EUROPEAN SOUTHERN OBSERVATORY
Organisation Européenne pour des Recherches Astronomiques dans l'Hémisphère Austral
Europäische Organisation für astronomische Forschung in der südlichen Hemisphäre

Reflex Workflow Development Guide

Author: Cesar Enrique Garcia Dabo (cgarcia@eso.org)
Department: Pipeline Systems Department
Version: 1.0
Date: 01/10/2013

Contents

1 Introduction and scope

`Reflex` is the ESO Recipe Flexible Execution Workbench, an environment to run ESO VLT pipelines which employs a workflow engine (Kepler[1]) to provide a real-time visual representation of a data reduction

[1] https://kepler-project.org

cascade, called a workflow, which can be easily understood by most astronomers.

It is important that workflows present the overall data flow of a pipeline in a way that is intuitive and self- explanatory. Workflows are complex programs and designing them in such a way that they meet these high-level requirements takes significant planing and effort.

This document is a guide for those who want to write a workflow that uses VLT pipelines, which are based in CPL. Although not all the components in Reflex are specific to the VLT pipelines most of them are and what we describe here is based on our experience designing some VLT workflows. Using Reflex for other purposes is very limited.

We will present the basic blocks of a pipeline workflow, a step by step procedure to create a simple workflow and some hints on specific aspects of the workflow development. Finally we show some of the guidelines used by the current VLT workflows and a section to transition from Reflex 1 workflows to Reflex 2.

This guide assumes that you have already some knowledge of Reflex at least from the user point of view. Also, this tutorial does not cover installation issues. It is recommended that you read the Reflex User Manual, the Kepler user manual and maybe some of the pipeline workflow tutorials (currently UVES tutorial and Xshooter tutorial are available).

User support for this software is available by sending enquiries to `usd-help@eso.org`.

2 Overview on the development of a Reflex workflow

Developing a workflow is usually not as easy as dragging and dropping some actors in the canvas. In order to avoid some mistakes, we recommend to follow these guidelines, during the development:

- Workflows are most useful for pipeline modularized with an appropriate granularity. Older monolithic pipelines should be broken up into individual recipes. The modularization of pipeline should be done from an astronomers point of view. This means that independent steps which might need to be redone are contained in a single recipe. Intermediate data products should be useful for monitoring of the progress in data processing and for diagnostic purposes. Recipe parameters should be independent from each other, i.e. situations where the setting of a parameter in one recipe implies a particular value for a parameter in a different recipe have to be avoided.

- Collect all the supported observing modes that the workflow should support. A workflow might support different observing modes or calibration strategies, although sometimes it is recommended to develop different workflows for substantially different observing modes.

  In some cases the calibration strategy has different calibration chains, for instance when a given step is optional (flux calibration is an example). It is important to collect all the possible calibration chains in order to design the workflow with those cases in mind. Therefore, the modes to be supported have to be carefully considered.

- It is recommended to start a workflow with a fairly stable pipeline, at least in terms of interface. This means that the pipeline recipes should have well-defined inputs and outputs frames, with corresponding PRO.CATG tags as used by *esorex*. The recipe chain depends strongly on this interfaces and therefore a valid design for some inputs/outputs might not be valid if these are changed. There is however some flexibility, for instance, adding a product created by a recipe which is not going to be used anywhere else by other recipe doesn't impact the workflow design.

  It is also strongly recommended that the list of recipe parameters are well defined before creating a workflow. If the parameters are changed, once a workflow already exists it may be tedious and time consuming (and it is not recommended) to change the parameters. Also, the order in which the

parameters are defined by the recipe should be kept. See section on how to change the RecipeExecuter if this happens.

- Reflex workflows are driven by the data files, which are sorted and routed based on the file tags. It is therefore important that the file tags uniquely describes the purpose of a file. For example, if science data and calibration data use different kind of sky observations, these sky observations should get different tags.
- Start designing your workflow with the different calibration chains of the previous step in paper. Rather than starting the workflow design with the Reflex tool itself, it is sometimes better to start a design in paper, so that
- Design the interactive points in the workflow. It is usually useful to think about the interactivity points in the workflow. Usually the interactivity allows to iterate on same parameters of a given recipe. Therefore it has some impact in the modularity of the pipeline and/or the parameters that might be exposed to the user by the recipe.
- Create a OCA rules file which mimics the relationships outlined in previous steps. One important feature of Reflex is that the calibration cascade is somehow coded in two places: the workflow connections of the recipes and the OCA rules. For this reason the OCA rules should be compatible with the workflow connections. More information about OCA rules in section 4.2.1.
- Create workflow following the layout of the template workflow.

3 Mini-HOWTO: Creating a simple workflow

In this section we will show how to create a simple working workflow. Take into account that the complexity of workflow creation depends very much on the complexity of the calibration cascade and the interactivity points. In this case we will target a workflow with only two recipes, where one recipe creates a calibration needed by the next recipe and no interactivity points (see 5 for the interactivity part).

Take into account that you can start already from a template workflow rather than writing a workflow from scratch (see section 3.2).

3.1 Step by step

This step by step procedure will create a workflow similar to the template workflow distributed by Reflex (see section 3.2)

1. It is recommended that Reflex is installed using the manual method rather than the `install_reflex` script. See http://www.eso.org/sci/software/reflex/ for details.
2. A CPL-based pipeline has to be created and made it available via esorex. This guide doesn't cover how to create a pipeline or esorex configuration. Please refer to the CPL and esorex documentation guide for more details. For this HOWTO, it is assumed that a pipeline with recipes named `rrrecipe` and `rrrecipe_calib` are visible via esorex.
3. The first thing is to create a OCA-rule file that will be used to classify, group and associate the proper files together.

   The classification part of the OCA rules will look like this:

```
if DPR.CATG like "%SCIENCE%" and DPR.TYPE like "%OBJECT%" then
{
  REFLEX.CATG = "RRRECIPE_DOCATG_RAW";
  REFLEX.TARGET = "T";
}

if DPR.CATG like "%CALIB%" and DPR.TECH like "%IMAGE%"
   and DPR.TYPE like "%STD%" then
{
  REFLEX.CATG = "RRRECIPE_CALIB_DOCATG_RAW";
}
```

This basically specifies that based on some keywords of the main header of the files, the files should be assigned some classification keywords. Reflex will use `REFLEX.CATG` mainly.

Next, with all the files that have been classified with the same keywords, there is the need to group them and trigger a specific action. This part will look like this:

```
select execute(CALIB_IMG) from inputFiles
   where REFLEX.CATG == "RRRECIPE_CALIB_DOCATG_RAW"
  group by INS.FILT1.NAME, OBS.ID, OBS.NAME,
  OBS.TARG.NAME, TPL.START as (TPL_A,tpl);
select execute(COMBINE_IMG) from inputFiles
  where REFLEX.CATG == "RRRECIPE_DOCATG_RAW"
  group by INS.FILT1.NAME, OBS.ID, OBS.NAME,
  OBS.TARG.NAME, TPL.START as (TPL_A,tpl);
```

The next thing is to define the actions, and within the actions, the calibrations or dependencies needed by each of the actions:

```
action CALIB_IMG
{
minRet = 0; maxRet = 1;
select file as STATIC_MASK from calibFiles
where REFLEX.CATG == "STATIC_MASK";
minRet = 0; maxRet = 1;
select file as IMG_STD_CATALOG from calibFiles
where REFLEX.CATG == "IMG_STD_CATALOG";

recipe rrrecipe_calib;
product IMG_CALIBRATED { REFLEX.CATG = "IMG_CALIBRATED";
        PRO.CATG = "IMG_CALIBRATED"; PRO.EXT="tpl_0000.fits";}
}

action COMBINE_IMG
{
minRet = 0; maxRet = 1;
select file as STATIC_MASK from calibFiles
```

```
where REFLEX.CATG == "STATIC_MASK";
minRet = 1; maxRet = 1;
select file as IMG_CALIBRATED from calibFiles
where PRO.CATG == "IMG_CALIBRATED";

recipe rrrecipe;
product IMG_OBJ_COMBINED { PRO.CATG = "IMG_OBJ_COMBINED";
        PRO.EXT="tpl_0001.fits";}
}
```

The `minRet, maxRet` keywords specify constraints on the number of matching files. If at least `minRet` files are not found the dataset will be incomplete.

The `product` clause defines the products created by a given action, in order to associate it later (like the `IMG_CALIBRATED` file in the example).

4. Next thing will be to define some directories that will be used bu the workflow. There are three types of directories:

   (a) Input directories. These directories will be scanned by the DataOrganizer using the OCA rules to define the datasets to process. It usually contains the user data and maybe the specific pipeline calibration files. Use the name is `RAW_DATA_DIR`.

   (b) Working directories. These directories are used internally by the Reflex actors. The user might have to look at them only for debugging purposes. The common names are `BOOKKEEPING_DIR`, `LOGS_DIR` and `TMP_PRODUCTS_DIR`. The latest one is likely to grow very quickly in size.

   (c) Output directory. This directory will contain the final reduced data in a easy to browse directory structure. The common name is `END_PRODUCTS_DIR`.

The way this is created is via variables in Reflex whose value points to the desired directory. Use the `FileParameter` Kepler standard actor to define these variables. To change the name of the variable, right click on the variable and select `Customize Name`. Figure 1 show how the standard directories implementation looks like.

Figure 1: *Standard setup directories.*

Take into account that this is the standard for workflows but you can use other directories setup for your workflows if needed. However, the working directories must be always be there and with those names, since they are used by several standard Reflex actors. See section 4.1 for more information about the standard directories.

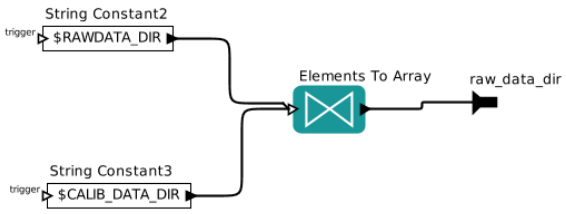

Figure 2: *Directory preparation for the DataOrganizer.*

5. Now we start putting actors which represent the real workflow. The first thing is to combine the calibration directory and the input directory in a single input that will be fed into the data organisation. For that we put two `StringConstant` actors with the values `$RAWDATA_DIR` and `$CALIB_DATA_DIR`. Make sure that the `firingCountLimit` parameter is set to 1 in both cases. Then we connect the output of these two actors to an `ElementsToArray` actor. Figure 2 shows the result of that.

