found, RAW calibrations matching the first rule are searched. If not found, MASTER calibrations matching the second rule are searched, if not found RAW calibrations matching the second rule are searched. The sequence goes on until the last rule.

6.2.3 Definition of validity ranges for associations

Time ranges are defined with the `RelativeTimeRange(`$-M$`,`N`)` function. The function considers a time range included between $-N$ days and $+M$ days around the file that needs the association. M and N are real numbers. The following time ranges are pre-defined in EDPS.

```
    ONE_AND_HALF_HOURS = RelativeTimeRange(-0.0625, 0.0625)
    SAME_NIGHT = RelativeTimeRange(-0.4, 0.4)
    NEXT_DAY = RelativeTimeRange(0, 1)
    ONE_DAY = RelativeTimeRange(-1, 1)
    TWO_DAYS = RelativeTimeRange(-2, 2)
    THREE_DAYS = RelativeTimeRange(-3, 3)
    FOUR_DAYS = RelativeTimeRange(-4, 4)
    FIVE_DAYS = RelativeTimeRange(-5, 5)
    ONE_WEEK = RelativeTimeRange(-7, 7)
    TWO_WEEKS = RelativeTimeRange(-14, 14)
    THREE_WEEKS = RelativeTimeRange(-21, 21)
    ONE_MONTH = RelativeTimeRange(-30, 30)
    QUARTERLY = RelativeTimeRange(-90, 90)
    IN_THE_PAST = RelativeTimeRange(NEGATIVE_INF, 0)
    UNLIMITED = RelativeTimeRange(NEGATIVE_INF, INF)
```

The above values must be imported in the `instrument_datasource.py` file, e.g.:

```
from edps.generator.time_range import *
```

6.3 The rules file: complex classifications and associations

This file contain the functions used for "complex" classification and association rules. The conversion is to start with the classification rules, followed by the association rules.

An example of "complex" classification rule is:

example_classification.py

```
from . import crires_rules as rules
CAL_WAVE_GAS_CELL = classification_rule('CAL_WAVE_TW', rules.is_gas_cell)
```

example_rules.py

```
def is_gas_cell(f):
    return f[kwd.pro_catg] == "CAL_WAVE_TW" and f[kwd.object] not in ["WAVE,FPET", "WAVE,
   UNE", "WAVE,SKY"]
```

It is recommended to use "nested" functions, as the above example.

An example of "complex" association rule is:

```
    #Data source in the data source file:
    raw_standard = (data_source('RAW_STANDARD')
              .with_classification_rule(standard_class)
              .with_group_keywords(['tpl.start'])
              .with_match_function(match_standard_science)
              .build())

    #Association rule in the instrument_rules.py file
    def match_standard_science(ref,f):
        return f['det.id'] == ref['det.id'] and \
           abs(f['airmass']-ref['airmass']) <= 0.1
```

In the above example, the standard star is associated if it has the same `det.id` and `airmass` difference not more than 0.1 than the main input of the task requiring the standard star. The convention used for the association functions is the following. The first input of the function, `ref`, is the "trigger", i.e. the file requesting the calibration. The second input of the function, `f`, is the file to associate. It is advisable to write the following comment into the rules file, right before the start of association functions:

```
 # ASSOCIATION RULES
 #  -  first input, e.g.  ref=trigger (e.g. science)
 #  -  second input, e.g.  f=file to associate (e.g. calibration)
```

7 The parameters file

7.1 "General" and "workflow" parameter files

Static parameters (e.g. Section 8) in the workflow and recipe parameters are stored in the so-called parameter file (in yaml format). The name of the parameter file should be <instrument>_parameters.yaml, and it should be located in the same directory where the workflows files are.

The structure of a workflow parameter file is (mind the spaces):

```
parameters_set1:  #name of the 1st parameter set
   is_default: yes/no #tells if this set is default)
   workflow_parameters: #list of static parameters
      param1: value1
      param2: value2
      ...
   recipe_parameters:  #list of task/recipe parametrs
     task1:
        recipe1.param1: value1
        recipe1.param2: value2
     task2:
        recipe2.param1: value3
        recipe2.param2: value4
     ...
parameters_set2:  #name of the 2nd parameter set
   is_default: yes/no #tells if this set is default)
   workflow_parameters: #list of static parameters
```

Notes:

- Different tasks can run the same recipe.
- If a recipe parameter is not listed, its default value is used.
- Only 1 default is admitted.
- It is not mandatory to specify multiply parameter sets. Technically, the file can be empty or not provided in `parameters.yaml` at all.
- Tasks are allowed a special parameter: `OMP_NUM_THREADS`: <value>, that defines the number of threads to be used by the recipe of that task, overriding (for that task only) the default specified in `application.properties`.
- workflow parameters or recipe parameters specified in the request call have precedence over those specified in the parameter file.
- parameters that are strings or boolean must be written in quotation marks (see example below).

7.2 Example

Example extracted from the MUSE EDPS workflow.

muse_parameters.yaml

```
qc1_parameters:
  is_default: yes
  workflow_parameters:
    lsfmode: "arc"
    skysubtraction: "auto"
    recompute_geometry: "no"
    recompute_astrometry: "no"
    wavelength_min: 4000.
    wavelength_max: 10000.
    telluric_correction: "TRUE"
    combine_science: "obs.id"
    max_diameter: 2
    max_separation: 0.25
    use_darks: "no"
  recipe_parameters:
    bias:
      muse.muse_bias.nifu: -1
      muse.muse_bias.merge: "TRUE"
    dark:
      muse.muse_dark.nifu: -1
      muse.muse_dark.merge: "TRUE"
    flat_lamp:
      muse.muse_flat.nifu: -1
      muse.muse_flat.merge: "TRUE"
    linearity_and_gain:
      muse.muse_lingain.nifu: -1
    wavelength:
      muse.muse_wavecal.nifu: -1
      muse.muse_wavecal.merge: "TRUE"
    line_spread_function:
      muse.muse_lsf.nifu: -1
      muse.muse_lsf.merge: "TRUE"
    line_spread_function_2:
      muse.muse_lsf.nifu: -1
      muse.muse_lsf.merge: "TRUE"
    throughput:
      muse.muse_ampl.nifu: -1
      muse.muse_ampl.savemaster: "TRUE"
      muse.muse_ampl.savetable: "TRUE"
      muse.muse_ampl.merge: "TRUE"
    preprocess_standard:
      muse.muse_scibasic.nifu: -1
      muse.muse_scibasic.merge: "TRUE"
    preprocess_science:
      muse.muse_scibasic.nifu: -1
      muse.muse_scibasic.merge: "TRUE"
    preprocess_astrometry:
      muse.muse_scibasic.nifu: -1
      muse.muse_scibasic.merge: "TRUE"
    preprocess_sky:
      muse.muse_scibasic.nifu: -1
      muse.muse_scibasic.merge: "TRUE"
    science:
      muse.muse_scipost.save: "cube,individual"
      muse.muse_scipost.format: "sdpCube"
```

```
56       muse.muse_scipost.skymodel_fraction: 0.4
57     science_sky:
58       muse.muse_scipost.save: "cube,individual"
59       muse.muse_scipost.format: "sdpCube"
60       muse.muse_scipost.skymodel_fraction: 0.4
61     science_combination:
62       muse.muse_exp_combine.format: "sdpCube"
63     mask:
64       OMP_NUM_THREADS: 12
65 test_parameters:
66   is_default: no
67   workflow_parameters:
68     lsfmode: "lsf"
69     skysubtraction: "auto"
70     recompute_geometry: "no"
71     recompute_astrometry: "yes"
72   recipe_parameters:
73     bias:
74       muse.muse_bias.nifu: -1
75       muse.muse_bias.merge: "TRUE"
76     dark:
77       muse.muse_dark.nifu: -1
78       muse.muse_dark.merge: "TRUE"
79     flat_lamp:
80       muse.muse_flat.nifu: -1
81       muse.muse_flat.merge: "TRUE"
82     linearity_and_gain:
83       muse.muse_lingain.nifu: -1
84     wavelength:
85       muse.muse_wavecal.nifu: -1
86       muse.muse_wavecal.merge: "TRUE"
87     line_spread_function:
88       muse.muse_lsf.nifu: -1
89       muse.muse_lsf.merge: "TRUE"
90     line_spread_function_2:
91       muse.muse_lsf.nifu: -1
92       muse.muse_lsf.merge: "TRUE"
93     throughput:
94       muse.muse_ampl.nifu: -1
95       muse.muse_ampl.savemaster: "TRUE"
96       muse.muse_ampl.savetable: "TRUE"
97       muse.muse_ampl.merge: "TRUE"
98     preprocess_standard:
99       muse.muse_scibasic.nifu: -1
100      muse.muse_scibasic.merge: "TRUE"
101    preprocess_science:
102      muse.muse_scibasic.nifu: -1
103      muse.muse_scibasic.merge: "TRUE"
104    preprocess_astrometry:
105      muse.muse_scibasic.nifu: -1
106      muse.muse_scibasic.merge: "TRUE"
107    preprocess_sky:
108      muse.muse_scibasic.nifu: -1
109      muse.muse_scibasic.merge: "TRUE"
110    science:
111      muse.muse_scipost.save: "cube,individual"
```

```
112       muse.muse_scipost.format: "sdpCube"
113       muse.muse_scipost.skymodel_fraction: 0.4
114     science_sky:
115       muse.muse_scipost.save: "cube,individual"
116       muse.muse_scipost.format: "sdpCube"
117       muse.muse_scipost.skymodel_fraction: 0.4
118     science_combination:
119       muse.muse_exp_combine.format: "sdpCube"
```

Part III

Advanced features

8 Tasks advanced features

8.1 Conditional association

It is possible to specify a condition under which a file (or a task) is associated to another task. Conditions can be based on “workflow parameters” (static) or can depend on the properties of the main input data “data driven conditions” (dynamic).

8.1.1 Conditional association based on static parameter

Example: associate the bias task to the science task only if it is the desired reduction strategy. The behaviour is regulated by a parameter (named, for example `use_bias` defined in the `parameter.yaml` file.

example_parameters.yaml

```
qc1_parameters:
  tags: [ qc1calib ]
  is_default: yes
  workflow_parameters:
    use_bias: "NO"
```

example_task_functions.yaml

```
def use_bias(params: JobParameters) -> bool:
    return get_parameter(params, "use_bias") == "YES"
```

example_wkf.py

```
from edps import task
from .demo_datasources import *
from .demo_task_functions import *
[...]
    science_task=(task("science")
          .with_main_input(raw_science)
          .with_associated_input(bias, [MASTERBIAS],condition=use_bias)
          .with_recipe('run_science')
          .build())
```

In the above example, we omit the definition of bias task, datasources, and classification rules.

