ESO - EUROPEAN SOUTHERN OBSERVATORY

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

Reflex User Manual

VLT-MAN-ESO-19000-5037

Issue 3.9
20/04/2018
38 pages

Prepared:	V. Forchì		
	Name	Date	Signature
Approved:	T. Bierwirth		
	Name	Date	Signature
Released:	M. Péron		
	Name	Date	Signature

CHANGE RECORD

Issue	Date	Affected Paragraphs(s)	Reason/Initiation/Remarks	Author(s)
0.1	18/01/10	All	First version	V. Forchì
0.2	28/01/10	All	Added installation and troubleshooting sections	V. Forchì
0.3	16/03/10	All	Many corrections	V. Forchì
0.4	07/04/10	4, 5	Added Iteration	V. Forchì
0.5	28/05/10	All	Updated to beta2	V. Forchì
0.6	14/06/10	All	Added ProductRenamer chapter, minor fixes	V. Forchì, C. Garcia
0.7	29/06/10	All	Minor fixes, preparing for first public release.	V. Forchì
1.0	27/07/10	**Error! Reference source not found.**, **Error! Reference source not found.**	Parameter order, actor description	V. Forchì
2.0	16/12/10	**Error! Reference source not found.**, APPENDIX B:	Updated to Reflex 1.1 and some minor fixes	V. Forchì
3.0	05/04/12	All	Updated to Reflex 2.0	V. Forchì
3.1	16/05/12	All	Updated to Reflex 2.1	V. Forchì
3.2	12/11/12	3.2, 3.3, **Error! Reference source not found.**	Updated to Reflex 2.2	V. Forchì
3.3	15/01/13	APPENDIX B:	Updated to Reflex 2.3	V. Forchì
3.4	11/04/13	3, 4	Updated to Reflex 2.4	V. Forchì
3.5	12/11/13	All	Updated to Reflex 2.5	V. Forchì
3.8	07/08/15	All	Updated to Reflex 2.8	V. Forchì, A.Szostak
3.9	20/04/18	All	Updated to Reflex 2.9	V. Forchì, A. Szostak, L. Coccato

TABLE OF CONTENTS

1. INTRODUCTION

The ESO Recipe Flexible Execution Workbench (Reflex) is an environment which allows an easy and flexible way to execute VLT pipelines. It is built using the Kepler workflow engine (https://kepler-project.org), which itself makes use of the Ptolemy II framework (http://ptolemy.eecs.berkeley.edu/ptolemyII).
The Kepler project has thorough documentation both for the casual and experienced user (https://kepler-project.org/users/documentation).
Reflex allows the user to process his scientific data in the following steps:

- Associate scientific files with required calibrations
- Choose datasets to be processed
- Execute several pipeline recipes

This process, also called a workflow in Kepler terminology, is visually represented as a sequence of interconnected boxes (actors) that process data: the workflow allows the user to follow the data reduction process, possibly interacting with it. The user can visualize the data association and the input files and decide what scientific data he wants to process. It is also possible to visualize intermediate products, using components provided by Reflex, or modify the data flow with custom components.
Reflex uses EsoRex (http://www.eso.org/sci/data-processing/software/pipelines) to execute the pipeline recipes, but this is not exposed to the user.
This document is intended as a guide for workflow developers: if you are interested in reducing scientific data, please refer to the relative instrument workflow tutorial.

2. INSTALLATION

The prerequisite to run Reflex is the installation of EsoRex and the pipeline you are interested in (please refer to the VLT pipelines webpage for more detailed information http://www.eso.org/pipelines).
The preferred installation on OS X is using MacPorts
(http://www.eso.org/sci/software/pipelines/reflex_workflows/macports.html), there is also an installation script, that works on OS X and Linux, please refer to the following URL for more detailed instructions:
https://www.eso.org/sci/software/pipelines/reflex_workflows/#installation_procedure

2.1 Reflex installation

This step is not required if you used the installation script.

- Download the latest version of reflex for your architecture from the ESO website (http://eso.org/sci/software/reflex) or FTP server (ftp://ftp.eso.org/pub/dfs/reflex):

```
$ curl -O ftp://ftp.eso.org/pub/dfs/reflex/reflex-{version}-{linux|osx}.tar.gz
```

- Install Reflex:

```
$ cd /path/to/install/reflex
$ tar xzf /download/location/reflex-{version}-{linux|osx}.tar.gz
```

2.2 Execute Reflex

If you used MacPorts execute:

```
$ esoreflex
```

Otherwise execute:

```
$ /path/to/install/bin/esoreflex
```

2.3 Upgrading from previous Reflex versions

Reflex 2.9 changes some bookkeeping procedures, therefore it cannot reuse bookkeeping directories and databases generated from previous Reflex versions. It is therefore recommended to clear such directories, please refer to the workflow user manual for more information about the location of the bookkeeping directories.

3. REFLEX OVERVIEW

3.1 General concepts

Kepler visually represents a workflow as a sequence of actors with a single director: the latter schedules the execution of the actors and the former manipulates the data.
The main components of a workflow are:

- Director: determines the execution order of the actors and tells them when they can act. There are several director types, the default for Reflex is the Dynamic Dataflow (DDF) director. There must only be one director per workflow.
- Actor: represents a single step of execution. The actor takes data from the input ports, processes them and sends the results to the output ports. Each actor can have a number of parameters that control its execution: to edit the parameters double click on the actor.
- Port: each actor can possess one or more ports, which allow it to exchange data with other actors. Ports can be input-only (a triangle pointing into the actor), output only (a triangle pointing out from the actor), or bidirectional (a circle). A port can be singular or multiple: the former can be connected to only one port, the latter can be connected to many. Singular ports are black, multiple ports are white.
- Token: an object that encapsulates data. Actors exchange data in the form of tokens through ports. A token can contain several types of data: integers, strings, floating point, etc.