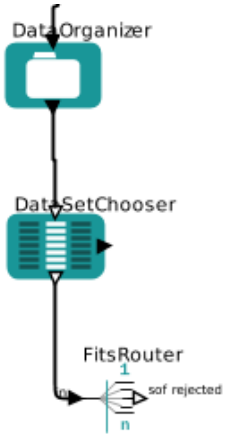

Figure 3: *Connections of the DataOrganizer, DataSetChooser and FitsRouter.*

6. The next thing to do is to connect the output of the array to the input of `DataOrganizer`, i. e., the port `input_data`. The `DataOrganizer` has to be configured with a proper path to the OCA rules. For that, use the parameter `OCA File` of the `DataOrganizer`. The output port of the `DataOrganizer` has to be connected then to the input port `datasets_in` of the `DataSetChooser` and finally the output port `datasets_out` of the `DataSetChooser` has to be connected to the input port `in` of the `FitsRouter`.

After all these connections, the workflow supports the creation of datasets using the definitions of the OCA rules, displaying the datasets for selection and inspection and feeding the selected datasets to the `FitsRouter`.

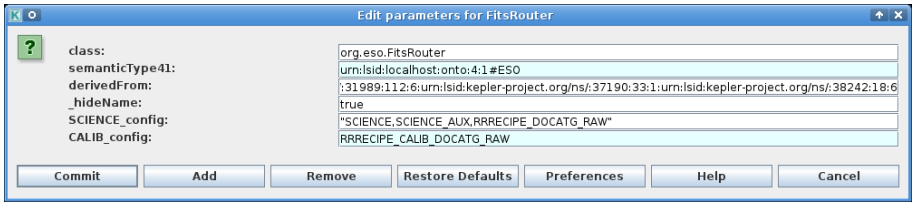

Figure 4: *Configuration of the FitsRouter.*

7. The next thing is to setup the `FitsRouter` to deliver the proper data to different connections. Our simple workflow has two main recipes, one calibration and one science. With this scheme, it would be enough to create two channels: one with all the input needed by the calibration recipe and one with science frames. We first create two ports in the `FitsRouter`: `CALIB` and `SCIENCE`. Then, we create two parameters of the `FitsRouter` using the `Add` button in the `Edit Parameter` window called `CALIB_config` and `SCIENCE_config`. By default, the parameter type is generic, so we will have to quote the list of PRO.CATGs assigned to that port. Other option is to change the parameter type to `StringParameter` and then the quotes are not needed.

   The calibration port will just need one type of files: RRRECIPE_CALIB_DOCATG_RAW. The science port however will redirect several types of files: STATIC_MASK and RRRECIPE_DOCATG_RAW. The result is the configuration shown in figure 4. See section 4.4 for more details.

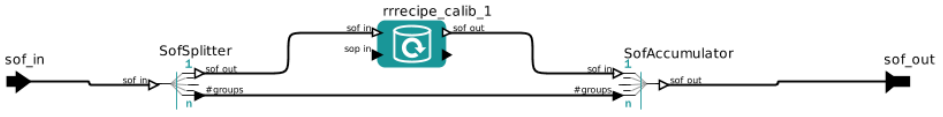

Figure 5: *Composite actor which contains a recipe executer surrounded by a SofSplitter and SofAccumulator. The top level canvas will show this composite actor with the name Calib, as shown in figure 6.*

8. Now it is time to create the recipes which are going to be executed. First, create a composite actor, i. e. a kind of subworkflow, by searching for CompositeActor. Drag it to the canvas and open it (right click `Open Actor`) to start adding more actors in it. We will create an input port and an output port and between them we will put three actors: a `SofSplitter`, a `RecipeExecuter` and a `SofAccumulator`. The first and last can be instantiated easily with the components left menu, however the `RecipeExecuter` should be instantiated using the `Tools -> Instantiate Component` option. This will show a list of available recipes as seen by the esorex command which is in the current path. If your recipe is not shown there, check your esorex and pipeline installation.

   The `sof_out` port of the `SofSplitter` should be connected to the `sof_in` port of the `RecipeExecuter` and the `sof_out` of the latest to the `sof_in` of the `SofAccumulator`. Additionally, the `#groups`

port of both `SofSplitter` and `SofAccumulator` should be connected together. The result is shown in figure 5 for a recipe called `rrrecipe_calib`.

9. Follow the same procedure as before with the science recipe. You can rename the top level composite actors with the names Calib and Science for identification purposes.

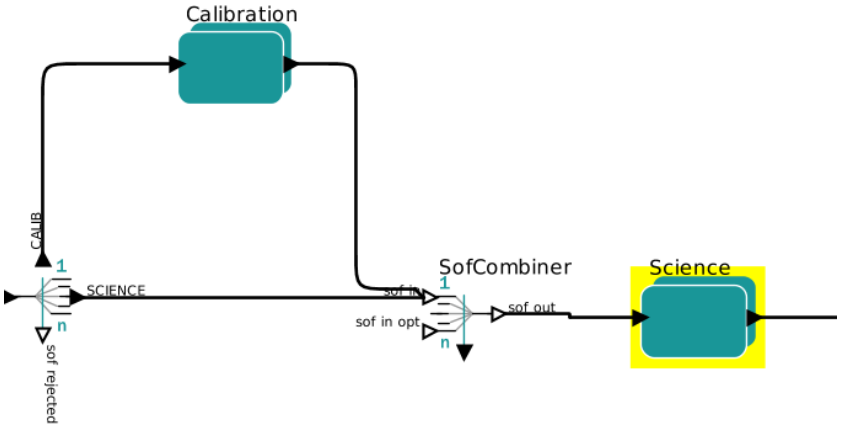

Figure 6: *Connection of the recipe composite actors and the FitsRouter for a simple workflow with one calibration recipe and one science recipe.*

10. Connect the output of the `CALIB` port in the `FitsRouter` to the input of the calibration composite actor.

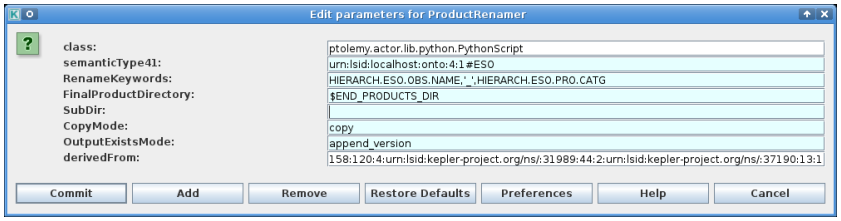

Figure 7: *Configuration of the* `ProductRenamer`.

11. Drag a `SofCombiner` actor in front of the science composite actor and connect both the output of the calibration composite actor and the `SCIENCE` port of the `FitsRouter` to the input of the `SofCombiner`, and the output of the latest to the input of the science composite actor. The result is shown in figure 6.

12. Finally, add a `ProductRenamer` actor to rename the output of the science recipe to some more meaningful names. Set up the `RenameKeywords` parameter of the `ProductRenamer` to some keywords that are representative of the type of data or target. Figure 7 shows a typical configuration.

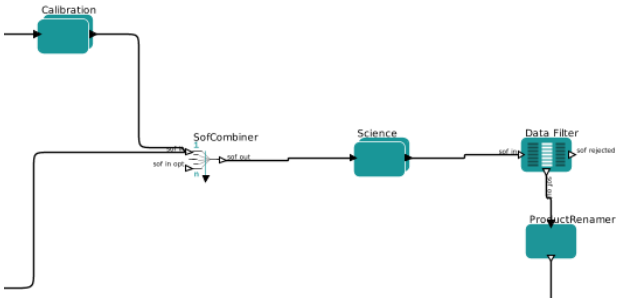

Figure 8: *Layout of the workflow with a* `DataFilter` *and a* `ProductRenamer`.

13. Optionally, a `DataFilter` actor can be put before the `ProductRenamer` to save only a selection of the products created by the science recipe. Figure 8 shows the final layout of this second part of the workflow.

14. Place a DDF director in the top level canvas.

15. Finally, it is recommended to document the workflow in a way that it is self-explanatory, as it is described in section 8.

3.2 The TEMPLATE workflow

When Reflex is installed using the *install_reflex* script as explained in the Reflex User Manual, you will be presented with a list of pipelines to install. One of them is listed as TEMPLATE and contains a basic pipeline with a basic workflow that can be used as a template or an example to build other pipelines/workflows. This workflow uses the pipeline called `iiinstrument` and it can also be retrieved from this link:

`ftp://ftp.eso.org/pub/dfs/pipelines/iiinstrument/.`

Figure 9 shows how this basic workflow looks like. The OCA rules delivered with this workflow are also very basic and just contain one level of dependency: a science recipe just needs the result of a calibration recipe.

The steps followed in the previous section will create a simplified version of this template workflow.

4 Main Components of a Reflex Workflow

In this section we will describe more in detail which are the main components used most commonly in a Reflex workflows.

4.1 Setup directories

Some actors require that some variables are set in order to work properly. In particular, the *RecipeExecutor* actor requires the following variables:

- `BOOKKEEPING_DIR`: a directory where each pipeline recipe execution will create a subdirectory to use as a working directory. Useful for debugging purposes, since it contains the input sof for each call of the recipe, the output sof created, the parameters used and the esorex command executed.