If the workflow parameter "use_bias" is set to NO, then the condition (evaluated by the function use_bias) is False, and the bias is not associated. The job is considered complete.

If the workflow parameter use_bias is set to YES, then the condition (evaluated by the function use_bias) is True. If appropriate biases are found, then they are associated and the job is complete. If appropriate biases are not found, then no bias is associated and the job is incomplete.

8.1.2 Conditional association based on dynamic parameter

The condition can be evaluated on the basis of the properties of the main input.In the following example, we associate the bias only if the `INS.MODE` header keyword of the main input is set to `IMG`.

example_parameters.yaml

```
1 def which_ins_mode(files: List[ClassifiedFitsFile]):
2     ins_mode =  files[0].get_keyword_value("ins.mode", None)
3     return ins_mode
4
5 def is_img(params: JobParameters) -> bool:
6     return params.get_workflow_param('ins_mode') == 'IMG'
```

example_wkf.py

```
1 from edps import task
2 from .demo_datasources import *
3 from .demo_task_functions import *
4 [...]
5     science_task=(task('SCIENCE')
6          .with_main_input(raw_science)
7          .with_dynamic_parameter("ins_mode", which_ins_mode)
8          .with_associated_input(bias,[MASTERBIAS],condition=is_img)
9          .with_recipe('run_science')
10         .build())
```

In this example, the main input of the task (that can change for every job) are inspected by the `which_ins_mode` function. The task defines a parameter (named `ins_mode`) whose value is computed by the function `which_ins_mode`. Because the inputs to the task can vary, the parameter is called “dynamic”. The condition is evaluated by the function `is_img`, which returns True if the `ins_mode` parameter is equal to `IMG`; otherwise it returns False.

As for the case of static parameters, the job is not checked for completeness if the association condition is False.

8.2 Execute a task under certain conditions

The execution of a task can be subordinated to certain conditions, expressed via a workflow parameter. This is useful, for example, when specifying the reduction cascade (activate certain tasks instead of others). The conditions that triggers the task is as in Section 8.1.1.

Example:

example_parameters.yaml

```
1 qc1_parameters:
2   tags: [ qc1calib ]
3   is_default: yes
4   workflow_parameters:
5     use_sky_flats: "NO"
```

example_task_functions.py

```

def use_sky_flats(params: JobParameters) -> bool:
    return get_parameter(params, "use_sky_flats") == 'YES'
```

example_wkf.py

```
[...]
     sky_flat=(task('SKYFLAT')
           .with_condition(use_sky_flat)
           .with_main_input(raw_sky_flats)
           .with_recipe("run_sky_flats")
           .build())
```

In the example, the tasks `SKYFLAT` is executed only if the recipe parameter "`use_sky_flats`" is set to "YES". In the example, we omit import statements and the definition of datasources and classification rules. Section 12 present a similar example within a complete workflow.

8.3 How to specify alternative inputs

Alternative inputs can be specified with the object alternative_associated_input. The object lists a series of alternatives, if the first is not found it goes to the second and so forth. If the first is found, it is returned and the others are not associated.

Example:

```
from edps import task, alternative_associated_inputs

  calibrations=(alternative_associated_input()
          .with_associated_input(bias,[MASTERBIAS])
          .with_associated_input(dark,[MASTERDARK]))

  task = (task("reduction")
         .with_recipe("run_reduction")
         .with_main_input(raw_science)
         .with_alternatives(calibrations)
         .build())
```

In the example above, bias calibrations have the precedence: if found they are associated to the task. Otherwise dark calibrations are associated. If none are present, the task is incomplete. If the calibrations are optional, add min_ret=0 to the last alternative in the list (dark, in this example).

The `.with_associated_inputs()` methods inside `alternative_associated_inputs()` have the same properties as if they were used in a task: `min/max_ret`, `condition`, and `match_rules`.

8.4 Setting the recipe parameters

The parameters of a recipe in a task can be set in several ways, that are listed here in order of precedence (first has precedence)

- Using the option `-rp TASK PARAMETER VALUE` in the `edps-client` request: e.g.:

```
edps -w muse.muse_wkf -rp bias muse_bias  muse.muse_bias.nifu 0
```

- Via the instrument_parameters.yaml file (see Section: 7).
- Via a job function (see Section 8.5)
- No particular specification: the recipe default is used.

8.5 Modifying the job properties

It is possible to modify at run-time the properties of a job (e.g., the values of the recipe parameters) depending on the properties of the input data. In the following example, taken from the UVES workflow, we modify the job created by the task "object" to adjust the recipe parameter depending on the input data.

uves_task_functions.py

```
def object_type(job: Job):
    # Setting some recipe parameters in uves_obs_scired depending if the main input is extended or point-like
    #Note: job.command is the recipe name.
    reduce_extract_method = f'{job.command}.reduce.extract.method'  #full name of recipe paramter
    reduce_backsub_method = f'{job.command}.reduce.backsub.mmethod' #full name of recipe paramter
    dp_types = [f.get_keyword_value("dpr.type", None) for f in job.input_files]
    if "OBJECT,EXTENDED" in dp_types or "OBJECT,EXTENDED,EXTENDED" in dp_types:
        #Parameters to be set in the case of extended objects.
        job.parameters.recipe_parameters[reduce_extract_method] = "2d"
        job.parameters.recipe_parameters[reduce_backsub_method] = "minimum"
    elif "OBJECT,POINT" in dp_types or "OBJECT,POINT,POINT" in dp_types:
        #Parameters to be se in case of point-like objects.
        job.parameters.recipe_parameters[reduce_extract_method] = "optimal"
        job.parameters.recipe_parameters[reduce_backsub_method] = "median"
```

uves_wkf.py

```
from . import uves_task_functions
# Task to reduce long slit observations
    science_slit = (task("object")
                    .with_recipe("uves_obs_scired")
                    .with_main_input(raw_science)
                    [...]
                    .with_job_processing(object_type)  # set recipe params depending on target (extended or point-like)
                    .build())
```

The following example, taken from the KMOS workflow, is design to pass only the last input file to the recipe.

kmos_fit_profiles.py

```
[...]
# Process only the last acquisition
def process_last_file(job: Job):
    job.input_files = [job.input_files[-1]]

acquisition_reconstruct = (task('acquisition_reconstruct')
            .with_recipe('kmos_reconstruct')
             [...]
            .with_job_processing(process_last_file)# Process only the last main input file.
            .build())
```

8.6 Modifying the input category tag

For a certain recipe in the reduction chain, the input category of a file, as specify in the sof, can differ from the category the file is classified in the classification rules. In order to change the tag of the files that are going to be written in the recipe sof, the `.with_input_map()` method in the task can be used.

Example:

```
reduction = (task("task")
             .with_recipe("recipe_name")
             .with_main_input(raw_input)
             .with_associated_input(calibration)
             .with_input_map({CATEGORY1: NEW_CATEGORY1,
              CATEGORY2: NEW_CATEGORY2})
             .build())
```

In the above example, the inputs that are tagged as CATEGORY1 will be tagged in the sof that will be given as input to the recipe `recipe_name` as `NEW_CATEGORY1`. Similarly, the `CATEGORY2` inputs will be tagged as `NEW_CATEGORY2`. This renaming does not alter the header of the files.

9 Running python scripts, looping on recipes

The task is primarely designed to run recipe pipelines specified with the `.with_recipe("recipe_name")` method. It is possible, however, pass a function that runs a script. In the following example we show the task that performs the telluric correction on FORS2 spectra. The task (named `telluric_correction`) runs a function (also named `telluric_correction`) instead of a recipe.

The function performs the following steps

- Runs a Python function that selects, among the input spectra to correcgt, the one with higher signal-to-noise (read from the file header). Let's call it reference spectrum.
- Runs the recipe fors_molecfit_model on the selected reference spectrum.
- Loops on all the input spectra to correct, running the following recipes:
  - Generates a telluric correction for the input spectrum, using the model results obtained from the reference spectrum (recipe fors_molecfit_calctrans).
  - Corrects the input spectrum for telluric absorption (recipe fors_molecfit_correct).

fors_spec_wkf.py

```
1 [...]
2  # --- Task for telluric correction (only for Long Slit data).
3  telluric = (task('telluric_correction')
4              .with_function(telluric_correction)
5              .with_main_input(science)
6              [...]
7              .build())
```