3.2 Reflex specific actors

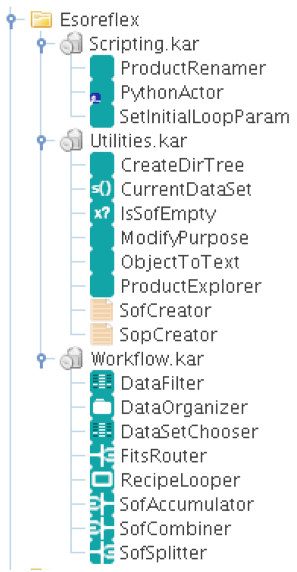
Reflex is composed of a number of custom actors that allow the workflow to interact with the VLT pipeline and with FITS files, namely:

- CreateDirTree
- CurrentDataSet
- DataFilter
- DataOrganizer
- DataSetChooser
- FitsRouter
- IsSofEmpty
- ModifyPurpose
- ObjectToText
- ProductExplorer
- ProductRenamer
- PythonActor
- RecipeExecuter
- SetInitialLoopParam
- SofAccumulator
- SofCombiner
- SofCreator
- SofSplitter
- SopCreator

All these actors, except for the RecipeExecuter, can be inserted into the workflow from the component tree on the left hand side of the Reflex window, in the Esoreflex sub-menu (Figure 1).
The RecipeExecuter is not present in the menu and must be instantiated from the menu bar: click on Tools -> Instantiate Component, change the Class name to org.eso.RecipeExecuter and click OK (Figure 2). You will then be presented with a combobox containing all the available CPL recipes (the list is obtained from the output of the command "`esorex --recipes`", as configured by the esoreflex startup script): select the

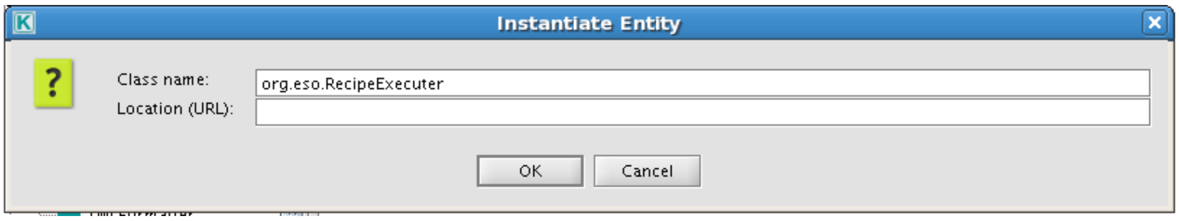
desired one and click OK (Figure 3). Once a particular recipe has been chosen it cannot be changed: if you chose the wrong one then you have to remove the actor and instantiate a new one.
In the same component tree that contains Reflex actors you can find many other actors, which are part of the standard Kepler distribution; a detailed description can be found in the Kepler User Manual, but they mostly solve general needs, such as:

- Mathematical operations (arithmetic, statistical, logic...)
- File manipulation (open, read, write...)
- File system interaction (list directory content, remove directory...)
- Command execution (shell, ssh, condor...)
- Workflow control (switch, pause, stop...)



Esoreflex
Scripting.kar
ProductRenamer
PythonActor
SetInitialLoopParam
Utilities.kar
CreateDirTree
CurrentDataSet
IsSofEmpty
ModifyPurpose
ObjectToText
ProductExplorer
SofCreator
SopCreator
Workflow.kar
DataFilter
DataOrganizer
DataSetChooser
FitsRouter
RecipeLooper
SofAccumulator
SofCombiner
SofSplitter

Figure 1: ESO specific actors

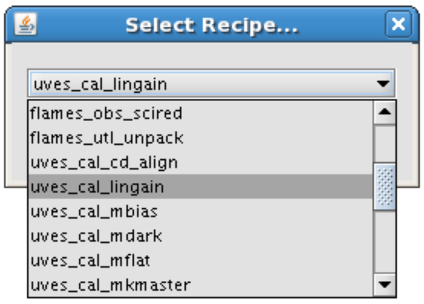


Instantiate Entity

Class name: org.eso.RecipeExecuter
Location (URL):

OK Cancel

Figure 2: How to instantiate the RecipeExecuter



Select Recipe...

uves_cal_lingain

flames_obs_scired
flames_utl_unpack
uves_cal_cd_align
uves_cal_lingain
uves_cal_mbias
uves_cal_mdark
uves_cal_mflat
uves_cal_mkmaster

Figure 3: Recipe selection

3.3 Data types

Reflex specific actors exchange data in the form of Datasets, Set of Files (SoF) and Set of Parameters (SoP): they are all represented as a JSON structure and are not meant to be read by a human. Use the dedicated actor ObjectToText to visualize them.

- Dataset: contains some general information and a tree structure that describes the calibration cascade.
- SoF: contains a set of science frames and associated calibration files which are required to process them
- SoP: contains the values of the parameters for the execution of the pipeline recipes.

3.4 General workflow structure

The general structure of a Reflex workflow is depicted in Figure 4. On the top there are a number of parameters required for the workflow execution:

- RAWDATA_DIR: a directory containing all the raw data to be processed (Note: subdirectories are recursively scanned)
- BOOKKEEPING_DIR: a directory where each pipeline recipe execution will create a subdirectory to use as a working directory
- BOOKKEEPING_DB: a file containing a SQLite database that stores information about the actor executions
- LOGS_DIR: a directory where the recipe logs will be saved
- TMP_PRODUCTS_DIR: a directory where the workflow intermediate products will be saved
- END_PRODUCTS_DIR: a directory where the workflow final products will be saved
- FITS_VIEWER: executable used to visually inspect FITS files
- ESORexArgs: additional parameters passed to esorex by the RecipeExecuter

Note: all the aforementioned parameters cannot contain spaces in the directory names, otherwise the workflow will fail.
The data under each directory, except the one containing the final products, are organized in the following way:
BASE_DIR/ActorName/ExecutionTime
e.g. /home/reflex/reflex_tmp_products/Uves_Blue/uves_cal_mbias_1/2010-01-01T12:23:12.123 contains the products of the execution of the actor named uves_cal_mbias_1 at the timestamp “2010-01-01T12:23:12.123”.
If you double click one of these parameters you will be presented with a configuration window, where you can select the value of the parameter, either by typing it or browsing the filesystem (do not forget to save your modifications, workflows contain hardcoded paths and in general don’t work if the underlying filesystem changes – e.g. you send it to somebody).