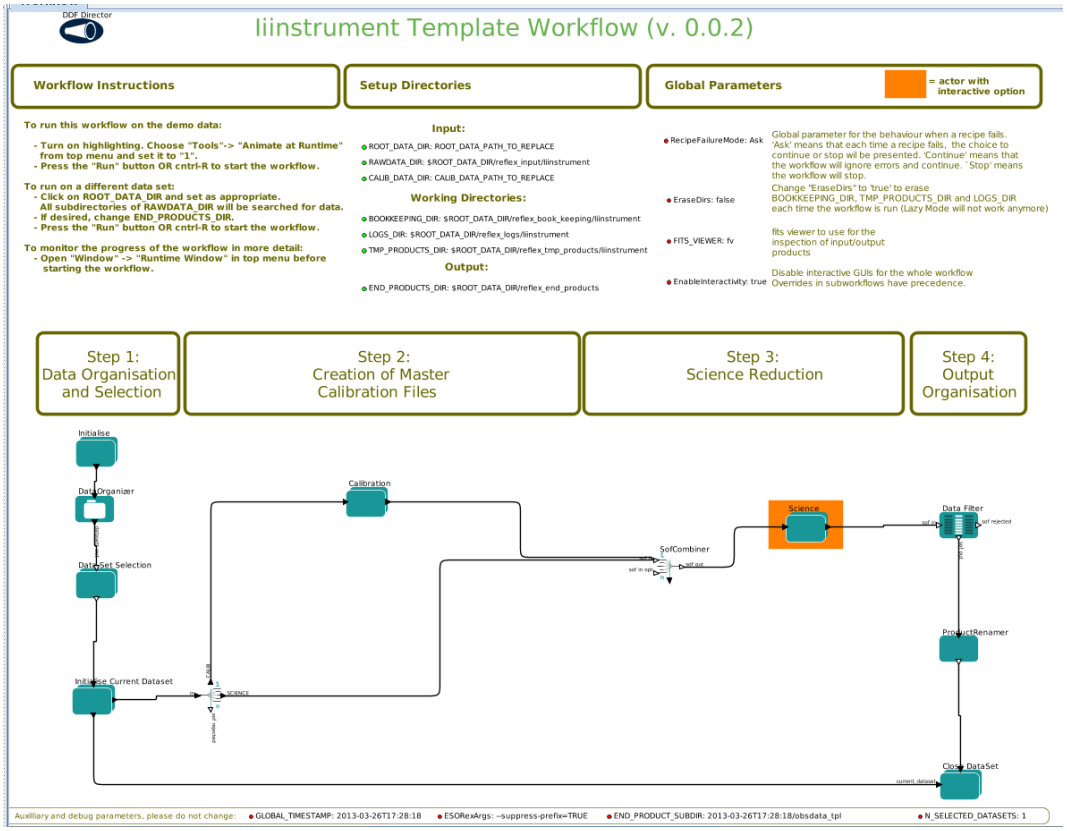

Figure 9: *The layout of the template workflow.*

- `LOGS_DIR`: a directory where the recipe logs will be saved. In particular, the esorex log will be stored here.
- `TMP_PRODUCTS_DIR`: a directory where the workflow intermediate products will be saved. Basically, for each `esorex` execution, the output directory will be pointed to here. Therefore, it can grow very easily, and it is recommended that it is pointed to a file system with enough disk space.
- `ESORexArgs`: additional parameters passed to esorex by the RecipeExecuter

Each time the `RecipeExecuter` launches a recipe via `esorex`, it will create a subdirectory under the mentioned directories named after the execution time stamp. In this way it is easy to recognise all the configuration, logs and products of each recipe execution.

It is actually possible to use a *RecipeExecuter* which does not use those variables, defining the corresponding actor parameters. However, being the default parameters, when the actor is instantiated, it requires them to be present. Moreover, it is convenient to have them as a global variable in the main canvas, so that they can be accessed by all the *RecipeExecuter* actors, including the subworkflows underneath.

A very convenient way to setup a workflow is to create a variable that defines a root directory and define the rest of the directories as subdirectories of that. For instance, create a variable called `ROOT_DATA_DIR` and point `BOOKKEEPING_DIR`, `LOGS_DIR` and `TMP_PRODUCTS_DIR` to subdirectories of it. A variable

can be used as part of the value of other variable if it is referenced using the $ operator, in a similar way as in shell programming.

Other variables that we recommended to define are:

- END_PRODUCTS_DIR: a directory where the workflow final products will be saved. Typically the *ProductRenamer* actor will use this variable to store the final renamed and copied products from the science recipes.
- FITS_VIEWER: executable used to visually inspect FITS files. This variable is used by the *DataFilter* actor.

4.2 Data Organisation

One of the most useful tasks of a workflow is the automatic organisation of large lists of files available on the disk. The organisation of the data is provided by the DataOrganiser actor, which actually uses a OCA file that specifies the organisation logic.

4.2.1 The OCA rules and Reflex

Reflex uses the OCA rules of the purpose of classification, grouping and association. The OCA syntax is explained in detail in 4.2.1 documentation, and here we will just point out the details specific to the use of OCA rules by the Reflex application.

OCA rules written for other applications (Gasgano, DO, ABbuilder) could in principle be used with Reflex, provided that some additions are included for the rules to work within Reflex. In fact Reflex uses the same library to parse the rules as all the other applications

4.2.2 Classification

For the classification part of the OCA rules, Reflex will use the special keyword REFLEX_CATG. Other applications, for instance DO, use the keyword DO_CATG, which Reflex will simply ignore. Usually, the value of both keywords would be the same.

Each of the classification entries look like the following:

```
 if DPR.CATG like "%SCIENCE%" and DPR.TYPE like "%OBJECT%" and INSTRUME=="TEMPLATE" ther
{
  REFLEX.CATG = "RRRECIPE_DOCATG_RAW";
}
```

The *like* operand will match for keyword values which contain the string between percentages. Any Java regular expression can be used.

In contrast, the == operator will do an exact matching of the values of the keyword. Note that it is recommended to match always the *INSTRUME* keyword, since it is a common use case that a user has data for different instruments in the same directory and for a given workflow, only the supported instruments should be matched. The use of *INSTRUME* is recommended also when matching static calibrations in the OCA rules, using the syntax *inputFile.INSTRUME==INSTRUME*. Even for virtual product associations this is a good practice.

One problem that might be encounter is that the user might have calibrations in the input directory of the workflow. Those calibrations might be picked instead of recreated from pure raw frames. The way to avoid that is to use a different frame type for the definition of the product. For instance, one might use the following syntax when the product of the bias action is defined:

```
product MASTER_BIAS_WKF { PRO.CATG=``MASTER_BIAS_WKF''; PRO.EXT=``tpl_0000.fits'';}
```

The association must be done with MASTER_BIAS_WKF as well of course, and this will avoid that a MASTER_BIAS frame type already found in the input dir is chosen.

4.2.3 Target of the workflow

Reflex uses the OCA rules to define the calibration/association cascade. However, there must be a way to indicate where the chain should be started. For instance, many workflows aim to reduce science frames, while some other specialised workflows will just create calibration files.

The way to specify the target is using the REFLEX_TARGET classification:

```
 if DPR.CATG like "%SCIENCE%" and DPR.TYPE like "%OBJECT%" and INSTRUME=="TEMPLATE" ther
{
  REFLEX.TARGET = "T";
}
```

A workflow can have more than one target, for instance science images and standard star images, although the rest of the workflow has to support both reduction chains, of course.

4.2.4 Grouping and action triggering

Reflex doesn't need any special syntax or additions to the grouping and select commands in OCA. The usual syntax is like

```
  select execute(DARK) from inputFiles where RAW.TYPE=="DARK"
  group by DET.READ.CLOCK,DET.CHIP1.ID,DET.WIN1.BINX,DET.WIN1.BINY,TPL.START as (TPL_A,t
```

Note that the grouping usually contains TPL.START, which will group all the images of the same template together, or ARCFILE, which will create just one group per file, since this keyword is unique to each file.

The clause *where* can use any of the classification keywords defined in the classification part. If compatibility with other software (DO, ABbuilder), the same keyword used there could be used here.

If a group should contain at least a minimum number of files (for instance a raw bias group should have at least 5 raw biases), then the clause *minRet* can be added before the select *clause*, like here:

```
 minRet = 5;
 select execute(BIAS_BLUE) from inputFiles where RAW.TYPE=="BIAS_BLUE"
  group by DET.CHIPS,DET.WIN1.BINX,DET.WIN1.BINY,DET.READ.SPEED,TPL.START as (TPL_A,tpl)
```

4.3 File Purpose

In Reflex 2.0, each file which is part a Sof carries around a *Purpose*. It represents what a file is needed for. The purpose of a file describes its position in the whole reduction cascade that will make use of the file. For example, a raw BIAS might have purpose MASTER_BIAS/SCIENCE if it's going to be used to produce the master bias that will be used to reduce the science frames.

4.4 FitsRouter

The FitsRouter actor is used to send the different types of frames to the relevant recipes. The name of the port is used to determine the type of frame (as created by the DataOrganiser) that will be output to that specific port. If one needs to output more than on type of file for that port then do the following:

- Create an output port with the desired name.
- Configure the actor and add a parameter.
- Name the parameter after the name of the port with _config appended.
- Specify that the parameter is StringParameter (if this is not done, the value of the parameter must be quoted).
- Fill the value of the parameter with a common separated list of frame types that should be output to that port

4.5 ProductRenamer

After having processed the input data for a data set, the workflow highlights and executes the `Product Renamer` actor, which, by default, will copy the most important final products of the UVES pipeline recipe `uves_obs_scired` to the directory specified by `END_PRODUCTS_DIR` and rename them with names derived from the values of certain FITS header keywords. Specifically, final products are renamed by default with names of the form `<HIERARCH.ESO.OBS.NAME>_<HIERARCH.ESO.PRO.CATG>.fits`, where `<HIERARCH.ESO.OBS.NAME>` and `<HIERARCH.ESO.PRO.CATG>` represent the values of the corresponding FITS header keywords. These names are fully configurable by double-clicking on the `Product Renamer` actor and editing the string as appropriate.

4.6 Recipe Execution

The relations in a Reflex workflows contain XML files that describes a Sof. This Sof are used as input to execute recipes in general. However, through a single relation there could be several files with different purposes. For instance, the input of the bias recipe could have biases used for the flat and biases used for the science. In order to properly use the relevant files, the actor *SofSplitter* has been created. The *SofSplitter*, as its name implies, will create several Sof with the same purposes, one at a time. This way, the *RecipeExecuter* will receive a coherent Sof for only one purpose. The *SofAccumulator*, on the other hand, waits until all the products of the recipes have been produced and joins them all together.

The proper way to use the *RecipeExecuter* will be to place a *SofSplitter* in front of it and a *SofAccumulator* after it. This two actors have to be connected each other by the *# groups* port. Figure 10 shows how to do this in general.

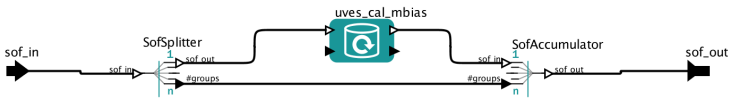

Figure 10: *This figure shows how to properly use a RecipeExecuter actor in combination with a SofSplitter and a SofAccumulator.*

As explained in the Mini-HOWTO, in order to instantiate the RecipeExecuter one has to use the menu option `Tools -> Instantiate Component`. The list of recipes that will appear there is the result of the command `esorex --recipes`. In othere words, the recipes have to be visible to esorex before they can be instantiated in the workflow. If you have installed Reflex using the install_reflex script, the `reflex` command sets the path to the `esorex` found within the installation, which in turn will found the recipes within that installation too. If you have your own recipes, then you will have to install them in the installation tree. If you installed Reflex directly from the tar file, then it will use the `esorex` command found in your path. Consult the documentation for `esorex` to see how to configure it in order to found your installed recipes.