fors_task_functions.py

```
1  from astropy.io import fits
2  from edps import File, FitsFile
3  from edps import List, ClassifiedFitsFile, JobParameters, get_parameter, Job
4  from edps import RecipeInvocationArguments, RecipeInvocationResult, InvokerProvider,
       RecipeInputs, ProductRenamer
5
6  from . import fors_keywords as kwd
7
8
9  # --- FUNCTIONS TO PERFORM THE TELLURIC CORRECTION -------------------------------------
10
11 # --- This function checks the parameter telluric_correction in the fors_parameters.yaml
12 # file and determines whether the telluric correction has to be carried on or not.
13
14 def perform_telluric_correction(params: JobParameters) -> bool:
15     return get_parameter(params, "telluric_correction") == "TRUE"
16
17
18 # --- This function finds the spectrum with higher signal-to-noise
19 def find_reference_file(science_files: List[File]):
```

```
20      max_snr = 0
21      reference_file = None
22      for file in science_files:
23          with fits.open(file.file_path) as hdus:
24              snr = hdus[0].header['SNR']
25              wavelen_max = hdus[0].header['WAVELMAX']
26              if snr > max_snr and wavelen_max > 680.:
27                  max_snr = snr
28                  reference_file = file
29      if reference_file:
30          return File(reference_file.file_path, reference_file.category, ""), max_snr
31      else:
32          return None, None
33
34
35  # --- Generic function to run a recipe
36  def run_recipe(input_file, associated_files, parameters, recipe_name, args, invoker,
        renamer) -> (
37          RecipeInvocationResult, List):
38
39      # input_file: main input and category. Format: File(string_with_full_path,
40      #                                                    string_with_category, "")
41      # associated_files: calibrations. Format List[file], where files have the format:
42      #                           File(string_with_full_path, string_with_category, "")
43      # parameters: non default recipe parameters. Format {'parameter_name1': value1,
44      #                                                       'parameter_name2': value2}
45      # recipe_name: recipe name  Format: string
46      # args, invoker: extra stuff provided by the task that calls the function
47      #                             calling run_recipe()
48
49      inputs = RecipeInputs(main_upstream_inputs=[input_file],
50                            associated_upstream_inputs=associated_files)
51      arguments = RecipeInvocationArguments(inputs=inputs, parameters=parameters,
52                                            job_dir=args.job_dir, input_map={},
53                                            logging_prefix=args.logging_prefix)
54
55      results = invoker.invoke(recipe_name, arguments, renamer, create_subdir=True)
56      output_files = [File(f.name, f.category, "") for f in results.output_files]
57      return results, output_files
58
59
60  # --- This function coordinates the telluric correction process.
61  #   - It finds the spectrum with higher S/N
62  #   - It runs fors_molecfit_model on the higher S/N spectrum and determines the
63  #     properties of the atmosphere.
64  #   - For each spectrum to correct, it
65  #       - computes the full telluric correction spectrum (fors_molecfit_calctrans)
66  #       - applies the correction to the science spectrum.
67  #
68  def telluric_correction(args: RecipeInvocationArguments,
69                          invoker_provider: InvokerProvider,
70                          renamer: ProductRenamer) -> RecipeInvocationResult:
71      invoker = invoker_provider.recipe_invoker
72      results = []
73      ret_codes = []
74      calibration_categories = ['MOLECULES', 'WAVE_INCLUDE', 'WAVE_EXCLUDE', 'PIX_EXCLUDE',
```

```
75                              'ATM_PROFILE_STANDARD','KERNEL_LIBRARY', 'GDAS']
76      science_categories = ['REDUCED_IDP_SCI_LSS']
77
78      calibration_files = [File(f.name, f.category, "") for f in args.inputs.combined if
79                           f.category in calibration_categories]
80      science_files = [File(f.name, f.category, "") for f in args.inputs.combined if
81                       f.category in science_categories]
82
83      # find the file with higher signal-to-noise
84      reference_file, max_snr = find_reference_file(science_files)
85
86      # run fors_molecfit_model on reference_file if available
87      if reference_file:
88          molefit_parameters = {'WAVE_INCLUDE': '0.61,0.64,0.68,0.71,0.711,0.740,0.75,0.78,
89                                 0.81,0.84,0.91,0.95,0.96,0.98',
90                                'LIST_MOLEC': 'H2O, O2'}
91
92          result_reference, reference_files = run_recipe(reference_file, calibration_files,
93                        molefit_parameters, 'fors_molecfit_model', args, invoker, renamer)
94
95          ret_codes.append(result_reference.return_code)
96          reference_files = [File(f.name, f.category, "")
97                             for f in result_reference.output_files]
98
99          # LOOPING fors_molecfit_calctrans and fors_molecfit_correct, using results from
    the
100         # reference file for all the inputs
101         for science_file in science_files:
102             # run fors_molecfit_calctrans and for_molecfit_correct
103             results_calctrans, caltrans_output_files = run_recipe(science_file,
    reference_files, {},
104                                                 'fors_molecfit_calctrans', args, invoker,
    renamer)
105             result_correct, correct_output_files = run_recipe(science_file,
    caltrans_output_files, {},
106                                                 'fors_molecfit_correct', args, invoker,
    renamer)
107
108             ret_codes.extend([results_calctrans.return_code, result_correct.return_code])
109             results.append(result_correct)
110
111         # Construct final RecipeInvocationResult
112         ret_code = min(ret_codes)
113         output_files = [FitsFile(name=f[0].name, category=f[0].category)
114                         for f in ([r.output_files for r in results])]
115         corrected_files = RecipeInvocationResult(return_code=ret_code,
116                                                  output_files=output_files)
117     else:
118         output_files = [FitsFile(name=f.name, category=f.category)
119                         for f in args.inputs.combined]
120         corrected_files = RecipeInvocationResult(return_code=0,
121                                                  output_files=output_files)
122     return corrected_files
123
124
125 # -----------------------------------------------------
```

10 Associations: advanced features

10.1 Tasks requiring the same datasources but with different association rules

There could be the case where the same datasouce (e.g. calibration) is required by two different tasks using two different rules. In this case, the association rule (or matching keywords) does not need to be specified in the datasource, but it can be specified with a `match_rule` object, that is passed to the task. For example:

```
from edps import classification_rule, data_source, task, match_rule

  calibration_class = classification_rule("CALIB",{"dpr.type": CALIB})
  science_class     = classification_rule("SCIENCE",{"dpr.tpye": SCIENCE})

  raw_calibration = (data_source("CALIBRATION")
                  .with_classification_rules(calibration_class)
                  .with_group_keywords("arcfile")
                  .with_match_keywords(["instrume", "ins.grating", "ins.filter"],
                                       time_range=ONE_DAY)
                  .build())

  raw_science = (data_source("SCIENCE")
               .with_classification_rule(science_class)
               .with_group_keywords(["arcfile"])
               .build())

  task1   = (task("reduction1"))
             .with_recipe("recipe1")
             .with_main_input(science)
             .with_associated_input(calibration)
             .build())

  new_rule = (match_rules()
              .with_match_keywords(["instrume", "ins.filter"],   time_range="ONE_MONTH"))

  task2   = (task("reduction2"))
            .with_recipe("recipe2")
            .with_main_input(science)
            .with_associated_input(calibration, match_rules=new_rule)
            .build())
```

In the example above, the task `reduction1` uses the rules attached to the calibration `data_source` to find a calibration for the science input. The task `reduction2` uses the rules defined by `new_rule`, overriding those defined in the `data_source`.

10.2 Optional and mandatory products of a task

A task (e.g., `calibration`) can generate several products (e.g., `MASTER1` and `MASTER2`), some of them might be required and other optional by one task. They cannot be specified together inside

the same .with_associated_input because it accepts only one min_/max_ret input. They must be associated using alternatives

In the below example, the task `science` requires the products from the `calibration` task; however MASTER1 is mandatory, whereas MASTER2 is optional

```
from edps import classification_rule, data_source, task, alternative_associations

  calibration_class = classification_rule("CALIB", {"dpr.type": "CALIB"})
  science_class     = classification_rule("SCIENCE", {"dpr.tpye": "SCIENCE"})

  raw_calibration = (data_source("CALIBRATION")
                  .with_classification_rules(calibration_class)
                  .with_group_keywords("arcfile")
                  .with_match_keywords(["instrume", "ins.filter"])
                  .build())

  raw_science    = (data_source("SCIENCE")
                 .with_classification_rule(science_class)
                 .with_group_keywords(["arcfile"])
                 .build())

  calibration    = (task("CALIBRATION")
                 .with_recipe("calibration_recipe")
                 .with_main_input(raw_calibration)
                 .build())

#First line all calibrations. Second line only mandatory calibrations
  calibration_files = (alternative_associatiated_inputs(
                      .with_associated_input(calibration, [MASTER1, MASTER2])
                      .with_associated_input(calibration, [MASTER1]))

  science = (task("SCIENCE"))
            .with_recipe("science_recipe")
            .with_main_input(science)
            .with_alternatives(calibration_files)
            .build())
```

In the example above we assume the association preferences to be set to "master". If both MASTER1 (mandatory) and MASTER2 (optional) are found, then both are associated. If only MASTER1 is found, then only that one is associated and the task is still complete. If none, or if only MASTER2 is found, then the task is incomplete. If the association preference were set to "raw", then the raw calibrations are processed and both products (MASTER1 and MASTER2) are sent to the task "SCIENCE".

11 Subworkflows

11.1 General concepts

EDPS allows the inclusion of sub workflows within a workflow. Subworkflows are the equivalent of sub-routines in a program, and therefore they can be an useful way to "isolate" a portion of the reduction cascade with precise input/output. Also, a subworkflow

- can be reused in the same or in other workflows.
- can be used to "hide" part of the reduction chain, that might contain a number of steps but only the final task(s) are needed in the rest of the reduction cascade.
- can be used to simplify the overall workflow structure, as the sub-workflow appears as a single block.

Important note: all the tasks in a workflow, including those in the subworkflow, must have unique names. Therefore, if the same workflow is used several times within a workflow, it must be inserted so that tasks names are unique.

11.2 Example

Example extracted from the muse workflow.

muse_wkf.py

```
from .muse_response import process_standard
[...]
# --- Subworkflow to generate response curve and telluric correction
response = process_standard(bias, dark, lamp_flat, wavelength, geometry_calibrations,
    sky_flat)
```

muse_response.py

```

from edps import subworkflow, task

# This subworkflow reduces standard star observations and produces
# response curve and telluric correction

@subworkflow("response", "")
def process_standard(bias, dark, lamp_flat, wavelength, geometry_calibrations, sky_flat):

    # --- Pre-process standard star raw calibrations
    standard = (task("preprocess_standard")
                .with_recipe("muse_scibasic")
                .with_main_input(raw_std)
                .with_associated_input(badpix_table, min_ret=0)
                .with_associated_input(bias, [MASTER_BIAS])
                .with_associated_input(lamp_flat, [TRACE_TABLE, MASTER_FLAT])
```

```
17                .with_associated_input(wavelength, [WAVECAL_TABLE])
18                .with_alternatives(geometry_calibrations)
19                .with_associated_input(sky_flat, [TWILIGHT_CUBE], min_ret=0)
20                .with_associated_input(raw_flat_illum, min_ret=0)
21                .build())
22
23    #--- Generation of response curve and telluric correction
24    response = (task("response")
25                .with_recipe("muse_standard")
26                .with_main_input(preprocess_standard)
27                .with_associated_input(extinct_table)
28                .with_associated_input(std_flux_table)
29                .with_associated_input(telluric_regions, min_ret=0)
30                .with_associated_input(filter_list, min_ret=0)
31                .with_meta_targets([QC0, CALCHECKER, QC1_CALIB])
32                .build())
33
34    return response
```

Part IV

Workflow examples

12 A complete simple workflow

In this section we implement a workflow as in 3.1 with the addition of classification, and association a detailed by an "hypothetical" instrument and pipeline.

The types of data, their association rules and the data reduction pipeline are designed according to the following specifications.

The Workflow graphic representation is exactly as in Figure 3.1.

The biases are processed with run_bias, that needs at least 3 files with DPR.TYPE = BIAS that must have the same ARM and TPL.START header keywords. The product is named MASTERBIAS. Its validity range according to the calibration plan is 1 day, but biases as old as 1 week are sufficient.

```
-- Recipe: run_bias -----------------------------------------
Inputs    number    Grouping keywords  Match keywords
BIAS        3       TPL.START, ARM        N/A
Products:
MASTERBIAS (validity: 1 day)
Validity ranges:
 Calibration plan: 1 day
 Sufficient quality for certification: 1 week
-------------------------------------------------------------
```

The flats are processed with run_flat, that needs at least 3 files with DPR.TYPE = FLATS that must have the same ARM and TPL.START header keywords. A MASTERFLAT is needed as associated input. The product is named MASTERFLAT. Its validity range according to the calibration plan is 1 day, but biases as old as 1 week are sufficient.

```
-- Recipe: run_flat -----------------------------------------
Inputs    number    Grouping Keywords  Match keywords
FLAT        3       TPL.START, ARM      N/A

MASTERBIAS 1        N/A                 INSTRUME, ARM,
                                        INS.SLIT

Products:
MASTERFLAT(validity: 1 night)
Validity ranges:
 Calibration plan: 1 day
 Sufficient quality for certification: 1 week
-------------------------------------------------------------
```

Science frames are processed individually. They need a MASTERFLAT and a MASTERBIAS, that must match ARM, INSTRUME, and INS.SLIT. A SKY frame matching ARM and INS.SLIT is also

needed; in the case of VIS observations, it has also to match TPL.START (to follow the calibration plan) or within 1 day for sufficient precision. In the case of NIR observations, it has to match TPL.START otherwise the quality of the reduction is not good.