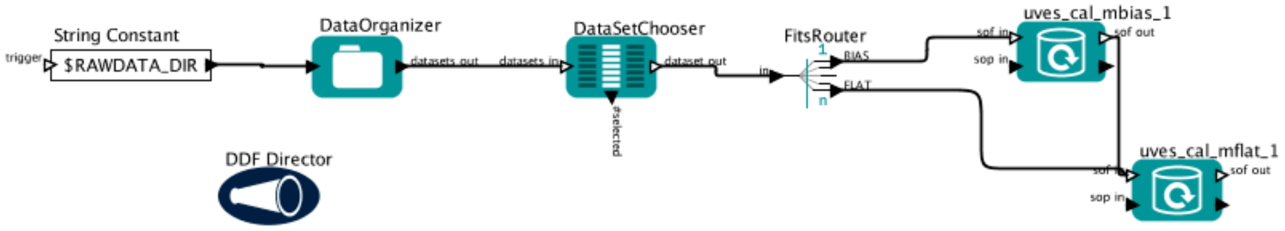


Figure 4

The preferred director for Reflex workflows is the DDF director, but the PN director can also be used, although it is not officially tested. The first step of the workflow execution is the DataOrganizer, which organizes a set of files and groups them together in Datasets according to some classification and grouping rules, written with the OCA language (a full set of rules is provided together with each workflow): the output is a list of Datasets, which contain science frames and all the associated raw and static calibrations required to process them.

The second step is the DataSetChooser, which displays all the Datasets provided by the DataOrganizer, and allows the user to view their contents and select the ones that are to be processed. The output of this actor is the serialized list of selected Datasets. The output of the DataSetChooser must be connected to a FitsRouter, which splits the Dataset and sends the files to different output ports based on their observation type: these ports can then be used to feed the various pipeline recipes of the workflows.

3.5 File purpose

The file purpose is a new concept introduced in Reflex 2 that represents what a file is needed for; the purpose of a file describes the whole reduction cascade that will make use of the file. For example, a raw BIAS might have the 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, or MASTER_BIAS/MASTER_FLAT/SCIENCE, if the master bias will be used to generate a master flat, which in turn will be used to reduce the science frames.
This new concept allows Reflex to recreate the whole calibration cascade, for instance we might have different master biases in different points of the workflow, produced with different recipe parameters.
The purpose is used in many places of the workflow to decide the routing and the scheduling of files, for example:

- The SofSplitter groups the files by purpose and emits a number of tokens equal to the number of different purposes present in the input SoF;
- The SofCombiner, and any input multiport that accepts SoFs as an input, combine the input SoFs based on their purpose, collecting only those files whose purpose is present in all input ports.

In order to help the users in the process of customizing the workflow, e.g. by plugging in their own reduction steps or precomputed files, a special purpose has been defined, named UNIVERSAL, that matches any other purpose when it comes to combining or splitting SoFs.

3.6 General features

- Workflow execution: you can start, pause, stop and resume the workflow by using the buttons in the toolbar. The highlighted button indicates which state the workflow is in. Please note that the stop button immediately interrupts any running pipeline recipe, while the pause button lets the current recipe or actor finish before the workflow is actually paused. After pressing the pause button, it is also possible that more than one actor is executed, since this behaviour depends on the scheduling policy. For instance, if there are two actors in parallel, and you pause the workflow while one is being executed, then both of them will be executed before the workflow is actually paused.
- The DDF and the PN directors support actor highlighting; this feature is disabled by default: if you want to enable it click on Tools->Animate at Runtime, select an interval (e.g. 10ms) and click ok. From now on the active workflow actor will be highlighted in red. Note: if you pause and resume a workflow the actor is not highlighted upon resume.

4. ACTOR DOCUMENTATION

The documentation of the Reflex actors is available from the workflow, by right clicking on an actor and selecting Documentation->Display. RecipeExecuter instances also display the help of the recipe parameters.

5. RECIPE ITERATION

It is sometimes useful to be able to visualize the products of a recipe execution, tweak some recipe parameter and execute the recipe again, until the products are as expected.
This can be easily achieved in Reflex by means of the RecipeLooper and the PythonActor.
A typical subworkflow that allows iteration is depicted in Figure 5, and it is composed of the following elements:

- Sof coming from previous actor
- Sop containing recipe parameters' initial values
- A RecipeLooper
- A RecipeExecuter containing the pipeline recipe to be optimized
- A PythonActor containing a custom python script that allows the user to view the results of the recipe execution and decide whether he wants to change some parameters or not. The script should either generate a "true" token on the control port and a SoF to the next downstream actor or a "false" token on the control port and a SoF and a SoP to the RecipeLooper loop input port.

This general structure must be customized for each pipeline recipe. The user must:

- Identify sensible recipe parameters he wants to tweak to optimize the products
- Define some initial conditions for these parameters and provide them to the sop_in port of the RecipeLooper. The simplest way to do this is to define a string in the format described in APPENDIX A: and connect it to a SopCreator.
- Configure the RecipeExecuter, changing the value of the recipe parameters you want to optimize to PORT (e.g. if you want to optimize a recipe parameter called `par1` look for a parameter in the RecipeExecuter called `recipe_param_nn` whose value is `par1=some_value` and change it to `par1=PORT`).
- Write a python script that allows the user to evaluate the product quality, change the value of the parameters and decide whether he wants to continue or not. The script does not have to be interactive, it can implement an optimization algorithm defined by the user.

Sample implementations of this system are provided with the workflows included in the pipeline distribution.
Note: it is possible to iterate over an arbitrary number of actors, you are not forced to iterate over one RecipeExecuter.

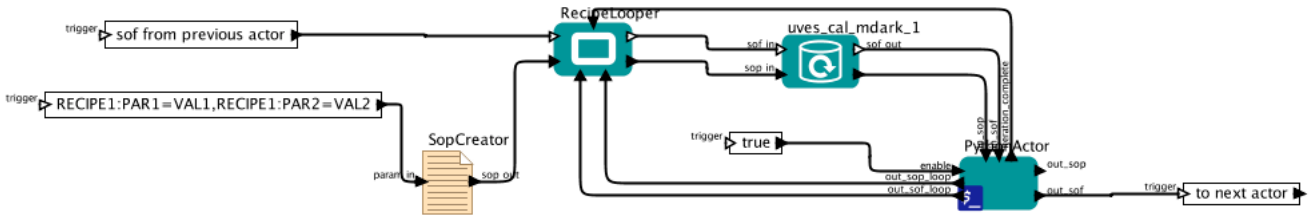


Figure 5: sample looping workflow

6. PYTHON BASED RECIPES

EsoReflex is now able to support Python based recipe plugins that are executed through the normal EsoRex interface. To take advantage of this feature, you need to have EsoReflex 2.9.0 or newer installed, together with an appropriate version of EsoRex (3.13 or newer).