Take into account that in order to help for the readability of the workflow, it is recommended to place these three actors inside a *CompositeActor*.

One important question is which is the *Purpose* of the products created by a given recipe.

In the following we describe in more detail the function of some of the parameters for a recipe executor actor:

- The "recipe" parameter states the UVES pipeline recipe which will be executed.
- The "mode" parameter has a pull-down menu allowing the user to specify the execution mode of the actor. The available options are:
  - `Run`: The pipeline recipe will be executed, possibly in Lazy mode (see Section 4.6.1). This option is the default option.
  - `Skip`: The pipeline recipe is not executed, and the actor inputs are passed to the actor outputs.
  - `Disabled`: The pipeline recipe is not executed, and the actor inputs are not passed to the actor outputs.
- The "Lazy Mode" parameter has a tick-box (selected by default) which indicates whether the recipe executor actor will run in Lazy mode or not. A full description of Lazy mode is provided in the next section.
- The "Recipe Failure Mode" parameter has a pull-down menu allowing the user to specify the behaviour of the actor if the pipeline recipe fails. The available options are:
  - `Stop`: The actor issues an error message and the workflow stops. This option is the default option.
  - `Continue`: The actor creates an empty output and the workflow continues.
  - `Ask`: The actor displays a pop-up window and asks the user whether he/she wants to continue or stop the workflow.
- The set of parameters which start with "recipe param" and end with a number correspond to the parameters of the relevant UVES pipeline recipe. By default in the recipe executor actor, the pipeline recipe parameters are set to their pipeline default values. If you need to change the default parameter value for any pipeline recipe, then this is where you should edit the value. For more information on the UVES pipeline recipe parameters, the user should refer to the UVES pipeline user manual (Larsen et al. 2010[2]).

[2] Available from http://www.eso.org/sci/facilities/paranal/instruments/uves/doc/index.html

4.6.1 Lazy mode in recipe execution

By default, the recipe executor actors have "Lazy Mode" enabled. This means that when the workflow attempts to execute such an actor, the actor will check whether the relevant pipeline recipe has already been executed with the same input files and with the same recipe parameters. If this is the case, then the actor will not execute the pipeline recipe, and instead it will simply broadcast the previously generated products to the output port. The purpose of the Lazy mode is therefore to minimise any reprocessing of data by avoiding data rereduction where it is not necessary.

One should note that the actor Lazy mode depends on the contents of the directory specified by `BOOKKEEPING_DIR` and the relevant FITS file checksums. Any modification to the directory contents and/or the file checksums will cause the corresponding actor when executed to run the pipeline recipe again, thereby rereducing the input data.

The forced rereduction of data at each execution may of course be desirable. To force a rereduction of all data for all recipe executor actors in the workflow (i.e. to disable Lazy mode for the whole workflow), set the `EraseDirs` parameter under the "Global Parameters" area of the workflow canvas to `true`. To force a rereduction of data for any single recipe executor actor in the workflow, right-click the actor, select `Configure Actor`, and uncheck the Lazy mode parameter tick-box in the "Edit parameters" window that is displayed. For a composite actor, you will first need to open the subworkflow by right-clicking on the composite actor and selecting `Open Actor`.

4.6.2 Variable Setters

This actor can be used to set a variable of the workflow at run time. For instance, it is used in most of the VLT workflows to set variable `GLOBAL_TIMESTAMP` using the current time. It is recommended that all the `Variable Setter` actors have the parameter `delayed` unchecked.

4.6.3 Update to a change in the recipe

Sometimes, a change in the recipe interface cannot be avoided and a given parameter is renamed, deleted or added. Reflex stores in the workflow the order of the parameters. For this reason, even a change in the order in which the parameters are defined is strongly discouraged. However, if none of the parameters of the recipe have been changed from the defaults, the workflow probably didn't store any value of the parameters and nothing should be changed.

The RecipeExecuter actor stores the recipe parameters by the order they appear in the recipe (i.e. like in esorex –man recipe_name). If a recipe has been changed, the best way to incorporate the changes is simply to delete the current actor and reinstantiate it: Reflex will read the recipe parameters again and will create a new configuration. In other cases, reinstantiating will cause too much trouble and the only change was maybe a parameter renaming. In that case, the workflow .xml could be edited and searched for strings like `<property name="recipe_param_`. Once the right parameter has been found (make sure that the recipe is the correct one: `<entity name="name_of_recipe_1"`), you can change the value of the *Reflex* parameter, which is in fact the name of the *recipe* parameter and the value together.

4.7 Subworkflows

A subworkflow can be created with a `CompositeActor`. To open the subworkflow canvas, right click on the actor and select `Open Actor`. Note that it is handy to use the arrows toolbox to create ports directly for the subworkflow. The name of the ports, however, should be changed in the usual way for the composite actor.

During the overall design of the workflow it should be considered which parts should be grouped into sections, and which parts should go into a subworkflow. As a rule of thumb, most workflows should not

contain more than 10 to 15 actors or subworkflows. Subworkflows should be kept simple and have clean and simple input and output ports. For example, if a recipe or other actor is called several times with different input or in different modes which depend on available data, these actors and the associated logical elements should typically be moved to a dedicated subworkflow.

4.8 Provenance actor

The Provenance actor is used to display the full reduction chain for a given pipeline product. As explained above, each time a recipe is run, its inputs and outputs are stored in the bookkeeping database. This database can be queried to reconstruct the full set of frames that were used for a particular product. In fact this information is better represented by a tree, and that's how the Provenane actor will display it.

The Provenance actor allows to specify a range in time for the products to be displayed. A port can be used to specify the starting date and hour. In In a workflow, which usually stores the starting time, this can be used to display only the products created during that particular run of the workflow.

Other parameter needed by the Provenance actor is the path to the provenance database. In a standard configured workflow this should point to $BOOKKEEPING_DIR/bookkeeping.db.

5 Interactivity in the workflow

One of the benefits of using Reflex to create workflows is the ability to insert some interactivity in the reduction process which includes some kind of data visualisation and fine tuning of pipeline parameters. Reflex has full support for this kind of interactivity using a dedicated Python module.

The Python module makes use of some third-party modules: numpy, pyfits, matplotlib and wxPython. It is strongly recommended to write interactive windows using these tools.

The general philosophy of the interactive windows is to place them after a specific pipeline recipe and show a subset of the results where the user can asses the quality of their data. Additionally, the user can easily modify the most relevant pipeline parameters to accomplish a successful reduction. Figure 11 shows an example of such a window.

5.1 Reflex Python Interactive Module

The Reflex Python module is comprised of 4 main components:

1. The basic Python interface with Reflex. This is the same explained in section 5.
2. The Interactive Application module. This module is responsible for the definition of inputs and outputs of an interactive actor as well as the creation and handling of the interactive window
3. The Pipeline Products module. This module helps reading common types of data from FITS files as delivered by VLT instruments.
4. The Pipeline Display module. This module allows quite easily to display specific types of data with just a few of Python commands.

The first two modules are mandatory for any interactive window, while the last two can be useful to read and display common data structures, but they might not implement the support for a given instrument data. Currently images, spectra and scatter plots from tables are supported, with more types of data and/or visualisations coming in the future.

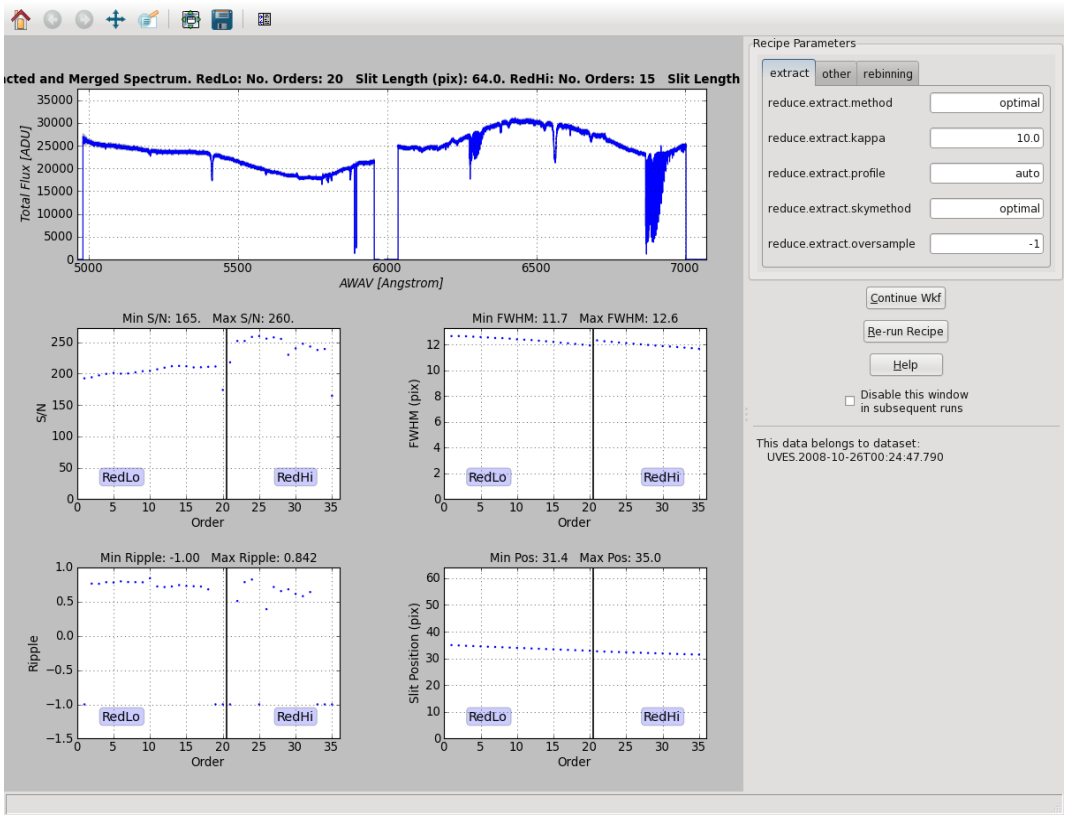

Figure 11: *An example of a Reflex Python interactive window.*

5.1.1 Python Interactive Module interface

The developer can use the above described framework to easily prototype and implement and interactive window. There are some basic ingredients that the developer must provide, however:

- The set of files that have to be read.
- The plotting layout.
- The actual plotting of the different data.
- The recipe parameters that can be changed.
- The window title and help (these being optional).