Optional calibration CATALOGUE is optional, and has to be provided only if photometric calibration is needed. In the case of NIR observations, the input CATALOG has to match the ARM of the observations. In the case of VIS observations, the input CATALOG needs to have either ARM=VIR or ARM=RED.

```
-- Recipe: run_science -------------------------------------
Inputs     number   Grouping keywords    Match keywords
SCIENCE     1        ARCFILE               N/A

SKY         1        ARCFILE               ARM, INS.SLIT (for VIS and NIR)
                                           TPL.START (preferred) or
                                           within 24 hrs, if VIS.
                                           TPL.START only if NIR.

MASTERBIAS 1         N/A                   INSTRUME, ARM, INS.SLIT

MASTERFLAT 1         N/A                   INSTRUME, ARM, INS.SLIT

CATALOGUE   0        N/A                   ARM=NIR for NIR observations
                                           ARM=RED or ARM = VIS for VIS observations.

Product:
REDUCED_SCIENCE
```

demo1_wkf.py

```
from edps import task, SCIENCE, alternative_association
from .demo1_datasources import *
from .demo1_rules import *
from .demo1_task_functions import *

# --- Processing tasks -------------------------------------------------------------------

# - Task to reduce raw biases
bias_task = (task("bias")
             .with_recipe("run_bias")
             .with_main_input(raw_bias)
             .build())

# - Task to reduce raw flats
flat_task = (task("flat")
             .with_recipe("run_flat")
             .with_main_input(raw_flat)
             .with_associated_input(bias_task, [MASTERBIAS])
             .build())
```

```
20
21 # The sky is associated under certain conditions, that depends on the properties of
22 # the input data.
23 # If the input is taken with arm=VIS, then the rules attach_sky_vis are used to
24 # associate the sky exposure.
25 # If the input is taken with arm=NIR, then the rules attach_sky_nir are used to
26 # associate the sky exposure.
27 alternative_sky = (alternative_association()
28     .with_associated_input(raw_sky,  condition=is_input_VIS, match_rules=attach_sky_vis)
29     .with_associated_input(raw_sky,  condition=is_input_NIR, match_rules=attach_sky_nir))
30
31 # - Task to reduce science observations.
32 # If the user wants to do photometric calibration, then a static catalogue is associated,
33 # otherwise it is not associated. The choice is done by setting the parameter
34 # flux_calibration in the demo1_parameters.file
35 # flux_calibration = "TRUE", the static catalogue is associated, and the recipe
36 #                             performs the flux calibration
37
38 science_task = (task("object")
39           .with_recipe("run_science")
40           .with_dynamic_parameter("arm_used", which_arm)
41           .with_main_input(raw_science)
42           .with_alternative_associated_inputs(alternative_sky)
43           .with_associated_input(bias_task, [MASTERBIAS])
44           .with_associated_input(flat_task, [MASTERFLAT])
45           .with_associated_input(static_catalog, condition=perfor_photometric_calibration)
46           .with_meta_targets([SCIENCE])
47           .build())
```

demo1_datasources.py

```
1 from edps import data_source, RelativeTimeRange
2 from .demo1_classification import *
3 from . import demo1_keywords as kwd
4 from edps.generator.time_range import UNLIMITED, ONE_DAY
5
6 # Convention for Data sources Association rule levels:
7 # Each data source can have several match function which correspond to different
8 # quality levels for the selected data. The level is specified as a number that
9 # follows this convention:
10 #   level < 0: more restrictive than the calibration plan
11 #   level = 0 follows the calibration plan
12 #   level = 1 quality sufficient for QC1 certification
13 #   level = 2 probably still acceptable quality
14 #   level = 3 significant risk of bad quality results
15
16 # --- General variables -----------------------------------------------------------------
17 # Header keywords that defines the instrument setup
18 setup = [kwd.tpl_start, kwd.arm, kwd.ins_slit]
19 # Header keywords to be used for the grouping
20 grouping = [kwd.tpl_start, kwd.arm]
21
22 # --- Raw data sources ------------------------------------------------------------------
23
24 # Raw biases
25 raw_bias = (data_source("BIAS")
```

```
26            .with_classification_rule(bias_class)
27            .with_min_group_size(3)
28            .with_setup_keywords(setup)
29            .with_grouping_keywords(grouping)
30            .with_match_keywords([kwd.arm], time_range=RelativeTimeRange(-3, 3), level=0)
31            .with_match_keywords([kwd.arm], time_range=UNLIMITED, level=3)
32            .build())
33
34 # Raw flats
35 raw_flat = (data_source("FLAT")
36            .with_classification_rule(flat_class)
37            .with_min_group_size(3)
38            .with_setup_keywords(setup)
39            .with_grouping_keywords(grouping)
40            .with_match_keywords([kwd.arm, kwd.ins_slit], time_range=ONE_DAY, level=0)
41            .with_match_keywords([kwd.arm, kwd.ins_slit], time_range=UNLIMITED, level=3)
42            .build())
43
44 # Raw sky exposures
45 raw_sky = (data_source("SKY")
46            .with_classification_rule(sky_class)
47            .with_grouping_keywords([kwd.arcfile])
48            .build())
49
50 # Raw science exposures
51 raw_science = (data_source("OBJECT")
52              .with_classification_rule(science_class)
53              .with_grouping_keywords([kwd.arcfile])
54              .build())
55
56 # --- Static calibrations ---------------------------------------------------------------
57
58 # Catalogue of standard stars
59 static_catalog = (data_source("catalog")
60               .with_classification_rule(static_catalog_class)
61               .with_match_function(rules.is_assoc_catalogue, time_range=UNLIMITED)
62               .build())
```

demo1_classification.py

```
1 from edps import classification_rule
2 from . import demo1_keywords as kwd
3 from . import demo1_rules as rules
4
5 # Dictionaries with general keywords
6 demo = {kwd.instrume: "DEMO"}
7 calib_keywords = {**demo, kwd.dpr_catg: "CALIB"}
8 science_keywords = {**demo, kwd.dpr_catg: "SCIENCE"}
9
10 # RAW FILES
11 bias_class = classification_rule('BIAS', {**calib_keywords, kwd.dpr_type: "BIAS"})
12 flat_class = classification_rule('FLAT', {**calib_keywords, kwd.dpr_type: "FLAT"})
13 science_class = classification_rule('SCIENCE', {**science_keywords, kwd.dpr_type: "OBJECT"
    })
14
15 sky_class = classification_rule('SKY', {**science_keywords, kwd.dpr_type: "SKY"})
```

```
16
17 # MASTER CALIBRATIONS
18 MASTERBIAS = classification_rule("MASTERBIAS", {**demo, kwd.pro_catg: "MASTERBIAS"})
19 MASTERFLAT = classification_rule('MASTERFLAT', {**demo, kwd.pro_catg: 'MASTERFLAT'})
20
21 # STATIC CALIBRATIONS
22 static_catalog_class = classification_rule('CATALOG', {**demo, kwd.pro_catg: 'CATALOG'})
```

demo1_rules.py

```
1  # CLASSIFICATION RULES
2  from edps.generator.time_range import *
3  from edps import match_rules
4  from . import demo1_keywords as kwd
5
6
7  def is_demo(f):
8      return f[kwd.instrume] == "DEMO"
9
10
11 # The DPR.TYPE of sky exposures can be either SKY or OFFSET_SKY.
12 def is_sky(f):
13     return is_demo(f) and f[kwd.dpr_type] in ["SKY", "OFFSET_SKY"]
14
15
16 # ASSOCIATION RULES
17 #  - first, e.g.  ref=trigger (e.g. science)
18 #  - second, e.g.  f=file to associate (e.g. calibration)
19
20 def is_assoc_catalogue(ref, f):
21     # I must have a match between arms, or for the VIS arm, I can associate a catalogue
       with ARM=RED.
22     return f[kwd.arm] == ref[kwd.arm] or (f[kwd.arm] == "RED" and ref[kwd.arm] == "VIS")
23
24
25 # ASSOCIATION RULES THAT OVERRIDES THOSE IN SPECIFIED IN THE DATA_SOURCE
26
27 setup = [kwd.instrume, kwd.arm, kwd.ins_slit]
28
29 # The sky in the visual can be from the same night for decent quality
30 attach_sky_vis = (match_rules()
31                   .with_match_keywords(setup + [kwd.tpl_start], level=0)
32                   .with_match_keywords(setup, time_range=SAME_NIGHT, level=1))
33
34 # The sky in the NIR must be from the same template as the observation.
35 attach_sky_nir = (match_rules()
36                   .with_match_keywords(setup + [kwd.tpl_start], level=0))
```

demo1_keywords.py

```
1 # HEADER KEYWORDS USED FOR CLASSIFICATION, GROUPING, AND ASSOCIATION.
2 instrume = "instrume"
3 arm = "ARM"
4 ins_slit = "ins.slit"
5 tpl_start = "tpl.start"
6 mjd_obs = "mjd-obs"
```

```
dpr_type = "dpr.type"
dpr_catg = "dpr.catg"
pro_catg = "pro.catg"
arcfile = "arcfile"
```

demo1_task_functions.py

```
from edps import get_parameter, JobParameters, ClassifiedFitsFile, List

#--- Functions to determine the ARM of the input data------------------------------------
def is_input_NIR(params: JobParameters) -> bool:
    return get_parameter(params, "arm_used") == "NIR"


def is_input_VIS(params: JobParameters) -> bool:
    return get_parameter(params, "arm_used") == "VIS"


def which_arm(files: List[ClassifiedFitsFile]):
    # Note: files are only the main input files, not the associated files
    arm = files[0].get_keyword_value("ARM", None)
    return arm

#--- Function to read workflow parameters -----------------------------------------------
def perfor_photometric_calibration(params: JobParameters) -> bool:
    return get_parameter(params, "flux_calibration") == "TRUE"
```

demo1_parameters.yaml

```
qc1_parameters:
  tags: [ qc1calib ]
  is_default: yes
  workflow_parameters:
    # flux_calibration = "TRUE", the static catalogue is associated, and the recipe
    #                            performs the flux calibration. Otherwise, the static
    #                            catalogue is not associated to the dataset.
    flux_calibration: "TRUE"
  recipe_parameters:
    #List the recipe parameters, ordered per task, to be adopted
    #For the parameters not listed,  therecipe default are adopted.
    object:
      demo.run_science.trim: "TRUE"
```

13 Compacting the workflow: merged tasks and merged classification rules.

In this section we re-design a workflow following the same principals of the reduction cascade of 12, but adding extra constraints driven by different instrument and pipeline designs.