EsoRex must however have been built with Python based recipe support enabled, i.e. the necessary 3rd party dependencies must be installed (*libffi* or *libavcall.a*) and the *--enable-python-recipes* build option needs to be used if not enabled by default. Please refer to the EsoRex software documentation for further compilation and installation details.

RPM and MacPorts based packages of EsoRex as delivered by ESO should already have EsoRex 3.13 correctly built with Python recipe support enabled. You can simply follow the normal package installation procedure for EsoRex, if installing RPMs or MacPorts packages.

6.1 Configuring EsoRex

Python enabled EsoRex will search for Python based recipes in its configured recipe plugin directory list. The canonical method of configuring the plugin directories is to modify the system wide *esorex.rc* file[1] and add the directory where the new Python recipes are located to the *esorex.caller.recipe-dir* parameter. This parameter accepts a colon separated list of paths to search.

It is also possible to create a custom configuration file. In such a case, both a custom *esorex.rc* file needs to be created for customising the *esorex.caller.recipe-dir* parameter and a custom *esoreflex.rc* file must be created to tell EsoReflex to use the updated *esorex.rc* file for EsoRex. The easiest way to create both configuration files is to run the following commands:

```
esorex --create-config
esoreflex -create-config
```

This will create the two files *~/.esorex/esorex.rc* and *~/.esoreflex/esoreflex.rc* for EsoRex and EsoReflex respectively. The configuration file for EsoReflex must be modified by changing the *esoreflex.esorex-config* parameter. For example, the updated line in the *~/.esoreflex/esoreflex.rc* file should read:

```
esoreflex.esorex-config=~/.esorex/esorex.rc
```

In the *~/.esorex/esorex.rc* file the line containing *esorex.caller.recipe-dir* can be updated as follows:

```
esorex.caller.recipe-dir=/usr/lib64/esopipes-plugins:~/myrecipes
```

In the above example we have assumed that *esorex.caller.recipe-dir* previously contained the path */usr/lib64/esopipes-plugins* and that our desired location for Python recipes is *~/myrecipes*.

Note that if the default paths for the configuration files are not suitable, then different paths can also be used with the configuration creation commands as follows:

```
esorex --create-config=myesorex.rc
esoreflex -create-config myesoreflex.rc
```

However, in this case the *esoreflex* command will have to be explicitly invoked with the configuration file's path as follows:

[1]The *esorex.rc* configuration file's path is *<prefix>/etc/esorex.rc*, which defaults to */etc/esorex.rc* for typical system wide installations, e.g. RPM installations.

```
esoreflex --config myesoreflex.rc
```

For advanced users, if the *esoreflex.inherit-environment=TRUE* option is used in the EsoReflex configuration, then the recipe directory can also be controlled with the *ESOREX_PLUGIN_DIR* environment variable.

6.2 Writing Python Recipes

Developing a Python based recipe follows the conventional methodology as you would expect when writing any other Python code. The only strict requirements are,

- The source code file name must only contain characters that form valid symbol names in Python and end with the *.py* extension.
- The source file must reside in one of the recipe plugin directories that EsoRex searches in.
- Each plugin must be a class derived from *esorexplugin.RecipePlugin* and implement the appropriate API interface.

6.2.1 Python Imports

Inside the Python module file that implements the recipe, the following imports must be used:

```
import esorexplugin
from esorexplugin import *
```

This will make the *esorexplugin.RecipePlugin* class and various additional helper functions available. Of course, any needed additional imports can also be added.

You should not import the *RecipePlugin* class directly into your own Python code, since this will cause EsoRex to mistaken it for a recipe implementation, i.e. do not use the following import syntax “from esorexplugin import RecipePlugin”.

6.2.2 Recipe Declaration

The actual recipe class should be declared as follows:

```
class MyRecipe(esorexplugin.RecipePlugin):
```

Note that the name of the new class does not matter. In addition, if an *__init__* constructor is required then it must not require any additional mandatory arguments, except for *self*.

The recipe class needs to declare a number of class level attributes to configure the recipe parameters and various other properties. These include:

- **name** – (string)
  The name of the recipe as will be used in EsoReflex and EsoRex. If not provided then the Python class name will be used instead.
- **version** – (integer)
  The version number of the recipe. This is best configured using the *VersionNumber* helper function, which takes three numbers, major, minor and patch numbers to construct the appropriately encoded value. If this is not provided then 0 is used by default
- **synopsis** – (string)
  A short descriptive text for the recipe. If this is not provided then the first line from the class’s docstring will be used instead.

- **description** – (string)
  A more detailed description of the recipe that is used by EsoRex to generate the man-page. If this is not provided then the text in the class's docstring will be used instead.
- **author** – (string)
  The name of the recipe's author.
- **email** – (string)
  The author's email address.
- **copyright** – (string)
  An appropriate copyright message relevant for the recipe's code. A default GNU version 2 license text can be generated with the *EsoCopyright* helper function.
- **parameters** – (list)
  This must be a list of recipe parameter configurations, one for each recipe parameter that the recipe can accept. It is best to declare each entry of the parameters list using one of the helper functions *ValueParameter*, *RangeParameter* or *EnumParameter*. This can be an empty list if the recipe should not accept any parameters.
- **recipeconfig** – (list)
  An optional list of frame tags indicating the minimum and maximum number of expected frames, including the input and output inter-dependencies with other frame types. Each entry should be configured with the *FrameData* helper function.

It is also possible to configure and assign these attributes in the *__init__* constructor instead of declaring them at the class level. Remember that the *RecipePlugin.__init__* constructor should be invoked in any custom recipe constructor as follows:

```
def __init__(self):
  super(MyClass, self).__init__()
  ...
```

6.2.3 Recipe Parameters

Any input parameters that the recipe should accept must be declared in or appended to the *parameters* attribute. Each entry can be constructed with an appropriate helper function. Selecting the correct helper function to use depends on the type of parameter needed, which must be one of the following:

- **ValueParameter(name, default ...)**

This is used for a parameter that accepts any value that is only constrained by a type. The name of the parameter must be given as a string in the first argument to the function. A default value must also be given as the second argument. The type of the parameter is defined by the type of the default value. For example, if the default value is an integer then the recipe parameter will be of type integer; for a string default, the recipe parameter becomes a string also. The allowed types for the default value are: boolean, integer, float or string. The following is a minimal example for configuring a value parameter:

```
parameters = [ValueParameter('par1', 3)]
```