The way to provide this information is via an object which implements the following methods, in the same order as above:

- `readFitsData()`. This method will receive a `reflex.FitsFiles` object which contains the output of a recipe execution (previous actor connected) and will allow to select the relevant files to be read. Reading the files can be accomplished using the `PipelineProduct` class.

- `addSubplots()`. This method receives a matplotlib `matplotlib.figure.Figure` object and is used to define the number and location of subplots, mainly trough the `add_subplot` method. The `add_subplot` method takes three arguments (I, J, K) where IxJ defines a grid and K is the place in the grid starting from top left corner. Note that in the same plot several subplots can be added which don't neccessarily share the same grid dimensions.
- `plotProductsGraphics()`. This method receives a `Figure` object and a `FigureCanvasWxAgg` object. The first one will be used to actually do the plotting, while the second argument is deprecated and should not be used.
- `setInteractiveParameters()`. This method does not receive any arguments, and it should return a list of `reflex.RecipeParameter` objects which will be listed by the interactive window as potential recipe parameters to be changed.
- `setWindowTitle()`. This method with no arguments should return a string with the title of the interactive window.
- `setWindowHelp()`. This method with no arguments should return a string with the help of the interactive window, as displayed when the `help` button is clicked.

5.1.2 Python Plotting Module

As mentioned above, Reflex includes a module to ease the reading and displaying data from ESO pipelines. There are two components, one for reading and one for displaying. They can be seen as orthogonal in functionality, since they don't depend on each other, but used together they simplify the creation of interactive plots. Here is a description of the basic functionality:

1. `PipelineProduct`. This class implements easy reading in a Python structure of some of the most common pipeline products. So far the following methods have been implemented.
    - `PipelineProduct()`. Constructor. The constructor receives a `reflex.FitsFile` object.
    - `readImage()`. This method will read a FITS image. It takes the FITS extension number as an argument, starting at 0, which is also the default if it is not given. After calling the method, the image is accessible through the `self.image` member.
    - `readSpectrum()`. This method will read a FITS spectrum, considering that a 1-D FITS image. It takes the FITS extension number as an argument, starting at 0, which is also the default if it is not given. After calling the method, the spectrum is accessible through the `self.flux` member. The wavelength range is available through the `self.wave` member. Other additional members are `self.start_wave`, `self.end_wave`, `self.crval1`, `self.crpix1`, `self.cdelt1`, `self.bunit` and `self.type1`.
    - `readTableXYColumns()`. This method will read two columns from a FITS table. It takes the FITS extension number as first argument `fits_extension`, starting at 0. Two more arguments `xcolname, ycolname` specify the names of the columns to read. After calling the method, the X, Y values are stored in members `self.x_column, self.y_column` columns respectively.
    - `readLinearWCS()`. This method will read a linear WCS solution represented by keywords CRVAL1, CRPIX1, CDELT1, and CTYPE1. It takes the FITS extension number as an argument, starting at 0, which is also the default if it is not given. After calling the method. The solution is stored in members `self.crval1`, `self.crpix1`, `self.cdelt1`, and `self.type1`. If the keywords are not present, these members are assigned `None`.

- `read2DLinearWCS()`. Similar to `readLinearWCS()` but it also reads keywords CRVAL2, CRPIX2, CDELT2, and CTYPE2.
- `hdulist()`. This method returns a `pyfits.HDUList` object with all the HDUs present in the FITS file. It can be used to get quick access to the pyfits object in case none of the above methods solves the reading of a particular type of data (for instance OIFits format).

2. `PipelineDisplay`. This module implements easy displaying in a matplotlib environment of common pipeline products. So far the following classes have been implemented:

   - `ScatterDisplay`. This class allows to plot a scatter graph, X vs Y. The basic usage is as follows:

     ```
     scadsp = ScatterDisplay()
     scadsp.display(subplot, title, tooltip, x, y)
     ```

     where x and y have the same length and where `subplot` is a subplot of a matplolib.Figure object.

     Additionally, the following methods can be used to change the aspect of the plot: `setPointSize(self, size)`, `setLabels(self, xLabel, yLabel)`, `setLimits(self, xMin, xMax, yMin, yMax)`. All of them should be called before the `display` method.

   - `SpectrumDisplay`. This class allows to plot a spectrum. The basic usage is as follows:

     ```
     specdsp = SpectrumDisplay()
     specdsp.display(subplot, title, tooltip, wave, flux)
     ```

     where `wave` is the array with the wavelengths and `flux` contains the fluxes. The argument `errorflux` can also be used to plot error bars in the spectrum. Automatic axes limits in the plot can be set with `autolimits=True`.

     Additionally, the following methods can be used to change the aspect of the plot: `setWaveLimits(self, wave_limits)`, `setLabels(self, xLabel, yLabel)`. All of them should be called before the `display` method.

     Also, the method `overplot(self, subplot, wave, flux, color='green')` can be used to overplot a second or more spectra in the same plot. This method is called after the `display` and the `subplot` argument must be the same.

   - `ImageDisplay`. This class allows to a 2D image. The basic usage is as follows:

     ```
     imadsp = ImageDisplay()
     imadsp.display(subplot, title, tooltip, image)
     ```

     where `image` is the 2D image to plot. Additionally a `bpmimage` argument can be added which will flag in red all the pixels with bpmimage ¿ 1.

     Additionally, the methods `setXLinearWCSAxis(self, crval1, cdelt1, crpix1)`, `setYLinearWCSAxis(self, crval2, cdelt2, crpix2)` can be used to specify a different X or/and Y axis transformation than just pixels. It follows WCS convention for linear transformation.

     Also, `overplotScatter(self, x, y, marker='+', size=1.4, color='blue')` can be used to overplot some scattered points over the image. This method is called before `display` and the `subplot` argument must be the same.

5.1.3 Basic implementation of an interactive window

We will explain here how to create a basic interactive window using the ingredients explained above:

1. Create a python file with the following import statements:

```
try:
    import numpy
    import reflex
    from pipeline_product import PipelineProduct
    import pipeline_display
    import_success = True

except ImportError:
    import_success = False
    print "Error importing modules pyfits, wx, matplotlib, numpy"
```

   Note that some external modules are needed to be installed (`pyfits, wx, matplotlib, numpy`) while others are port of the Reflex Python modules (`reflex, pipeline_product, pipeline_display`).

2. Create a class which implements the interface needed by the interactive window framework:

```
class DataPlotterManager(object):
    def setInteractiveParameters(self):
        ...
    def readFitsData(self, fitsFiles):
        ...
    def addSubplots(self, figure):
        ...
    def plotProductsGraphics(self, figure, canvas):
        ...
```

3. Complete the function which defines the parameters to be displayed:

```
    def setInteractiveParameters(self):
        return [
            reflex.RecipeParameter(recipe="recipe"_name, displayName="stropt",
                                   group="group2", description="Desc1"),
            reflex.RecipeParameter(recipe="recipe"_name, displayName="boolopt",
                                   group="group1", description="Desc2"),
            reflex.RecipeParameter(recipe="recipe"_name, displayName="intopt",
                                   group="group1", description="Desc2"),
        ]
```

4. Complete the function which will organise the files received by the python actor:

```
    def readFitsData(self, fitsFiles):
        self.frames = dict()
        for f in fitsFiles:
            self.frames[f.category] = PipelineProduct(f)
```

5. Complete the function which sets up the subplots of the gui:

```
def addSubplots(self, figure):
    self.img_plot = figure.add_subplot(111)
```

6. Complete the function which finally plots everything:

```
def plotProductsGraphics(self, figure, canvas):
    # get the right category file from our dictionary
    p = self.frames["RRRECIPE_DOCATG_RESULT"]
    p.readImage()
    # setup the image display
    imgdisp = pipeline_display.ImageDisplay()
    imgdisp.setLabels('X', 'Y')
    tooltip = "This a an image"
    imgdisp.display(self.img_plot, "Image title", tooltip, p.image)
```

7. Write the `main` of the python script. This part will be basically the same for all the interactive windows written within Reflex:

```
#This is the 'main' function
if __name__ == '__main__':
    from reflex_interactive_app import PipelineInteractiveApp

    # Create interactive application
    interactive_app = PipelineInteractiveApp()

    # get inputs from the command line
    interactive_app.parse_args()

    #Check if import failed or not
    if not import_success:
        interactive_app.setEnableGUI(False)

    #Open the interactive window if enabled
    if interactive_app.isGUIEnabled():
        #Get the specific functions for this window
        dataPlotManager = DataPlotterManager()

        interactive_app.setPlotManager(dataPlotManager)
        interactive_app.showGUI()
    else:
        interactive_app.set_continue_mode()

    #Print outputs. This is parsed by the Reflex python actor to
    #get the results. Do not remove
    interactive_app.print_outputs()
    sys.exit()
```

The template pipeline `iiinstrumentp` contains an example of a basic interactive window script based on these instructions.

5.2 Looping on a recipe execution with an interactive window

5.2.1 Deciding which parameters can be modified

First thing is to decide which are the parameters that the user will see when running the interactive window. These parameters must have some initial values that are used for the first iteration. These initial values have to be created as variable in the canvas, like it is shown in figure 12.

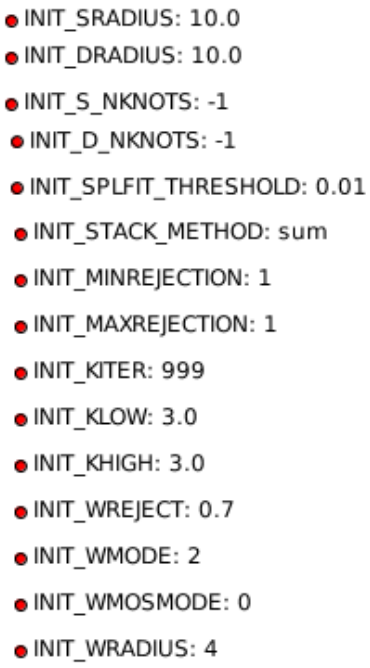

Figure 12: *Initial values for the first iteration of a looping subworkflow.*

The recipe that is going to be used for the looping must also be configured to accept the parameters values from external actors. For this, open the RecipeExecutor actor configuration and set those recipe parameters to the special value PORT. This means that the values will be retrieved from the sop port of the actor. Figure 13 shows an example of this configuration.