The types of data, their association rules and the data reduction pipeline are designed according to the specifications detailed in Section 13.1.

The general rule is to keep a workflow as simple as possible, minimising the number of tasks and datasources whenever possible. There are two main cases where it is possible merge tasks and classification rules.

If two tasks share the same main input and recipe, it should be possible to combine them in a single task, regardless they differ in terms of associated inputs or other parameters.

Also, if several categories of files can be processed by the same recipe, and have the same association rules, it should be possible to include them in the same data source

Keeping these consideration in mind, we present two different workflow designs that exploit different methods to combine tasks and datasources. The first design (Section 13.2) is a "straightforward translation" of the data reduction requirement, the second design (Section 13.3) is a "simplified" version that merges tasks and combines classification rules. This second approach should be adopted whenever possible.

13.1 Pipeline description

The bias recipe (run_bias) needs at least 3 raw bias input frames DPR.TYPE = BIAS). The inputs have to be grouped by the TPL.START, ARM, and INS1.SLIT keyewords. The classification to be written in the input sof must be either BIAS_VIS or BIAS_NIR, depending on the type of the inputs. The products are MASTERBIAS_VIS or MASTERBIAS_NIR, depending on inputs. The calibration plan foresees to take biases every day, but their validity for scientifically valid reduction is 1 week.

```
-- Recipe: run_bias ------------------------------------------------
Inputs    number    Grouping keywords  Match keywords dpr.type
BIAS_VIS    3       TPL.START, ARM        N/A         BIAS
                    INS1.SLIT

or;

BIAS_NIR    3       TPL.START, ARM        N/A         BIAS
                    INS2.SLIT

Products:
MASTERBIAS_VIS or
MASTERBIAS_NIR
```

```
Validity ranges:
 Calibration plan: 1 day
 Sufficient quality for certification: 1 week

--------------------------------------------------------------------
```

The recipe to process the flat fields (run_flat) requires 3 flats as main input (DPR.TYPE = FLAT), and they need to be flagged as FLAT_VIS or FLAT_NIR in the recipe input sof. The inputs have to be grouped by TPL.START, ARM, and INS1.SLIT and INS2.SLIT. The recipe also needs a masterbias, matching the same INSTRUME, ARM, and INS1.SLIT (if VIS observations) or INS2.SLIT (if NIR observations) as the input flats. The calibration plan foresees the acquisition of flats every day, but a flat from 1 week is sufficient for scientifically valid results. Optional inputs for the science is an offset sky (DPR.TYPE = SKY), that has to match the same ARM, INSTRUME and INS1.SLIT (if VIS observations) or INS2.SLIT (if NIR observation). The sky has to be either of the same tpl.start of the science, or within the same night for a quick check. If observations are in NIR, also a static catalogue is needed.

```
-- Recipe: run_flat ---------------------------------------------------
Inputs    number    Grouping Keywords  Match keywords          dpr.type
FLAT_VIS    3       TPL.START, ARM      N/A                       FLAT
                    INS1.SLIT
MASTERBIAS_VIS 1    N/A                 INSTRUME, ARM, INS1.SLIT  N/A

or:

FLAT_NIR    3       TPL.START, ARM      N/A                       FLAT
                    INS2.SLIT
MASTERBIAS_NIR 1    N/A                 INSTRUME, ARM, INS2.SLIT  N/A

Products:
MASTERFLAT_VIS or
MASTERFLAT_NIR
Validity ranges:
 Calibration plan: 1 day
 Sufficient quality for certification: 1 week
 --------------------------------------------------------------------
\end{verbsatim}
The science recipe (run\_science) reduces science inputs independently (DPR.TYPE =
\begin{verbatim}
-- Recipe: run_science ------------------------------------------------
Inputs     number   Grouping keywords   Match keywords        dpr.type
SCIENCE_VIS 1       MJD.OBS             N/A                    SCIENCE
SKY          0      N/A                 ARM, INS1.SLIT         SKY
                                        TPL.START (preferred) or
                                        within 24 hrs (quick check only)
```

```
MASTERBIAS_VIS 1    N/A                 INSTRUME, ARM, INS1.SLIT
MASTERFLAT_VIS 1    N/A                 INSTRUME, ARM, INS1.SLIT

or:

SCIENCE_NIR 1       MJD.OBS             N/A                     SCIENCE
SKY         0       N/A                 ARM, INS2.SLIT          SKY
                                        TPL.START (preferred) or
                                        within 24 hrs (quick check only)
MASTERBIAS_NIR 1    N/A                 INSTRUME, ARM, INS2.SLIT
MASTERFLAT_NIR 1    N/A                 INSTRUME, ARM, INS2.SLIT

CATALOG             N/A                 ARM

Product:
REDUCED_SCIENCE

----------------------------------------------------------------------
Notes:
1) ARM=VIS have slit width defined in INS1.SLIT. Values in INS2.SLIT
   could be wrong.
2) ARM=NIR have slit width defined in INS2.SLIT. Values in INS1.SLIT
   could be wrong.

3) A single observing template generates either VIS or NIR exposures.
```

13.2 First workflow.

This workflow represent a "straightforward translation" of the pipeline and instrument description. Each tag is assigned a datasource and a task for processing. For the bias case, we group the two biases tasks within one sub-workflow to highlight the difference with the other tasks. In this example, it is not necessary to determine the arm of the exposures via a dynamic parameter, because each VIS and NIR type of observation have their own datasource and task.

The workflow layout is shown in Figure 13.1. Note the orange color of the "bias" element, denoting the sub-workflow that contains the two bias tasks (bias_vis and bias_nir). In order to simplify the workflow layout, all the VIS/NIR tasks could be merged into a subworkflow. In the next example, we show how to create a single task per type (e.g., flat) that serves both arms.

demo2_wkf.py

```
1 from edps import task, SCIENCE
2 from .demo2_datasources import *
3 from .demo2_subworkflow import bias_arm
4
5
```

```
6
7  #--- Processing tasks ----------------------------------------------------------------
8
9  # Tasks for processing the biases
10 #Instead of creating 2 tasks for different types of biases,
11 #I "delegate" the task creation to a subworkflow
12 #exploiting the fact that the 2 tasks have the same "structure"
13 bias_vis_task = bias_arm(raw_bias_vis, 'vis')
14 bias_nir_task = bias_arm(raw_bias_nir, 'nir')
15
16 #- Tasks for processing the flat fields
17 #Here it is shown how two tasks are created without using a subworkflow
18 flat_vis_task = (task('flat_vis')
19                  .with_recipe('run_flat')
20                  .with_main_input(raw_flat_vis)
21                  .with_associated_input(bias_vis_task, [MASTERBIAS_VIS])
22                  .with_input_filter(MASTERBIAS_VIS)
23                  .build())
24
25 flat_nir_task = (task('flat_nir')
26                  .with_recipe('run_flat')
27                  .with_main_input(raw_flat_nir)
28                  .with_associated_input(bias_nir_task, [MASTERBIAS_NIR])
29                  .with_input_filter(MASTERBIAS_NIR)
30                  .build())
31
32 #- Tasks for science processing
33 science_vis_task = (task('science_vis')
34                     .with_recipe('run_science')
35                     .with_meta_targets([SCIENCE])
36                     .with_main_input(raw_science_vis)
37                     .with_associated_input(raw_sky_vis, min_ret=0)  # sky is optional
38                     .with_associated_input(bias_vis_task, [MASTERBIAS_VIS])
39                     .with_associated_input(flat_vis_task, [MASTERFLAT_VIS])
40                     .build())
41
42 science_nir_task = (task('science_nir')
43                     .with_recipe('run_science')
44                     .with_main_input(raw_science_nir)
45                     .with_meta_targets([SCIENCE])
46                     .with_associated_input(raw_sky_nir, min_ret=0)  # sky is optional
47                     .with_associated_input(bias_nir_task, [MASTERBIAS_NIR])
48                     .with_associated_input(flat_nir_task, [MASTERFLAT_NIR])
49                     .with_associated_input(static_catalog)
50                     .build())
```

demo2_subworkflow.py

```
1 from edps import subworkflow, task
2 from .demo2_datasources import *
3
4 @subworkflow("bias","")
5
6 def bias_arm(raw_bias,tag):
7     bias_task = (task('bias_'+tag)
8                  .with_recipe('run_bias')
```

```
                 .with_main_input(raw_bias)
                 .build())
    return bias_task
```

demo2_datasources.py

```
from edps import data_source, RelativeTimeRange
from .demo2_classification import *
from edps.generator.time_range import UNLIMITED, ONE_DAY, ONE_WEEK, SAME_NIGHT
from . import demo2_keywords as kwd

# Convention for Data sources Association rule levels:
# Each data source can have several match function which correspond to different
# quality levels for the selected data. The level is specified as a number that
# follows this convention:
#   level < 0: more restrictive than the calibration plan
#   level = 0 follows the calibration plan
#   level = 1 quality sufficient for QC1 certification
#   level = 2 probably still acceptable quality
#   level = 3 significant risk of bad quality results

# standard matching keywords:

setup1=[kwd.tpl_start, kwd.arm, kwd.ins1_slit]
setup2=[kwd.tpl_start, kwd.arm, kwd.ins2_slit]

raw_bias_vis = (data_source('RAW_BIAS_VIS')
            .with_classification_rule(bias_vis_class)
            .with_min_group_size(3)
            .with_setup_keywords(setup1)
            .with_grouping_keywords(setup1)
            .with_match_keywords([kwd.instrume,kwd.arm, kwd.ins1_slit],
                                 time_range=ONE_DAY, level=0)
            .with_match_keywords([kwd.instrume, kwd.arm, kwd.ins1_slit],
                                 time_range=ONE_WEEK, level=1)
            .with_match_keywords([kwd.instrume, kwd.arm, kwd.ins1_slit],
                                 time_range=UNLIMITED, level=3)
            .build())
# Bias nir
raw_bias_nir = (data_source('RAW_BIAS_NIR')
            .with_classification_rule(bias_nir_class)
            .with_min_group_size(3)
            .with_setup_keywords(setup2)
            .with_grouping_keywords(setup2)
            .with_match_keywords([kwd.instrume,kwd.arm, kwd.ins2_slit],
                                 time_range=ONE_DAY, level=0)
            .with_match_keywords([kwd.instrume, kwd.arm, kwd.ins2_slit],
                                 time_range=ONE_WEEK, level=1)
            .with_match_keywords([kwd.instrume, kwd.arm, kwd.ins2_slit],
                                 time_range=UNLIMITED, level=3)
            .build())

static_catalog=(data_source('catalog')
                .with_classification_rule(static_catalog_class)
                .with_match_keywords(['FILTER']).build())