- **RangeParameter(name, default, min, max ...)**

This is used to create a parameter that only allows a value within a certain range. It takes a name and default argument just like for *ValueParameter*, and two additional arguments to specify the minimum and maximum allowed value of the range. Only integers or floats are allowed for range parameters and both *min* and *max* must have the same type as *default*. The following is a minimal example for configuring a range parameter:

```
parameters = [RangeParameter('par1', 3, 1, 5)]
```

- **EnumParameter(name, default, choices ...)**

This allows configuring a recipe parameter that can only take on values from a selected set of predefined choices. It takes a name an default argument just like *ValueParameter*, and an additional list of values that form the allowed set of choices as the third argument. The *default* value must also be within the *choices* list. In addition, every element of the *choices* list must have the same type as *default*. The allowed types are: integer, float and string. The following is a minimal example for configuring an enumeration parameter:

```
parameters = [EnumParameter('par1', 'A', ['A', 'B', 'C'])]
```

Each of the above helper functions also accepts additional optional keyword arguments. For a detailed listing, please refer to the online python documentation for these functions. However, the following three will be mentioned here, since they are recommended to be used.

- **description** – (string)
  Provides a textual description of the recipe parameter used in the EsoRex generated man-page. This will also be forwarded to any connected PythonActors, where it can be used in the user interface.
- **context** – (string)
  Assigns a context string to the recipe parameter so that tools invoking the recipe can better organise the parameters from many different recipes. Typically this can simply be set to the recipe name, plus perhaps a dot separated sub-context.
- **displayName** – (string)
  This allows to set an automatic short name associated with the recipe parameter, which will be used by the PythonActor in the user interface by default. If this option is used then you will not need to use the corresponding *displayName* option in the PythonActor script, when configuring recipe parameters for display in the user interface. Although one can still nevertheless use *displayName* again in the PythonActor script to override the recipe's settings.

Here is an example of using the above three additional arguments:

```
ValueParameter('test.par1', 3, description='my integer parameter 1',
          context='test', displayName='par1')
```

6.2.4 Recipe Configuration

There is an optional feature available to provide additional meta-data with regards to the input frames that a recipe can handle and the inter-dependencies between these. This is specified as a list of meta-data objects created with *FrameData* in the *recipeconfig* class attribute. The *FrameData* helper function accepts the following five arguments:

- **tag** - (string)
  A tag string that identifies a frame. This should be the tag associated to a frame as is normally given in the Set-of-Frames.
- **min** - (integer)
  The minimum number of frames of type *tag*. This is set to 1 by default. It is also possible to set this to

None, which indicates that this value is not specified. Alternatively, use 0 to indicate that the frame is optional.

- **max** - (integer)
  The maximum number of frames of type *tag*. This is set to 1 by default. If you want to mark this value as unspecified then set it to *None*.
- **inputs** - (list)
  A list of additional input frames needed for processing a frame indicated by *tag*. The entries of this list must also be constructed with the *FrameData* helper function. However, only the *tag*, *min* and *max* arguments have any meaning for the additional input frames. Only those arguments should be used.
- **outputs** - (list)
  This must be a list of tag strings of the corresponding output frames produced by the recipe.

The following is an example of how the *recipeconfig* attribute can be constructed with *FrameData*:

```
recipeconfig = [
    FrameData('RAW', min = 1, max = None,
         inputs = [
           FrameData('STATIC'),
           FrameData('CALIB', min = 0, max = 1)
         ],
         outputs = ['PROD'])
  ]
```

The above example configuration indicates that the recipe expects at least one (min = 1) or more (max = None, i.e. upper limit unspecified) “RAW” frames to process. “RAW” would be the tag of the input FITS file as encoded in the Set-of-Frames. Processing a “RAW” frame will produce output frames of type “PROD”. “RAW” frames also require an additional input frame called “STATIC” and a single optional “CALIB” frame (min = 0 and max = 1).

The configuration structure produced by the *FrameData* function ultimately corresponds to the *cpl_recipeconfig* object. The purpose of this structure is to provide information about the frames that are required by the recipe to the higher layers of the overall Data Flow System (DFS). You can refer to the *cpl_recipeconfig* documentation from CPL for more details about recipe configurations.

Note that EsoReflex does not use this information in any special manner at the moment. However, by using the *recipeconfig* attribute, the recipe itself will be declared as a version 2 CPL recipe plugin to EsoRex. Without *recipeconfig* present, the recipe defaults to a version 1 plugin.

6.2.5 Grouping Input Frames

Every properly written recipe needs to update the meta-data for the input frames that it receives, to indicate what type, group and processing level they belong to. This information is then propagated back to EsoRex. The updating of the input frames should be coded in the mandatory *set_frame_group* method, which takes a single argument, the frame object. Frame objects have the following attributes: *filename*, *tag*, *type*, *group* and *level*. The *filename* will contain the path to the FITS file on disk. The *tag* will be the tag string as given to the recipe in the input Set-of-Frames. All other attributes will be set to default “none” values and should be updated by *set_frame_group* appropriately.

Currently EsoRex uses the *level* value to manage the final destination of files produced by the recipe and to update symbolic links needed by the DFS. The *group* attribute is used to categorise the FITS files for statistical information and also to control which FITS files have their *PIPEFILE*, *CHECKSUM*, *DATASUM* and *DATAMD5* keywords updated. However, the *type* attribute is not actually used in any special way by EsoRex at the moment and could be set to the default value *Frame.TYPE_ANY*.

In principle EsoReflex does not use the updated frame information in any way. It is meant primarily for the DFS. If one really does not want to update any of this information then one can simply write the *set_frame_group* method as follows:

```
def set_frame_group(self, frame):
   pass
```

For further details about valid values that can be assigned to the attributes see the online Python documentation for the *Frame* class.