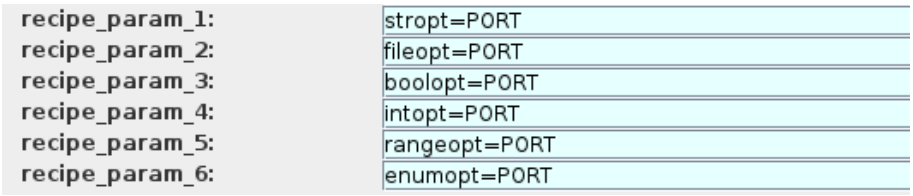

Figure 13: *Configuration of a RecipeExecuter to accept parameter values from a port.*

Finally, the python script has to be configured to show those parameters to the user. This is done in function `setInteractiveParameter`. Please refer to the Python framework section for details.

5.2.2 General layout for a looping

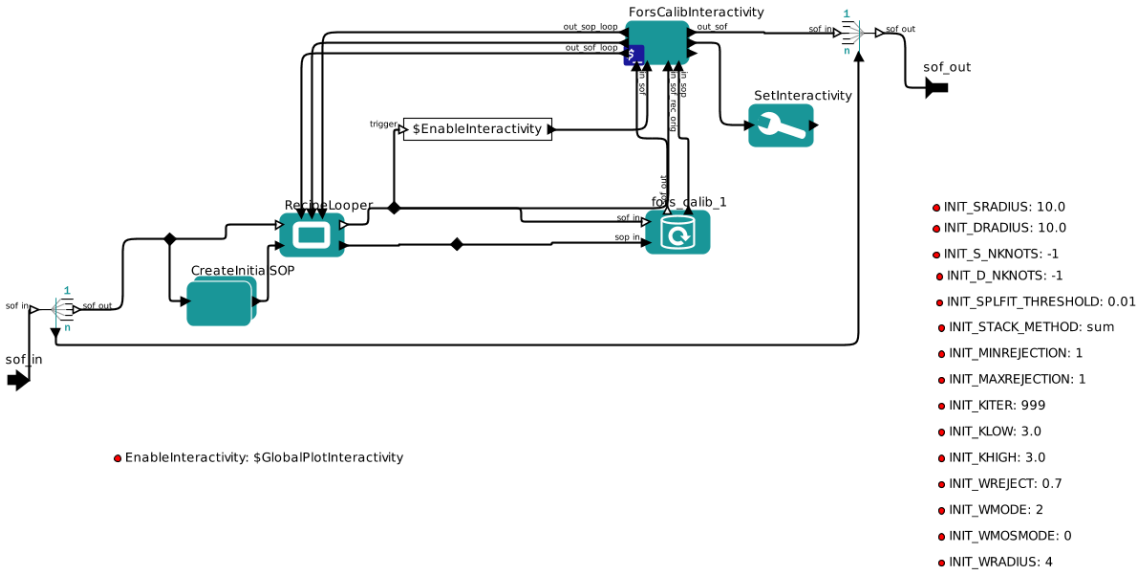

Figure 14: *General layout of a looping subworkflow.*

The general layout of a looping subworkflow is shown in figure 14. It contains the following elements:

- Create initial sop. This subworkflow will create a string with the initial list of recipe parameters in the form recipe:parameter=value, as shown in figure 15.

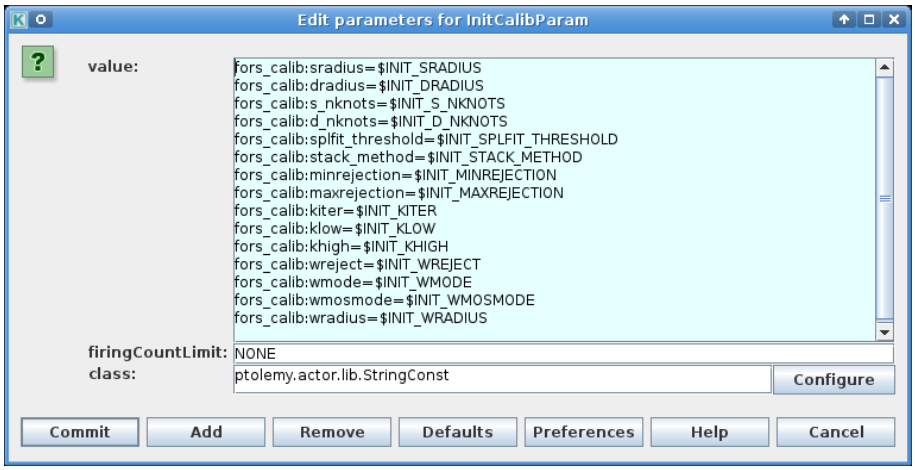

Figure 15: *A string constant that generates the initial values for the recipe parameters.*

This string is passed to a SopCreator actor that will format the recipe parameters in the appropiate way understood by the rest of the actors.

- RecipeLooper. This is the main actor that allows the looping. It has three set of ports:
  - The initial set of values ports: `sof_in and sop_in`. This ports are used only for the first iteration.
  - The looping set of values: `sof_loop, sop_loop and iteration_complete`. The `sof_loop, sop_loop` are the values that will be used in all iterations but the first one. The port `iteration_complete` indicates whether the loop has ended. If true then the RecipeLooper will stop emitting tokens.
  - The output ports `sof_out and sop_out`. These are the outputs that will be used by the recipe. Their values are created from the initial set of values ports in the first iteration or from the looping set of values in the rest of the iterations.
- The recipe. Do not forget to connect the `sop_out and sof_out` ports of the RecipeLooper to the `sof_in and sop_in` ports of the RecipeExecuter.
- The Python actor. This is the interactive actor that will pop up the window with the displays and parameter to be changed. It has to be connected to the rest of the actors in a specific way:
  - `in_sop` must be the output the recipe executer `sop_out` port.
  - `in_sof` must be the output the recipe executer `sof_out` port.
  - `in_sof_rec_orig` should be the input frames of the recipe, which is the `sof_out` port of the RecipeLooper.
  - `enable` must contain true or false to indicate that the interactivity is enabled.
  - `out_sop_loop, out_sof_loop and iteration_complete` must be connected to the equivalent port of the RecipeLooper.
  - `out_sof` is the final output frames of the whole loop.
  - `set_enable` could be connected to a VariableSetter actor that controls the interactivity.

The template workflow contains one example that can be used as a starting point to work on a looping subworkflow.

6 Most useful tips

6.1 Supported directors

The recommended director is the DDF director. Other directors could be used, but problems have been found with the Parallel director. These problems might be solved in the future.

One interesting behaviour of the DDF director is that a composite actor doesn't necessarily trigger all the actors in it before firing the next actor in the top level. The only way to assure that this happens is checking option `runUntilDeadlockInOneIteration` is checked.

6.2 Saving Workflows

Saving a workflow uses by default the .kar format. However this is highly inconvenient for a workflow developer, since this format cannot be easily modified. As explained in section 7, there are tools to integrate a workflow in a pipeline that need to update the workflow. Therefore it is highly recommended to export the workflow as XML using the `File` menu.

6.3 How to write workflows which do not use purposes

From the description above it is clear that the *Purposes* are a crucial feature in the design of the workflows. However, if one is not interested in them and just want to process with a given recipe all the files which come from a given relation, it is possible to do it. This is not, however, the recommended way to create workflows.

Take into account that the behaviour might not be what you expect. If for instance a given calibration set has been associated several times in the association tree, you would get duplicated files into your recipe execution. One possible solution for that is to associate calibration files in the OCA rules only in the action which has `REFLEX.TARGET = T`.

6.3.1 Introducing a purpose-less recipe in the middle of a workflow

There are cases where one is interested in introducing a recipe in the middle of two already existing recipes. In order to work properly with *Purposes*, that change has to be reflected in the OCA rule, possible through an intermediate action. However this might not work, if the new recipe does not have raw frames as an input (this limitation of OCA rules might disappear in the future).

The option is to configure the *RecipeExecuter* with the option *Do nothing* in the parameter **File Purpose Processing**, like it is shown in figure 16.

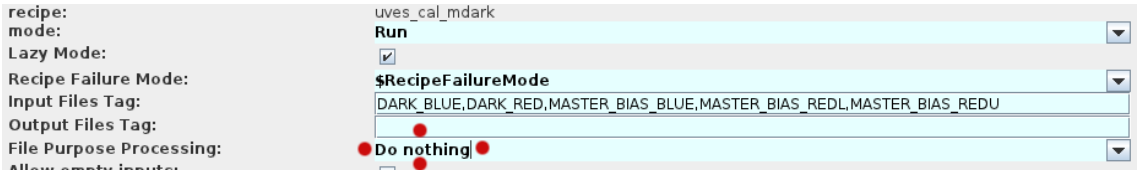

Figure 16: *Configuration of a RecipeExecuter to act as a pass-through recipe.*

6.4 Incomplete Datasets

Sometimes some of the datasets shown in the `DataSetChooser` are greyed out. Hovering the mouse over the dataset, or clicking on `Inspect Highlighted` will show what the problem is. This happens when the OCA rules contain mandatory associations that cannot be fullfilled. The way to make an association mandatory or not is changing the `minRet` keyword to 0 or greater than 0.

6.5 Browsing actor documentation

The most convenient way to read the documentation of an actor is to right click on it `Documentation -> Display`.

6.6 When to use the sof_opt port in the SofCombiner

The sof_opt port in the SofCombiner is used when there are some optional inputs. As explained in the Reflex User Manual, the SofCombiner will use all the common purposes found in the sof_in port to create the list of output purposes. The important thing here is to realise that the inputs in sof_opt port are not considered to found the common purposes.

An example on when to use the sof_opt is some optional step, for instance a dark recipe that needs extra calibrations, for instance a master bias. In this example, we assume that the dark recipe should be triggered

only if dark frames are present. If we connect to the sof_in the potential dark frames and the master bias, we might found the situation where there are no dark frames and the purposes of the master bias are used by the SofCombiner to determine the output purposes, which is not what we want. If we place the master bias (and other potential calibrations) in the sof_opt port, the SofCombiner will determine the common purposes based only on the presence of darks.

6.7 Reflex and the CalSelector

CalSelector is a tool to provide all the needed raw frames used to reduce a given type of data. In order to accomplish this, it uses OCA rules that define the calibration cascade. ESO sanctioned workflows use a similar set of OCA rules, allowing the workflow to be fed with the data retrieved using CalSelector. However, it has to be kept in mind that this is achieved only if the OCA rules do actually provide the same calibration cascade.

6.8 Requirements for input files

The input files of a workflow have to comply with some minimun requirements. The following keywords have to be found in the main header of the FITS file: ESO PRO CATG, INSTRUME and DATAMD5.