```

```
51 raw_flat_vis = (data_source('RAW_FLAT_VIS')
52             .with_classification_rule(flat_vis_class)
53             .with_min_group_size(3)
54             .with_setup_keywords(setup1)
55             .with_grouping_keywords(setup1)
56             .with_match_keywords([kwd.instrume, kwd.arm, kwd.ins1_slit],
57                                  time_range=ONE_DAY, level=0)
58             .with_match_keywords([kwd.instrume, kwd.arm, kwd.ins1_slit],
59                                  time_range=ONE_WEEK, level=1)
60             .with_match_keywords([kwd.instrume, kwd.arm, kwd.ins1_slit],
61                                  time_range=RelativeTimeRange(-365,365), level=3)
62             .build())
63
64 raw_flat_nir = (data_source('RAW_FLAT_NIR')
65             .with_classification_rule(flat_nir_class)
66             .with_min_group_size(3)
67             .with_setup_keywords(setup2)
68             .with_grouping_keywords(setup2)
69             .with_match_keywords([kwd.instrume, kwd.arm, kwd.ins2_slit],
70                                  time_range=ONE_DAY, level=0)
71             .with_match_keywords([kwd.instrume, kwd.arm, kwd.ins2_slit],
72                                  time_range=ONE_WEEK, level=1)
73             .with_match_keywords([kwd.instrume, kwd.arm, kwd.ins2_slit],
74                                  time_range=RelativeTimeRange(-365,365), level=3)
75             .build())
76
77 raw_science_vis = (data_source('RAW_SCIENCE_VIS')
78                .with_classification_rule(science_vis_class)
79                .with_grouping_keywords(['mjd-obs'])
80                .build())
81
82 raw_science_nir = (data_source('RAW_SCIENCE_NIR')
83                .with_classification_rule(science_nir_class)
84                .with_grouping_keywords(['mjd-obs'])
85                .build())
86
87
88
89 raw_sky_vis = (data_source('RAW_SKY_VIS')
90            .with_classification_rule(sky_vis_class)
91            .with_grouping_keywords(['mjd-obs'])
92            .with_match_keywords([kwd.instrume, kwd.arm,
93                                  kwd.ins1_slit, kwd.tpl_start], level=0)
94            .with_match_keywords([kwd.instrume, kwd.arm, kwd.ins1_slit],
95                                 time_range=SAME_NIGHT, level=2)
96            .build())
97
98 raw_sky_nir = (data_source('RAW_SKY_NIR')
99            .with_classification_rule(sky_nir_class)
100           .with_grouping_keywords(['mjd-obs'])
101           .with_match_keywords([kwd.instrume, kwd.arm, kwd.ins2_slit,
102                                 kwd.tpl_start], level=0)
103           .with_match_keywords([kwd.instrume, kwd.arm, kwd.ins2_slit],
104                                time_range=SAME_NIGHT, level=2)
105           .build())
```

demo2_classification.py

```
from edps import classification_rule
from . import demo2_keywords as kwd


#Dictionaries with general keywords
demo = {kwd.instrume: "DEMO"}
vis = {**demo, kwd.arm:"VIS"}
nir = {**demo, kwd.arm:"NIR"}

# RAW FILES
bias_vis_class = classification_rule('BIAS_VIS', {**vis, kwd.dpr_type: "BIAS"})
bias_nir_class = classification_rule('BIAS_NIR', {**nir, kwd.dpr_type: "BIAS"})

flat_vis_class = classification_rule('FLAT_VIS', {**vis, kwd.dpr_type: "FLAT"})
flat_nir_class = classification_rule('FLAT_NIR', {**nir, kwd.dpr_type: "FLAT"})

science_vis_class = classification_rule('SCIENCE_VIS', {**vis, kwd.dpr_type: "SCIENCE"})
science_nir_class = classification_rule('SCIENCE_NIR', {**nir, kwd.dpr_type: "SCIENCE"})

sky_vis_class = classification_rule('SKY', {**vis, kwd.dpr_type: "SKY"})
sky_nir_class = classification_rule('SKY', {**nir, kwd.dpr_type: "SKY"})

# MASTER CALIBRATIONS
MASTERBIAS_VIS = classification_rule('MASTERBIAS_VIS', {**demo, kwd.pro_catg: '
    MASTERBIAS_VIS'})
MASTERBIAS_NIR = classification_rule('MASTERBIAS_NIR', {**demo, kwd.pro_catg: '
    MASTERBIAS_VIS'})
MASTERFLAT_VIS = classification_rule('MASTERFLAT_VIS', {**demo, kwd.pro_catg: '
    MASTERBIAS_VIS'})
MASTERFLAT_NIR = classification_rule('MASTERFLAT_NIR', {**demo, kwd.pro_catg: '
    MASTERBIAS_VIS'})

# STATIC CALIBRATIONS
static_catalog_class = classification_rule('CATALOG', {**demo, kwd.pro_catg:'CATALOG'})
```

demo2_keywords.py

```
# HEADER KEYWORDS USED FOR CLASSIFICATION, GROUPING, AND ASSOCIATION.
instrume = "instrume"
arm = "ARM"
ins1_slit = "ins1.slit"
ins2_slit = "ins2.slit"
tpl_start = "tpl.start"
mjd_obs = "mjd-obs"
dpr_type = "dpr.type"
pro_catg = "pro.catg"
```

13.3 Simplified workflow.

This workflow represents a "simplification" of the pipeline and instrument description. Similar classification rules (e.g. bias, flats) are combined into a single data source, which is processed by a

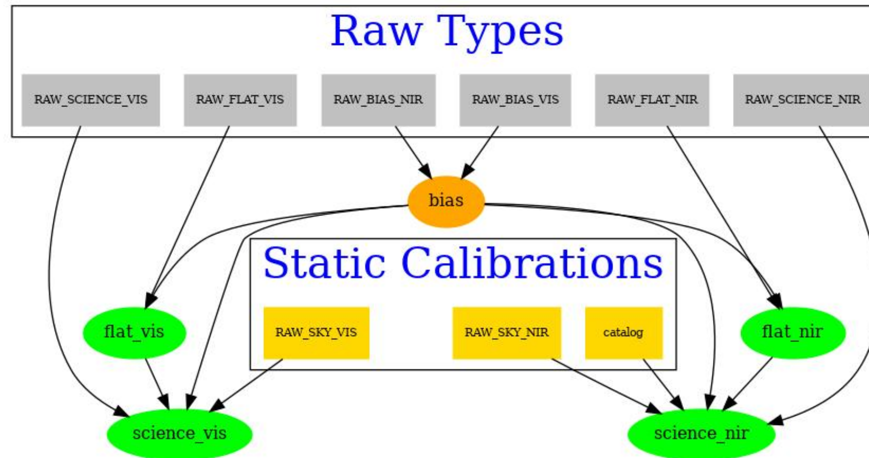

Figure 13.1: The "demo2_wkf" workflow.

single task, regardless o the file tag. Correct file association is carried on by alternative and conditional associations. The workflow layout is shown in Figure 13.2. Note that each box is a task, not a subworkflow with two tasks (one per ARM) within it. It is the same layout as in Fig 3.1, despite the pipeline is different and differentiate between VIS and NIR categories. Note that this solution is possible because both VIS and NIR observations are processed with the same recipes.

demo3_wkf.py

```
from edps import task, SCIENCE, alternative_association
from .demo3_datasources import *
from .demo3_task_functions import *
from .demo3_rules import *


bias_task = (task("bias")
             .with_recipe("run_bias")
             .with_main_input(raw_bias)
             .build())

flat_task = (task("flat")
             .with_recipe("run_flat")
             .with_main_input(raw_flat)
             .with_associated_input(bias_task, [MASTERBIAS_VIS], condition=is_input_VIS,
                                    match_rules=attach_bias_and_flat_vis)
             .with_associated_input(bias_task, [MASTERBIAS_NIR], condition=is_input_NIR,
                                    match_rules=attach_bias_and_flat_nir)
             .with_dynamic_parameter("arm_used", which_observation_type)
             .build())

alternative_sky= (alternative_association()
              .with_associated_input(raw_sky, min_ret=0,
                                     condition=is_input_VIS, match_rules=attach_sky_vis)
              .with_associated_input(raw_sky, min_ret=0,
                                     condition=is_input_NIR, match_rules=attach_sky_nir))
```

```
27
28 science_task = (task("science")
29                .with_recipe("run_science")
30                .with_main_input(raw_science)
31                .with_alternative_associated_inputs(alternative_sky)
32                .with_associated_input(bias_task, [MASTERBIAS_VIS], condition=is_input_VIS,
33                                       match_rules=attach_bias_and_flat_vis)
34                .with_associated_input(bias_task, [MASTERBIAS_NIR], condition=is_input_NIR,
35                                       match_rules=attach_bias_and_flat_nir)
36                .with_associated_input(flat_task, [MASTERFLAT_VIS], condition=is_input_VIS,
37                                       match_rules=attach_bias_and_flat_vis)
38                .with_associated_input(flat_task, [MASTERFLAT_NIR], condition=is_input_NIR,
39                                       match_rules=attach_bias_and_flat_nir)
40                .with_associated_input(static_catalog)
41                .with_dynamic_parameter("arm_used", which_observation_type)
42                .with_meta_targets([SCIENCE])
43                .build())
```