6.2.6 Processing Frames

Actual processing of data should be implemented in the mandatory *process* method. It must take as arguments the list of input frames, followed by arguments for each input recipe parameter. If no recipe parameters have been configured then only the list of frames needs to be specified, for example:

```
def process(self, frames):
   ...
```

For two recipe parameters one would write the following instead:

```
def process(self, frames, par1, par2):
   ...
```

The names of the arguments for the recipe parameters can be anything, and need not correspond to the names as configured in the *parameters* class attribute. They will be assigned in the same order as they are declared in the *parameters* list, i.e. the first configured parameter will be assigned to the first argument after *frames*, the second entry in *parameters* will be assigned to the second argument after *frames* and so on. When the recipe is invoked by the RecipeExecuter, the input Set-of-Frames will be represented in the *frames* argument as a list of *Frame* objects, while the values for the recipe parameters will be passed from the RecipeExecuter into the additional arguments of the *process* method. Detailed meta-data for each input recipe parameter can also be accessed from the *self.input_parameters* attribute if needed. This will be a list of *RecipeParameter* objects sorted in the same order as the recipe parameters were configured in the *parameters* attribute.

Inside the *process* method you can open the associated FITS file with the *Frame* object's *open* method as follows:

```
def process(self, frames, ...):
   hdulist = frames[0].open()
   ...
```

This will return a HDU list object that can be further manipulated with the Astropy API[2]. Any new output frames must be created as *Frame* objects and returned by the *process* method as a list. The *write* helper

[2]For old platforms this would be the Pyfits API instead.

method can be used to write a corresponding HDU list to the output FITS file associated with the output frame. For example:

```
def process(self, frames, ...):
   ...
   outframe = Frame('output.fits', 'PROD', type=Frame.TYPE_IMAGE)
   outframe.write(hdulist)
   return [outframe]
```

Note that the *write* method is a wrapper around the Astropy API *writeto* method. Thus, both methods accept the same set of additional arguments. In particular, the *overwrite* option may be useful, since the FITS files will not be overwritten by default. For example:

```
outframe.write(hdulist, overwrite=True)
```

6.2.7 Raising Errors

If an non recoverable error occurs in the recipe that must cause a fatal exception, you should use the *raise_error* method. This will raise a Python exception and abort further processing of data. An error message and error code will be forwarded back to EsoRex.

The *raise_error* method accepts three arguments, the error message to send (a string), the corresponding error code (an integer), and a boolean indicating if a traceback of the Python stack where the exception was raised should be printed to standard error. If an error code is not specified then a value of 1 is used by default. Also the traceback is not printed by default.

The following are examples of using the *raise_error* method:

```
def process(self, frames, ...):
   ...
   self.raise_error('Something went wrong')
   self.raise_error('Error with traceback', 123, print_traceback=True)
   ...
```

The value for the error code can be any non-zero integer. However, CPL error codes are typically returned by recipes. Please refer to the CPL documentation for a listing of available error codes.

6.2.8 Online Documentation

Additional online documentation for the Python recipe API can be viewed in the Python interpreter shell by running the following commands:

```
import esorexplugin
help(esorexplugin)
help(esorexplugin.RecipePlugin)
help(esorexplugin.RecipeParameter)
```

6.3 Example Code

A minimal working example of a Python based recipe can be found in the EsoReflex installation directory, under *<prefix>/esoreflex/python/example_recipe.py*. For example, with RPM installations this should be found under */usr/share/esoreflex-2.9.0/esoreflex/python/example_recipe.py*. The recipe takes one input frame and simply copies it to an output FITS file “output.fits”. It also accepts the following three parameters:

- test.par1 – a boolean indicating if the output file should be overwritten.
- test.par2 – a dummy integer range parameter, who's value will be printed to the console.
- test.par3 – a dummy string enumeration parameter, that will also be printed.

To try the recipe example, make sure that Astropy is installed and that it can be imported by the Python interpreter used by EsoReflex (i.e. "import astropy" works in *python*). Copy the example file to a recipe directory that EsoRex will search in. See the EsoRex configuration section for details about how to configure the recipe directory. You can then instantiate a RecipeExecuter actor and the example recipe called "test" should appear in the drop down menu.

Note that when attempting to use the recipe from the command line with the *esorex* command, the Python interpreter must be told where to find the *esorexplugin.py* module by setting the *PYTHONPATH* environment variable appropriately, e.g. *PYTHONPATH=<prefix>/esoreflex/python*. When working within EsoReflex, this is normally already taken care of in the *esoreflex.rc* file with the *esoreflex.python-path* parameter. But this is not the case when using the *esorex* command directly.

6.4 Using Python Recipes

Once a Python based recipe has been written, it behaves exactly the same way as a compiled recipe plugin from within EsoReflex. They are instantiated in the same manner. Please refer to the section about instantiating the RecipeExecuter for details.

6.5 Debugging

The most straight forward procedure for entering a debugging session when trying to debug a Python based recipe, is to first attempt to invoke the recipe as normal within EsoReflex. This will leave a *cmdline.sh* file under the bookkeeping directory, which contains everything needed to run the recipe from the command line in exactly the same way that EsoReflex ran it. Before running *cmdline.sh*, you should add the following statement to the Python recipe code:

```
import ipdb; ipdb.set_trace()
```

This break point should be added in the place where you want to start debugging, i.e. the source code location where the program should stop and enter the debugger. If *ipdb* is not available on your system, you can fall back to using *pdb* as follows[3]:

```
import pdb; pdb.set_trace()
```

In the case of *example_recipe.py* mentioned in section 6.3., the break point statement can for example be added to line 114 in that file.

With the break point set, you can then launch the *cmdline.sh* script. This will run EsoRex, launch the Python interpreter and stop at the break point within the debugger's shell. The code can then be stepped through, additional break points set and the variables explored. For details about how to use the *ipdb* or *pdb* shell commands, you can type the "help" command in the shell or refer to the relevant documentation for those tools[4].

[3]Note that the *pdb* debugger has a more primitive shell than *ipdb*. Thus, *ipdb* is usually preferred.
[4]For *pdb* refer to https://docs.python.org/2/library/pdb.html and *ipdb* to https://pypi.python.org/pypi/ipdb. The commands and usage of *ipdb* is mostly the same as for *pdb*.

In some cases, it may be useful to see more details about the interaction between EsoRex and the Python interpreter. This can be enabled by changing the *esorex.caller.msg-level* parameter in the EsoRex configuration file. For example, *esorex.rc* should have the following line changed to:

```
esorex.caller.msg-level=debug
```

7. EXECUTING IDL SCRIPTS

IDL scripts can easily be launched by esoreflex trough a Python Actor. If the IDL executable or the needed scripts are not in the system path, then their location have to be hardcoded in the Python script associated to the Python Actor. The IDL script has to accept inputs and outputs filenames from the command line.