7 Deploying and delivering a workflow

If a workflow has to be delivered together with the pipeline, the recommended way is to incorporate it in the usual autotools setup of it. In this way many of the paths that appear in the workflow are automatically filled in by the autotools mechanism.

The procedure that we describe here will install a number of files when running the usual `make install` command:

- `$prefix/share/reflex/workflows/pipe-$vers`. This is the directory where workflows will be installed, where *pipe*−`vers` is the pipeline name and version. This place is where a user should look for a workflow
- `$prefix/share/esopipes/$pipe-$vers/reflex`. This is the directory where the extra files needed by the workflow (OCA rules, python scripts...) are installed.

The template pipeline implements this behaviour. We list here the steps to do:

- Place all your Reflex related files under `reflex` directory in the pipeline source tree.
- Edit file `acinclude.m4` in the pipeline source directory and under the function `[PIPE_SET_PATHS]` add the following:

```
if test -z "$wkfextradir"; then
    wkfextradir='${datadir}/esopipes/${PACKAGE}-${VERSION}/reflex'
fi

if test -z "$wkfcopydir"; then
    wkfcopydir='${datadir}/reflex/workflows/${PACKAGE}-${VERSION}'
fi
```

```
AC_SUBST(wkfextradir)
AC_SUBST(wkfcopydir)
```

- Edit file `configure.ac` and in the `AC_CONFIG_FILES` clause add the workflow:

```
reflex/Instrument.xml)
```

- Create a `reflex/Makefile.am` file with information about he workflow files, OCA rules and Python scripts. For example, for VIMOS, which contain two workflows but not Python scripts would be like this:

```
AUTOMAKE_OPTIONS = foreign
WORKFLOWS = VimosIfu.xml VimosMos.xml
OCAWKF =  vimos_ifu_wkf.oca vimos_ifu_wkf.dvd.oca vimos_mos_wkf.oca
PYTHONWKF =
wkfextra_DATA = $(WORKFLOWS) $(OCAWKF) $(PYTHONWKF)
EXTRA_DIST = $(WORKFLOWS).in $(OCAWKF) $(PYTHONWKF)
wkfcopy_DATA = $(WORKFLOWS)
```

- Make sure that in the top level `Makefile.am` the `reflex` subdirectory is in the `SUBDIRS` variable.
- The workflow should actually be named Instrument.xml.in (i. e., the name put in file `configure.ac` appended with .in). The paths to the OCA rule has to be set to `@prefix@/share/esopipes/pipe-@VERSION@/refl` where `pipe` is the name of the pipeline. Searching for string `OCA File` in the .xml will help.
- Also the paths to the python scripts should be updated accordingly. Search for string `Python script` and substitute the path with `@prefix@/share/esopipes/pipe-@VERSION@/reflex/script.py`.
- It is also convenient to set the title of the workflow to something descriptive plus the version of the pipeline. To do that, put the placeholder `@VERSION@` instead of the version.
- Also convenient is to substitute the data paths with the placeholder `ROOT_DATA_PATH_TO_REPLACE`. This is not substituted by autotools, but the `install_reflex` script will do it. Also the calibration directory should be `CALIB_DATA_PATH_TO_REPLACE/pipe-@VERSION@`.

The template pipeline/workflow contains a script called `reflex/parse_wkf_for_cvs` that will allow you to automatically parse the workflow file saved after an execution and prepare it for archiving in the SCM. Most likely the script should be edited to suit the particular needs and the workflow in particular.

8 Guidelines for VLT instruments workflows

Official workflows developed by ESO and delivered as part of the official pipeline shall follow a number of guidelines regarding graphical layout, standard components and user interface. Here we list these guidelines, which are also recommended for any user created workflow. All these guidelines are implemented in the template workflow, which can be used as a reference.

- The top level of a workflow should represent an overview of the whole data reduction flow, with some annotations and the main parameters to setup the workflow. Actors should be grouped in sections. Each section represents a subjects relevant to the astronomer. The sections should be be labelled

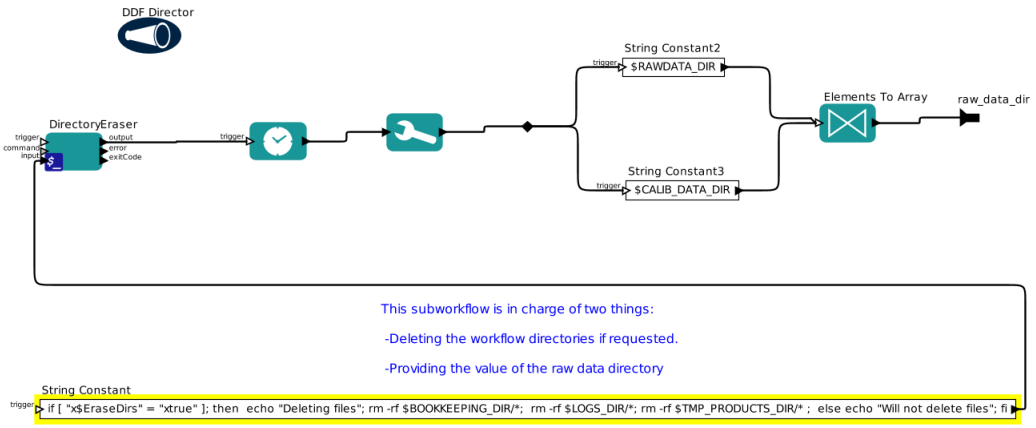

Figure 17: *Initialise composite actor.*

accordingly. The first and last section should be *Data Organisation* and *Output Organisation*. Other sections describe top level processing steps, such as *Master Calibrations*, *Preprocessing* or *Image Combination*.

- Workflow initialisation. Should be placed on the left side of the main canvas. These actors prepare the rest of the workflow to start with the data reduction. The workflow shall start with a composite actor that deletes all the directories and prepares the input for the `DataOrganiser`. It consists of an `External Execution` actor that receives a shell command to remove the `BOOKKEEPING_DIR` and `TMP_PRODUCTS_DIR` directories. For that this actor has to be configured with parameter `command = sh` and should have in the `input` port a string like:

  ```
  if [ "x$EraseDirs" = "xtrue" ]; then  echo "Deleting files"; rm -rf
  $BOOKKEEPING_DIR/*;  rm -rf $LOGS_DIR/*; rm -rf $TMP_PRODUCTS_DIR/* ;
  else echo "Will not delete files"; fi
  ```

  The `firingCountLimit` parameter should be set to 1. The output of this actor is used to trigger a `Time Stamp` actor which is used as an input to a `Variable Setter` actor that sets `GLOBAL_TIMESTAMP`. This in turn will trigger two `String Constant` actors that use the values of the global variables `RAWDATA_DIR` and `CALIB_DATA_DIR` to create an array using the `Elements to Array` actor. Remember to set parameter `firingCountLimit` to 1 for the `String Constant` actors.

  It is important to add a `DDF Director` with parameter `runUntilDeadlockInOneIteration` checked.

  Figure 17 shows the implementation of this initialisation composite actor.

- The `DataSetChooser` should be placed in a subworkflow as shown in figure 18. The variable that has to be updated is `N_SELECTED_DATASETS`. It is important to add a `DDF Director` with parameter `runUntilDeadlockInOneIteration` checked, otherwise it is not warrantied that the next actors would be triggered without the variable properly set. This variable is used for displaying purposes only.

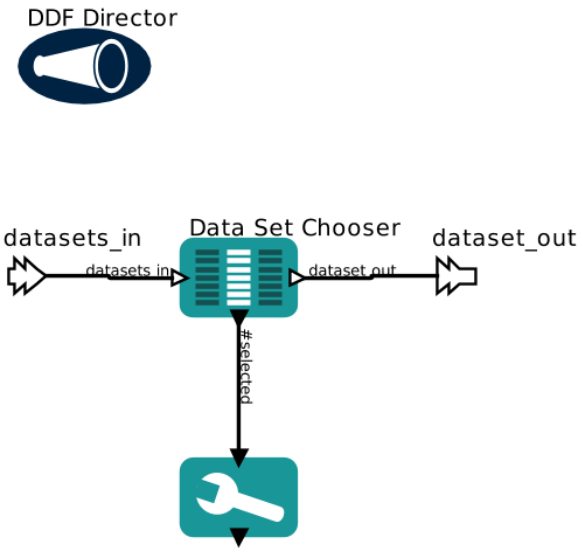

Figure 18: *Data selection actor.*

- After the `DataSetChooser`, a composite actor `Initialise Current Dataset` will be in charge of updating some variables for visualisation purposes. It is also the responsible of creating the final directory where the processing of *this* dataset will be stored. The implementation is not difficult, but actually the best way to do it is follow the template workflow example. Figure 19 shows the final outcome of this.

  Take into account that the `dataset` port of this composite actor will go to the `FitsRouter actor` in the top level canvas. The `current_dataset` port will be connected with the end of the workflow, the `CloseDataset` actor (see below).

  Also in the actor it is important to add a `DDF Director` with parameter `runUntilDeadlockInOneIteration` checked.

- `Closing Dataset`. This composite actor shall be places after the reduction cascade, the `DataFilter` and the `ProductRenamer`. It will display a window at the end of the reduction of *this* dataset showing the directory where the data has been successfully saved. Figure 20 shows how to implement this composite actor. The template workflow can also be used as a reference.

- The actors which include interactive steps shall have a distinctive orange background around them. You can create it using the `Rectangle` actor.

- Direction of Data Flow. The overall data flow in any workflow should be either from left to right, or from top to bottom. Actors and subworkflows should be placed along this overall direction of the workflow in such a manner that their position reflects the logical order of the flow.

- The connections between actors should also be grouped and labelled. Similar data should flow in parallel connections. It should be easy to trace back each input to an actor to its origin.

- Logical Elements. The implementation of relative easy logical decisions sometimes require fairly complex constructs in Kepler. Such constructs should be hidden in a subworkflow. An example of this is the Flat Combiner in the official UVES workflow for spectroscopy.