demo3_datasources.py

```
1 from edps import data_source
2 from .demo3_classification import *
3 from . import demo2_keywords as kwd
4
5
6 setup=[kwd.tpl_start, kwd.arm, kwd.ins1_slit, kwd.ins2_slit]
7 #Grouping by tpl_start is sufficient under the assumption that
8 # I do not have both VIS and NIR observations within the same
9 # observing template.
10
11 grouping=[kwd.tpl_start]
12
13 #- Raw datasources
14 raw_bias = (data_source('RAW_BIAS')
15             .with_classification_rule(bias_vis_class)
16             .with_classification_rule(bias_nir_class)
17             .with_min_group_size(3)
18             .with_setup_keywords(setup)
19             .with_grouping_keywords(grouping)
20             .build())
21
22
23
24 raw_flat = (data_source('RAW_FLAT')
25             .with_classification_rule(flat_vis_class)
26             .with_classification_rule(flat_nir_class)
27             .with_min_group_size(3)
28             .with_setup_keywords(setup)
29             .build())
30
31 raw_science = (data_source('RAW_SCIENCE')
32                .with_classification_rule(science_vis_class)
33                .with_classification_rule(science_nir_class)
34                .with_grouping_keywords(['mjd-obs'])
35                .build())
36
```

```
raw_science_vis = (data_source('RAW_SCIENCE_VIS')
                   .with_classification_rule(science_vis_class)
                   .with_grouping_keywords(['mjd-obs'])
                   .build())

raw_science_nir = (data_source('RAW_SCIENCE_NIR')
                   .with_classification_rule(science_nir_class)
                   .with_grouping_keywords(['mjd-obs'])
                   .build())


raw_sky = (data_source('RAW_SKY')
              .with_classification_rule(sky_vis_class)
              .with_classification_rule(sky_nir_class)
              .with_grouping_keywords(['mjd-obs'])
              .build())


static_catalog=(data_source('catalog')
                    .with_classification_rule(static_catalog_class)
                    .with_match_keywords([kwd.arm]).build())
```

demo3_task_functions.py

```
from edps import get_parameter, JobParameters, ClassifiedFitsFile, List

# These functions returns TRUE/FALSE depending on the value
# of the parameter "arm_used"
def is_input_NIR(params: JobParameters) -> bool:
        return get_parameter(params, "arm_used") == 'NIR'

def is_input_VIS(params: JobParameters) -> bool:
        return get_parameter(params, "arm_used") == 'VIS'

#This function determines the value of the parameter arm_used,
#depending on the value of the header keywor "ARM"
def which_observation_type(files: List[ClassifiedFitsFile]):
    # Note: files are only the main input files, not the associated files
    arm = files[0].get_keyword_value('ARM', None)
    return "NIR" if arm == "NIR" else "VIS" if arm == "VIS" else None
```

demo3_classification.py

```
from edps import classification_rule
from . import demo3_keywords as kwd
#Dictionaries with general keywords
demo = {kwd.instrume: "DEMO"}
vis = {**demo, kwd.arm:"VIS"}
nir = {**demo, kwd.arm:"NIR"}

# RAW FILES
bias_vis_class = classification_rule('BIAS_VIS', {**vis, kwd.dpr_type: "BIAS"})
bias_nir_class = classification_rule('BIAS_NIR', {**nir, kwd.dpr_type: "BIAS"})

flat_vis_class = classification_rule('FLAT_VIS', {**vis, kwd.dpr_type: "FLAT"})
flat_nir_class = classification_rule('FLAT_NIR', {**nir, kwd.dpr_type: "FLAT"})
```

```
14
15 science_vis_class = classification_rule('SCIENCE_VIS', {**vis, kwd.dpr_type: "SCIENCE"})
16 science_nir_class = classification_rule('SCIENCE_NIR', {**nir, kwd.dpr_type: "SCIENCE"})
17
18 sky_vis_class = classification_rule('SKY', {**vis, kwd.dpr_type: "SKY"})
19 sky_nir_class = classification_rule('SKY', {**nir, kwd.dpr_type: "SKY"})
20
21 # MASTER CALIBRATIONS
22 MASTERBIAS_VIS = classification_rule('MASTERBIAS_VIS', {**demo, kwd.pro_catg: '
       MASTERBIAS_VIS'})
23 MASTERBIAS_NIR = classification_rule('MASTERBIAS_NIR', {**demo, kwd.pro_catg: '
       MASTERBIAS_VIS'})
24 MASTERFLAT_VIS = classification_rule('MASTERFLAT_VIS', {**demo, kwd.pro_catg: '
       MASTERBIAS_VIS'})
25 MASTERFLAT_NIR = classification_rule('MASTERFLAT_NIR', {**demo, kwd.pro_catg: '
       MASTERBIAS_VIS'})
26
27 # STATIC CALIBRATIONS
28 static_catalog_class = classification_rule('CATALOG', {**demo, kwd.pro_catg:'CATALOG'})
```

demo3_keywords.py

```
1 # HEADER KEYWORDS USED FOR CLASSIFICATION, GROUPING, AND ASSOCIATION.
2 instrume = "instrume"
3 arm = "ARM"
4 ins1_slit = "ins1.slit"
5 ins2_slit = "ins2.slit"
6 tpl_start = "tpl.start"
7 mjd_obs = "mjd-obs"
8 dpr_type = "dpr.type"
9 pro_catg = "pro.catg"
```

demo3_rules.py

```
1 from edps import match_rules, ONE_DAY, ONE_WEEK, UNLIMITED, SAME_NIGHT
2 from . import demo3_keywords as kwd
3
4 # CLASSIFICATION RULES
5 # none
6
7 # ASSOCIATION RULES
8 #  -  first, e.g.  ref=trigger (e.g. science)
9 #  -  second, e.g.  f=file to associate (e.g. calibration)
10 # none
11
12 # ASSOCIATION RULES THAT OVERRIDES THOSE IN SPECIFIED IN THE DATA_SOURCE
13
14 setup_vis=[kwd.instrume, kwd.arm, kwd.ins1_slit]
15 setup_nir=[kwd.instrume, kwd.arm, kwd.ins2_slit]
16
17 #Definition of two different matching rules, one for VIS (needs to look ins1.slit) and
18 #one for NIR (needs to look for ins2.slit).
19
20 attach_bias_and_flat_vis = (match_rules()
21                             .with_match_keywords(setup_vis, time_range=ONE_DAY, level=0)
22                             .with_match_keywords(setup_vis, time_range=ONE_WEEK, level=1)
```

```
23                             .with_match_keywords(setup_vis, time_range=UNLIMITED, level=3))
24
25 attach_bias_and_flat_nir = (match_rules()
26                             .with_match_keywords(setup_nir, time_range=ONE_DAY, level=0)
27                             .with_match_keywords(setup_nir, time_range=ONE_WEEK, level=1)
28                             .with_match_keywords(setup_nir, time_range=UNLIMITED, level=3))
29
30 attach_sky_vis = (match_rules()
31              .with_match_keywords(setup_vis+[kwd.tpl_start], level=0)
32              .with_match_keywords(setup_vis, time_range=SAME_NIGHT, level=1))
33
34
35 attach_sky_nir = (match_rules()
36              .with_match_keywords(setup_nir+[kwd.tpl_start], level=0)
37              .with_match_keywords(setup_nir, time_range=SAME_NIGHT, level=1))
```

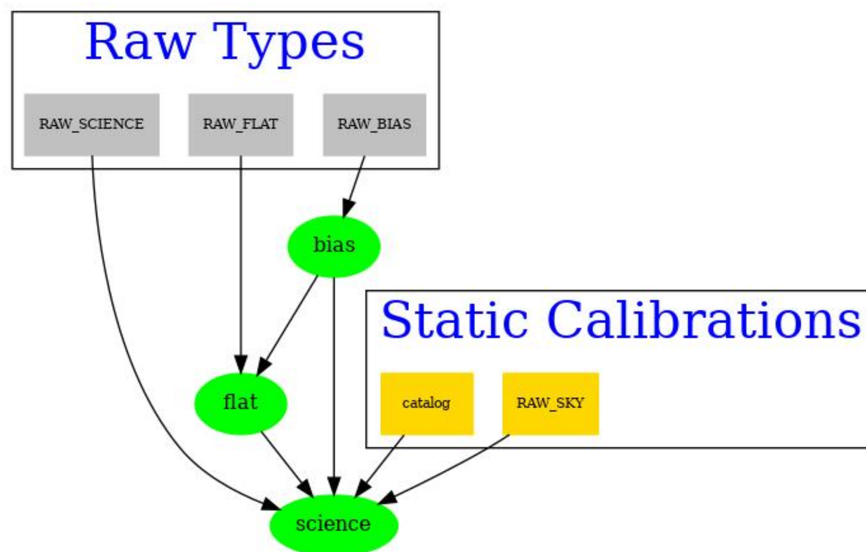

Figure 13.2: The "demo3_wkf" workflow.

Part V

How to run and debug workflows

14 Tips for debugging.

Debugging a workflow is as complicated as the data reduction cascades, classification and association rules te workflow itself has to support. Here, we collect some tips that could help during development and debugging. A tutorial on how to install and run EDPS is available at `https://www.eso.org/sci/software/edps.html`.

- Start simple! First get a clear design of the various steps, i.e. the sequence of tasks and their inputs. A simple structure like the one shown in Section 3.2 is the starting point.
- Create a plot of the workflow design and compare it to the requirements of the data reduction cascade(s). See Section 5.1 of the EDPS tutorial.
- Add classification and association rules only after the general design is ready.
- Advanced features such as conditional associations can be added after the overall design and classification are ready.
- Prepare a pool of data (fits file) where the file association is know. Do the data classification and organisation with EDPS on those data and check if they match the expectation. Header-only fits files are sufficient for this purpose. Use the option `-f` to do a tree-like organisation of files without starting the reduction (see Section 5.4 of the EDPS tutorial).
- Test the workflow by starting from the first task in the reduction cascade, not by running the full reduction chain.
- If a job is found to be incomplete and there is not clear indication about what is missing, edit the task and add one associated input at the time. This will help to identify what's missing.

15 EDPS integrated test suite: `json` tests.

'json_tests.py' contains generic logic which executes all test cases defined under 'tests/json_configuration' directory. Adding a new test requires putting new scenario description in one of existing '.json' files or creating a new file. New file will be picked-up automatically on the next test run.

15.1 Test suite JSON file

Each '.json' files contains a document with one field 'scenarios', which contains a list of scenario definitions. Those will be run in-order, sequentially, exactly as they are defined in the file.

15.2 Scenario definition

Each scenario has 3 major sections:

- Test-case metadata
- Input file definitions
- Result expectations

15.2.1 Test-case metadata

First section contains:

- 'description' of the test, which will be used as 'test name' in the execution result
- 'workflow' to be used for data organization
- 'workflow_parameters' optional dictionary to pass with the request to EDPS
- 'workflow_parameter_set' optional name of a 'named parameter set' for EDPS to use
- 'targets' list of target tasks to consider when generating jobs (EDPS will generate jobs for the targets and also for anything those targets depend on)
- 'meta_targets' list of labelled meta-targets which will be expanded by EDPS into a list of tasks to be used as targets

Example:

```
{
  "description": "fors bias flat",
```

```
  "workflow": "fors.fors_imaging_wkf",
  "workflow_parameters": {
    "a": "b"
  },
  "workflow_parameter_set": "qc0_parameters",
  "targets": [
    "bias",
    "flat"
  ],
  "meta_targets": [],
  "skip": false
}
```

15.2.2 Input file definitions

Each test scenario will run EDPS using generated FITS files. List ‘input_files‘ holds definitions of file templates. Template consist of:

- ‘name_prefix‘ which will be prepended to generated files (each file will have a random UUID suffix)
- ‘count‘ number of files to generate, defaults to 1
- ‘keywords‘ dictionary with keywords to place in primary header of the FITS file. Keywords defined like that will be put in the file as-is.