We provide below an example of a Python script (python_script.py) that is designed to launch an IDL script, and the IDL script itself (myidlscript.pro)

```
#!/usr/bin/env python
import reflex
import sys
from astropy.io import fits
from optparse import OptionParser
import json
import os
if __name__ == '__main__':

  parser = reflex.ReflexIOParser()
  #Define inputs/outputs
  parser.add_option("-i", "--in_sof", dest="in_sof")
  parser.add_output("-o", "--out_sof", dest="out_sof")
  inputs  = parser.get_inputs()
  outputs = parser.get_outputs()
  in_sof = inputs.in_sof
  files = in_sof.files


  #get the name of the output directory
  pattern = '--products-dir'
  infile=''
  for arg in sys.argv:
     if arg.split("=")[0] == pattern:
         output_dir = arg.split("=")[1]

  #Create a list of input files to pass to the IDL script
  for file in files:
     infile = infile + '   ' + file.name
  infile= '"'+infile+'"'

  #define the list of products and their categories
  # (2 files, as example)
  output_names=''
  output_filenames=list()
  output_catgs=list()
  for n in range(1,3):
    output_filenames.append(output_dir+'/output'+str(n)+'.fits')
    output_catgs.append('CATEGORY'+str(n))
    output_names=output_names+' '+output_dir+'/output'+str(n)+'.fits'
  out_str= '"'+output_names+'"'

  # If the idl script (myidlscript.pro) is not in the system library path,
  #I need to hardcode its location into the script
  os.environ["IDL_PATH"] = ":+/scisoft/share/idl/idl/lib/: /scratch/lcoccato/data/astron/"

  # If the IDL executable is not in the system path, I need to hardcode it in the script.
  os.system("bash -c '/scisoft/bin/idl -e myidlscript -args "+infile+" "+out_str+"'")

  # create output SOF:
  files = list()
  output_purpose=file.purposes
  output_datasetname=in_sof.datasetName
  #Outputs are 2
```

```
    for i in range(2):
       hdu=fits.open(output_filenames[i])
       csum=hdu[0].header['CHECKSUM']
       hdu.close()

files.append(reflex.FitsFile(output_filenames[i],output_catgs[i],csum,output_purpose))
    newsof = reflex.SetOfFiles(output_datasetname,files)
    outputs.out_sof = newsof

# broadcast output:
    parser.write_outputs()
    sys.exit()
```

We provide below an example of IDL script that is called by the python script. In the example, the IDL script accepts a list of input files and generates 2 outputs. The names of the inputs and outputs are determined by the python script and passed to IDL via command line.

```
pro myidlscript
    n_outputs = 2
    args=command_line_args()

    ;GET names of inputs and outputs
    input_names = strsplit(args[0:n_elements(args)-n_outputs],' ',/extract)
    output_names=  strsplit(args[n_elements(args)-(n_outputs+1)],' ',/extract)

    for i = 0, n_elements(input_names)-1 do print, input_names[i]
    for i = 0, n_elements(output_names)-1 do print, output_names[i]

    ;creation of first output (e.g., average of input files)
    average=0
    for i = 0, n_elements(input_names)-1 do begin
       average += readfits(strcompress(input_names[i],/remove_all),h,/silent)
    endfor
    average=average/n_elements(input_names)
    writefits,output_names[0],average,h,/CheckSum


    ;creation of second output (e.g., stddev of input files)
    rms=0
    for i = 0, n_elements(input_names)-1 do begin
       data = readfits(strcompress(input_names[i],/remove_all),h,/silent)
       rms += (average - data)^2
    endfor
    rms=sqrt(rms/(1.-n_elements(input_names)))
    writefits,output_names[1],rms,h,/CheckSum

 end
```

8. ESOREFLEX STARTUP SCRIPT

ESO-Reflex is executed by means of a startup script, named esoreflex, that help the user in setting up the proper environment for the execution of ESO pipelines and python scripts. This chapter describes all its features and configuration options.

8.1 Command line options

The *esoreflex* launch command can be customised with many optional command line arguments. Many of these are related to the underlying Kepler framework and will not be described here. Details can be found in Keplers online help. Here we describe the most important command line arguments introduced specifically by *esoreflex*.

8.1.1 -h | -help

This prints a help/usage message and stops. The message will contain a listing of command line options that can be given to the *esoreflex* launch command.

8.1.2 -v | -version

Prints version information and exits. The version of *esoreflex* being invoked is printed, including a list of pipelines that *esoreflex* knows about and each pipeline's version number.

8.1.3 -l | -list-workflows

This will list all installed workflow XML and KAR files that can be found and are available to run with *esoreflex*. These will be all the workflows found in a subdirectory under one of the paths set in the *esoreflex.workflow-path* variable. This variable can be configured in the system wide *esoreflex.rc* or private (per user) configuration file.
The format of the output will be a table with a short format name for each workflow and the full path to the workflow file. If multiple files have the same name, the short name will be prefixed with a part of the path to make each short name in the list unique. In such a case, the short names may not always point to the same file if there is a configuration change of *esoreflex*, the system or a filename change. It is best not to rely on the short name to uniquely identify the workflow, but rather the full path to the file should be used. The short names are only for convenience when working with *esoreflex* interactively on the command line.
Either the full file path or the short name as listed with the *-l | -list-workflows* option can be used in the *esoreflex* command to open or run the specified workflow directly. For example:

```
esoreflex fors_spec
```

8.1.4 -n | -non-interactive

This option enables non-interactive features of *esoreflex* to be used for batch and non-interactive execution of workflows. When used, additional options are passed to *esoreflex* and the Kepler framework to prevent the graphical user interface from showing and to automatically execute the workflow. You will need to pass the path to the workflow XML/KAR file or use the short name for the workflow as known to the *esoreflex* launch command (listed with the *-l | -list-workflows* option). For example:

```
esoreflex -n fors_spec
```

8.1.5 -config <file>

Use this option to force the *esoreflex* launch command to use a custom configuration file, rather than the system one or *~/.esoreflex/esoreflex.rc* if that exists. The option takes one argument, the name or path to the configuration file. If this option is used then any variables in either the system wide or

~/.esoreflex/esoreflex.rc files will be completely ignored. Only variables defined in the file given by *<file>* are used, and for any that are missing, an internal default value is used instead.