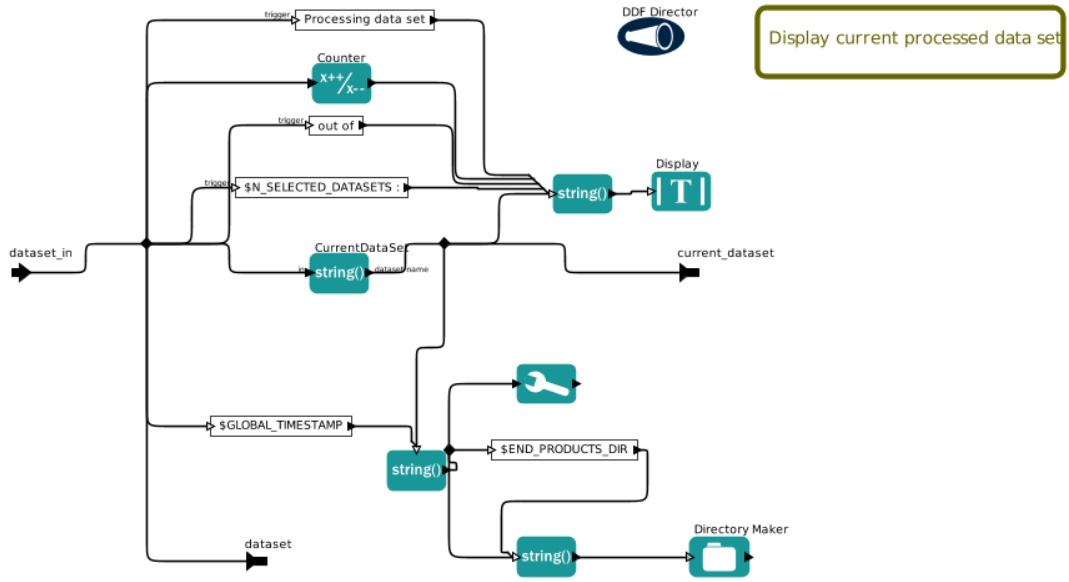

Figure 19: *Initialise current dataset actor.*

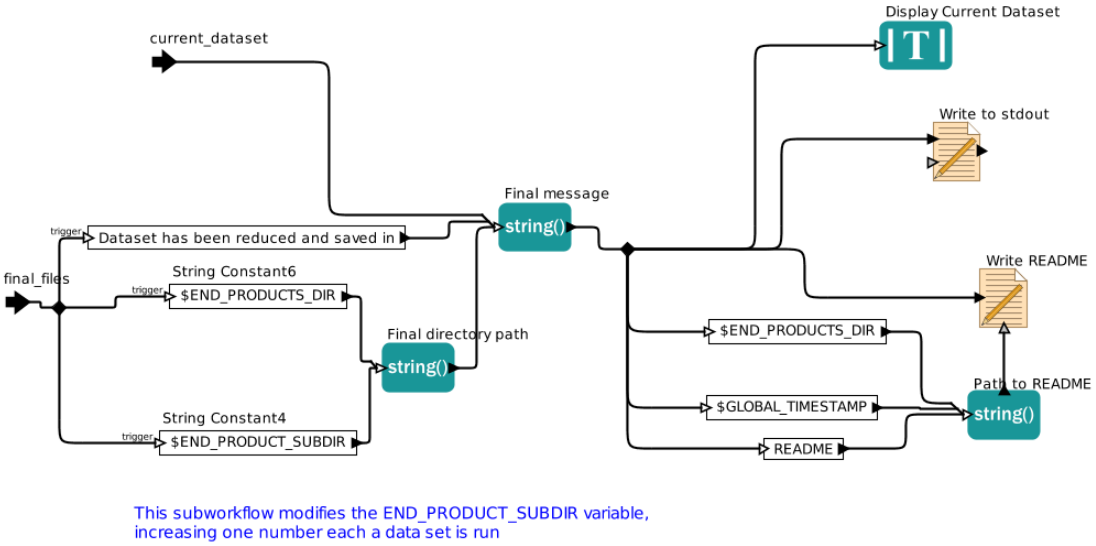

Figure 20: *Closing Dataset actor.*

- To draw the attention of the user to the most important steps (e.g. interactive actors), actors should be marked with coloured boxes. The color scheme should be explained on the workflow
- Use a global `RecipeFailureMode` variable. The individual `RecipeExecuters` can be con-

figured to use the value of a variable in the `Recipe Failure Mode` parameter.

- Use global non-interactive mode. It is useful to configure the workflows in such a way that all the non-interactivity is disabled. To do that, some global variables are defined in the main canvas: $GlobalPlotInteractivity should be used to activate/deactivate the interactivity for the python windows that are enabled by default. The EnableInteractivity parameter in those python actors is set to $GlobalPlotInteractivity. $ProvenanceExplorerEnabled should be used to enable/disable the provenance explorer actor window. The mode parameter of this actor should be set to this variable. $DataSelectionMethod should be used to set the bahaviour of the data selection actor. The default value should be *Interactive*. The parameter mode of this actor should be set to this variable.

A Porting from Reflex 1.x to 2.0

Here we have summarised the important points to take into account when porting a workflow developed with Reflex 1.x to Reflex 2.0.

A.1 Structural changes

- Place a SofCombiner before composite actors (where before there was simply a RecipeExecutor. This will ensure that only common purposes to all the channels go through.
- Change the EmptySof string. Use the SofCreator with an empty string as an input - Use the new IsSofEmpty actor
- Place PurposeSerialized/DeSerialized before each recipe.
- Set StripLast Purpose in all the recipes as a general rule.
- Change all the ports editing manually the XML.
- Change the ProductRenamer, since the code is new and the old one is directly embedded in the workflow.
- Set the widths of all the channels which go to the SofSplitter to Auto.

A.2 Design changes

- In the FitsRouter, if two (or more) DO.CATG have several purposes, one has to separate them in different ports, otherwise the Purpose matching of the SOFCombiner won't work.
- In the OCA rules, if recipe1 generates 2 virtual products, and recipe2 needs them, only one virt product can be associated. If you associate both, then there is a duplication of files.
- Do not connect the output of two recipes to the same SofCombiner unless one is really sure that they share exactly the same purposes. Otherwise the non-matching purposes will be lost.
- Set the Variable Setters not to be delayed. This is not specific to Reflex 2.0, it is more a guideline.
- For massive reduction of data. Put everything to Continue 'Failure mode' and non-interactivity. Run everything. Then put back to Ask and enable interactivity, to review the problems.

A.3 Testing

- Check that the executed purposes match the action tree (presented by the Data Set Chooser), for instance:

```
grep -i action reflex\_book\_keeping/Uves/uves\_cal\_mbias\_1/*/*xml | uniq
```

- For each recipe, make sure that all the input SOFs correspond to all the presences of that action/recipe in the action tree

B Porting from Reflex 2.0 to 2.2

Here we have summarised the important points to take into account when porting a workflow developed with Reflex 2.0 to Reflex 2.2.

B.1 General

- Reinstantiate the Product Renamer. It can happen that the workflow won't load at the beginning. Skip the error and reinstantiate it.

B.2 Python Framework

- PipelineInteractiveApp class:
  - Change isEnabled() to isGUIEnabled()
  - Change setEnable() to setEnableGUI()
  - Change showWindow() to changeGUI()
  - readFitsData() receives now a list of FitsFiles
  - Method passProductsTrough() has been changed to passProductsThrough()
- PipelineInteractiveApp class:
  - Change isEnabled() to isGUIEnabled()
  - Change setEnable() to setEnableGUI()
  - Change showWindow() to changeGUI()
  - readFitsData() receives now a list of FitsFiles
  - Method passProductsTrough() has been changed to passProductsThrough()
- PipelineProduct class:
  - The constructor now receives a FitsFile object
- RecipeParameter class:
  - Name changed to RecipeParameter from RecipeParam.
  - The constructor is now:

```
def __init__(self, recipe="", name="",longName="", group="", description="", val
```

B.3 Jython scripts

- XMLTools doesn't exist anymore. Use JSONTools instead like this:

```
#Read input file names
 input = self.sof_in.get(0)
 string = input.stringValue()
 obj = JSONTools.fromJSON(string)
...........
 #Broadcast outputs
 self.sof_out.broadcast(StringToken(str(output)))
```

B.4 Recipe Executer

- Change Input Files Tag –¿ Input Files Category (Better in the XML file directly)
- Change Output Files Tag –¿ Output Files Category (Better in the XML file directly)

C Porting from Reflex 2.2 to 2.4

Here we have summarised the important points to take into account when porting a workflow developed with Reflex 2.2 to Reflex 2.4.

C.1 General

- Reinstantiate the Product Renamer. It can happen that the workflow won't load at the beginning. Skip the error and reinstantiate it.

C.2 Python Framework

- PipelineInteractiveApp class:
  - There is a new parameter called –set_init_sop. This parameter is used to output in a port the recipe parameters to be used in the future as initial ones. There is a check box in the gui that expose this functionality. The idea is that afterwards, there is another python actor that would set the proper INIT_ variables (having as many outputs as recipe parameters and putting a VariableSetter after). However this whole thing is disabled by default. If you want to enable for your interactive application, create the PipelineInteractiveApp instance like this: PipelineInteractiveApp(enable_init_sop = True).
  - passProductsThrough() is deprecated and will be removed in the future. Use set_continue_mode() instead.
- RecipeParameter class:
  - The constructor has been changed to

    ```
    __init__(self, recipe="", displayName="", name="", group="", description=""
    ```

    This means that the previous variable called "name" now has changed to "displayName", and the old "longName" has been changed to simply "name".

C.3 Python Actor

The _conversion_mode parameters are deprecated. Use RecipeParameters or SetOfFiles directly as the output of a port, it will be automatically converted to the proper JSON format.

D Porting from Reflex 2.4 to 2.5

D.1 ProductRenamer

The ProductRenamer actor has to be reinstantiated in order to add support for the new database used in the bookkeeping. The new version of this actor uses variable END_PRODUCTS_SUBDIR. Some workflows used variable END_PRODUCT_SUBDIR, and in this case it will have to be changed.

D.2 New Provenance actor

The Provenance has to be placed at the end of the workflow, so it pops up everytime a new dataset is created. The template workflow has a subworkflow at the end which implements this.

Acknowledgements

The `Reflex` team in alphabetical order consists of Pascal Ballester, Daniel Bramich, Vincenzo Forchì, Wolfram Freudling, César Enrique García, Maurice Klein Gebbinck, Andrea Modigliani, Sabine Möhler & Martino Romaniello.

References

Forchì V., Reflex User Manual, VLT-MAN-ESO-19000-5037, Issue 0.7

Horne K., 1986, PASP, 98, 609

Kaufer A. et al., UV-Visual Echelle Spectrograph User Manual, VLT-MAN-ESO-13200-1825, Issue 86

Larsen J.M., Modigliani A. & Bramich D.M., UVES Pipeline User Manual, VLT-MAN-ESO-19500-2965, Issue 14.0