Example:

```
{
  "input_files": [
    {
      "name_prefix": "bias",
      "count": 4,
      "keywords": {
        "instrume": "FORS1",
        "dpr.catg": "CALIB",
        "dpr.type": "BIAS"
      }
    },
    {
      "name_prefix": "flat",
      "keywords": {
        "instrume": "FORS1",
        "dpr.catg": "CALIB",
```

```
        "dpr.type": "FLAT,SKY",
        "dpr.tech": "IMAGE"
      }
    }
  ]
}
```

15.2.3 Result expectations

Each test scenario will be validated against the defined expected results list. List ‘results‘ contains definitions of the jobs that are expected to be created by EDPS.

Each job is defined by:

- ‘recipe‘ name of the recipe which is supposed to be used
- ‘inputs_prefixes‘ list of allowed filename prefixes for the input files

Example:

```
{
  "results": [
    {
      "recipe": "fors_bias",
      "inputs_prefixes": [
        "bias"
      ]
    },
    {
      "recipe": "fors_img_sky_flat",
      "inputs_prefixes": [
        "flat"
      ]
    }
  ]
}
```

15.2.4 Full example

Example of a single scenario:

```
    {
```

```
"description": "fors bias flat",
"workflow": "fors.fors_imaging_wkf",
"workflow_parameters": {
  "a": "b"
},
"workflow_parameter_set": "qc0_parameters",
"targets": [
  "bias",
  "flat"
],
"meta_targets": [],
"input_files": [
  {
    "name_prefix": "bias",
    "count": 4,
    "keywords": {
      "instrume": "FORS1",
      "dpr.catg": "CALIB",
      "dpr.type": "BIAS"
    }
  },
  {
    "name_prefix": "flat",
    "keywords": {
      "instrume": "FORS1",
      "dpr.catg": "CALIB",
      "dpr.type": "FLAT,SKY",
      "dpr.tech": "IMAGE"
    }
  }
],
"results": [
  {
    "recipe": "fors_bias",
    "inputs_prefixes": [
      "bias"
    ]
  },
  {
    "recipe": "fors_img_sky_flat",
    "inputs_prefixes": [
      "flat"
    ]
  }
]
```

```
  }
```

15.3 Default behaviour

15.3.1 MJD-OBS

For each scenario a random ‘base mjd-obs‘ is generated. Unless MJD-OBS keyword is explicitly defined for given template, the ‘base‘ value will be used. If there is more than 1 file in the template, each consecutive file has the MJD-OBS slightly further in the future compared to previous one -> ‘base_mjd_obs + i * 0.02‘. If keyword is explicitly defined for template it will be used as-is, without the increment. Each consecutive template starts with mjd-obs further back in time, based on the order in which inputs are defined in the file.

15.3.2 TPL.START

If not explicitly defined ‘tpl.start‘ is set to a randomly generated value, same for each file of the template. Input files template definition supports only a single set of keywords, so in case different files should be marked as part of the same template it might be necessary to explicitly set ‘tpl.start‘.

Example:

```
{
  "input_files": [
    {
      "name_prefix": "orderdef_a",
      "count": 1,
      "keywords": {
        "instrume": "ESPRESSO",
        "dpr.catg": "CALIB",
        "dpr.type": "ORDERDEF,LAMP,OFF",
        "tpl.start": "1"
      }
    },
    {
      "name_prefix": "orderdef_b",
      "count": 1,
      "keywords": {
        "instrume": "ESPRESSO",
        "dpr.catg": "CALIB",
        "dpr.type": "ORDERDEF,OFF,LAMP",
        "tpl.start": "1"
      }
```

```
        }
    ]
}
```

With such definition both generated files will have the same ‘tpl.start‘.

15.3.3 Default keywords

Certain keywords are inserted automatically, even if not explicitly defined:

- ‘arcfile‘ set to the same as file name: `‘{prefix}_{i + 1}_{uuid.uuid4()}.fits‘`
- ‘tpl.nexp‘ set to number of template files
- ‘tpl.expno‘ set to numbers ‘1..n‘ for each template file

15.4 Known limitations

- It’s not possible to re-run one selected test, because they are generated dynamically. If you want to work on a single test only set ‘skip‘ flag for other tests.
- Synthetic data generation has no knowledge about type of the data or any real-world relative order in which such data are taken. Unless you explicitly specify MJD-OBS keyword you should not make any assumptions about the MJD-OBS value and therefore about chronological ordering of the generated files.
- Each template definition can have only one set of keywords to use.
- Tests are doing only the ‘data organization‘ part and are designed to verify workflow against expectations about what jobs should be created for given set of inputs. No recipes are run, so it’s still possible that the workflow is not really correct (eg. min/max-ret is set incorrectly or some task is not declaring association necessary for the recipe).
- Result verification does not check if all defined prefixes are included in the list of input files for the recipe (eg. if there is at least one file with each prefix), it checks only that there are no input files other than those with right prefix (eg. if the only expected prefix is ‘bias‘ and there is a file ‘flat_...‘ as input the test will fail)
- Result verification is ‘strict‘ and it requires that the number of resulting jobs matches the number of defined expectations and that there is at least one job matching each of the expectations.
- Tests are able to check only happy-paths, they always assert that request succeeded, so are not suitable for checking error conditions.

15.5 To ease Json implementation and verification

Different tasks often share common inputs, moreover association rules may share common header keywords. In particular this occurs when the full reduction chain is tested.

For this reason the EDPS library offers special ways to share common information. For example:

```
{

 "keywords": {
    "keys_instrume_setup": {
      "instrume": "HAWKI",
      "det.ncorrs.name": "A",
      "ins.filt1.name": "B",
      "ins.filt2.name": "C"
    },
    "keys_detector": {
      "det.dit": 1,
      "det.ndit": 1,
      "det.rspeed": 1
    }
  },

"inputs": [
   {
      "name_prefix": "dark",
      "count": 5,
      "common_keywords": [
        "keys_detector"
      ],
      "keywords": {
        "instrume": "HAWKI",

        "dpr.catg": "CALIB",
        "dpr.type": "DARK",
        "dpr.tech": "IMAGE",
        "tpl.id": "HAWKI_img_cal_Darks",
        "tpl.nexp": 3,
        "tpl.start": 1,
        "obs.start": "2010-11-22T05:16:50",
        "arcfile": "HAWKI.010-11-22T05:16:50.fits"
      }
    },
    {
      "name_prefix": "reference_dark",
```

```
    "count": 1,
    "common_keywords": [
      "keys_detector"
    ],
    "keywords": {
      "instrume": "HAWKI",
      "dpr.catg": "CALIB",
      "dpr.type": "DARK",
      "dpr.tech": "IMAGE",
      "pro.catg": "REFERENCE_DARK",
      "tpl.id": "HAWKI_img_cal_Darks",
      "tpl.nexp": 3,
      "tpl.start": 1,
      "obs.start": "2010-11-22T05:16:50"
    }
  },
  {
    "name_prefix": "flat_twilight",
    "count": 10,
    "common_keywords": [
      "keys_instrume_setup",
      "keys_detector"
    ],
    "keywords": {
      "instrume": "HAWKI",
      "dpr.catg": "CALIB",
      "dpr.type": "FLAT",
      "dpr.tech": "IMAGE",
      "tpl.id": "HAWKI_img_cal_TwFlats",
      "tpl.start": "2010-11-22T05:15:50"
    }
  },

  {
    "name_prefix": "master_bpm",
    "common_keywords": [
      "keys_instrume_setup",
      "keys_detector"
    ],
    "keywords": {
      "instrume": "HAWKI",
      "pro.catg": "MASTER_BPM"
    }
  },
  {
```

```
      "name_prefix": "master_conf",
      "common_keywords": [
        "keys_instrume_setup",
        "keys_detector"
      ],
      "keywords": {
        "instrume": "HAWKI",
        "pro.catg": "MASTER_CONF"
      }
    },
     {
      "name_prefix": "master_dark",
      "common_keywords": [
        "keys_detector"
      ],
      "keywords": {
        "instrume": "HAWKI",
        "pro.catg": "MASTER_DARK"
      }
    },
    {
      "name_prefix": "reference_twilight_flat",
      "common_keywords": [
        "keys_instrume_setup",
        "keys_detector"
      ],
      "keywords": {
        "instrume": "HAWKI",
        "pro.catg": "REFERENCE_TWILIGHT_FLAT"
      }
    },
  ],
  "scenarios": [
      {
      "skip": false,
      "description": "HAWKI master dark test",
      "workflow": "hawki.hawki_wkf",
      "targets": [
        "dark"
      ],
      "meta_targets": [],
      "common_inputs": [
        "dark",
        "reference_dark",
        "master_bpm",
```

```
      "master_conf"
    ],
    "results": [
      {
        "recipe": "hawki_dark_combine",
        "inputs_prefixes": [
          "dark"
        ],
        "assoc_prefixes": [
          "reference_dark",
          "master_bpm",
          "master_conf"
        ]
      }
    ]
  },
  {
    "skip": false,
    "description": "HAWKI master flat twilight test",
    "workflow": "hawki.hawki_wkf",
    "targets": [
      "flat"
    ],
    "meta_targets": [],
    "common_inputs": [
      "flat_twilight",
      "reference_twilight_flat",
      "master_dark",
      "master_bpm",
      "master_conf"
    ],
    "results": [
      {
        "recipe": "hawki_twilight_flat_combine",
        "inputs_prefixes": [
          "flat_twilight"
        ],
        "assoc_prefixes": [
          "reference_twilight_flat",
          "master_dark",
          "master_bpm",
          "master_conf"
        ]
      }
    ]
```

```
    },
```

As one can note, in the previous example, the the set of FITS keywords defined by `keys_instrume_setu` is common to `flat_twilight`, `reference_twilight_flat`, `master_bpm`, `master_conf`, and the set of keywords defined by `keys_detector` is common to these and `dark`, `reference_dark` and `master_dark`. Those can be shared using a syntax like the following:

```
"common_keywords": [
        "keys_name1",
        "keys_name2",
        ...
     ],
```

Moreover inputs like `master_bpm`, `master_conf` are common for the two shown examples and can be shared with a syntax like:

```
"common_inputs": [
        "input_name1",
        "input_name2",
        ...
     ],
```

In this way, once a given common input (or a set of common FITS keywords) has been validated, it is easy to extend the list of unit tests. The inputs and FITS keywords involved in the association rules that are peculiar will have to be explicitly written.