8.1.6 -create-config <file>

Generates a new configuration file and exits. This option is convenient to create a new working configuration file that can subsequently be modified by the user, rather than creating a configuration file from scratch. The option takes one argument, *<file>*. If the argument is set to the special value *TRUE* then a new configuration file is created in the user's home directory under *~/.esoreflex/esoreflex.rc*. Otherwise a file name or path must be given for *<file>*, in which case a new file will be written to that location. Any existing file is backed up to a file with a *.bak* extension, or *.bakN* where *N* is an integer, if a backup copy also already exists.

8.1.7 -debug

This prints additional information that can be useful for debugging problems with the *esoreflex* configuration. The following information is printed before invoking the actual *esoreflex* binary:

- The path to the configuration file used by the launch command.
- All environment variables as seen by the *esoreflex* binary at start time.
- The full paths to the *java*, *esorex* and *python* binaries if found.
- The full Java command used to actually invoke the *esoreflex* program.

8.2 The esoreflex.rc file

Reflex primarily relies on three binary programs to work: *java*, *esorex* and *python*. To function properly, Reflex must know where these programs are located and how they should be invoked. This can be configured within a configuration file, called *esoreflex.rc* by default.
Normally when using the Reflex tarball package directly, no configuration file is provided. Reflex will simply use some internal default values that are reasonable guesses. However, system installations of Reflex will have a default system wide *esoreflex.rc* file created. Alternatively, a per user configuration file can be placed in the following location in the home directory *~/.esoreflex/esoreflex.rc* or a file path can be passed to the *esoreflex* command with the *-config* option, e.g. *esoreflex -config myconfig.rc*
The configuration file provided in *-config* will take the highest precedence. If no configuration is given explicitly on the command line then the file *~/.esoreflex/esoreflex.rc* is used if it exists. If it does not exist then the system wide configuration is used, if that exists. This will usually be */etc/esoreflex.rc*, but might be placed in a different location depending on the platform. If no configuration file is provided anywhere then the internal default values are used. To summarise, the search order for the configuration file/variables is, starting from the highest precedence to lowest:

1. File given by the command line *-config* option.
2. *~/.esoreflex/esoreflex.rc*
3. System wide *esoreflex.rc*, usually under */etc/esoreflex.rc*.
4. Internal default values.

Be aware that any variables that are not specified in your custom configuration file will use the default internal values and not the values from the system wide configuration file. For example, if there exists a system */etc/esoreflex.rc* file and you pass *myconfig.rc* to *esoreflex* that does not contain the *esoreflex.python-path* variable, the internal default value will be used, rather than the one from the system */etc/esoreflex.rc* file. Putting this another way, no merging is performed between custom configuration files and the system wide one.
Note: using a custom esoreflex.rc together with a system wide installation is meant for power users only. Furthermore, users should consider that system wide upgrades of Reflex might break the custom configuration defined in esoreflex.rc

8.3 Creating a configuration file

The configuration file for *esoreflex* is just a text file that can be created or modified with any text editor. For convenience however, there is a command line option to the *esoreflex* launch command called *-create-config* that can be used to generate a customised configuration file filled with all valid variables. This can then be further modified as needed. The option takes one parameter, either TRUE, or the name of the file to which the configuration will be saved. Existing files will be backed up to a file with the '.bak' extension, or '.bakN' where N is an integer if the file name is already taken. For example, the following command will create a new configuration file in *~/.esoreflex/esoreflex.rc*:

```
  esoreflex -create-config TRUE
```

Alternatively, to create a file named myconfig.rc in the local directory, you can run the following command:

```
 esoreflex -create-config myconfig.rc
```

8.4 Configuration file format

The configuration files are plain text files that contain “name=value” pairs on separate lines. There should be no spaces between the name of the variable and the equal sign '='. Any whitespace in the value is used as is. Thus, any spaces immediately after the equal sign may have semantic meaning and should be avoided if not necessary. If a variable name is given more than once in the same file the value from the last declaration is used and all previous declarations are ignored. Comments start with the pound sign '#' character. All text starting from this character (including the '#') is ignored up to the end of the line.
To help make configuration files compatible across versions of *esoreflex*, the special *${esoreflex_base}* macro can be used in the value part of a variable declaration. This allows you to avoid hardcoding the base installation path of the *esoreflex* package, which can change from version to version or by installation method. The following shows an example usage for encoding the *python* path:

```
 esoreflex.python-path=${esoreflex_base}/esoreflex/python
```

The tilde character '~' can also be used as usual in any variable expecting a path. This will be expanded to the user's home directory.
All variables that can be used in a *esoreflex* configuration file are listed and described in the following sub-sections. None of the variables described below have to be present in the configuration file. For any variable that is not present, an internal default value will be used. However, it is possible that on certain platforms the default may not be appropriate and lead to errors. You can use the *-create-config* option to make sure you start from a working version of the configuration file, with all variables properly set.

8.4.1 esoreflex.inherit-environment

This controls if the user's environment variables should be used by the *esoreflex* command as is, i.e. inherited or not. The value must be either *TRUE* or *FALSE*. If set to *TRUE* then no changes are made to the user's environment variables when invoking the *esoreflex* binary and all of them will be visible to the binary. In addition, any environment variables configured by the user will take precedence over values set by the *esoreflex* launch command as derived from the configuration file. In contrast, for a value of *FALSE*, every environment variable will be unset and only a select number will be configured when invoking *esoreflex*. A listing of the environment variables used by *esoreflex* can be seen by adding the *-debug* option to the command line call.

8.4.2 esoreflex.java-command

This must contain the Java binary or command that must be used to start a Java process. In the simplest case, it can just contain the string *java*. Alternatively this can be a full path to a specific binary to use. Extra command line options can also be given if necessary. For example:

```
  esoreflex.java-command=/usr/bin/java -Xdiag
```

8.4.3 esoreflex.workflow-path

This must be a colon separated list of paths in which to search for workflow XML or KAR files. The tilde character can be used in the paths, which will be expanded to the user's home directory. As an example:

```
esoreflex.workflow-path=~/KeplerData/workflows/MyWorkflows
```

8.4.4 esoreflex.esorex-command

This variable must contain the required command to invoke the *esorex* binary. By default it is set to the string *esorex*. This variable can contain the full path to the binary and also include additional command line options. An example where this can be handy is to execute *esorex* with changed priorities (using *nice*) or to pin it to particular processors as shown below:

```
esoreflex.esorex-command=likwid-pin -c N:0-3 esorex
```

8.4.5 esoreflex.esorex-config

This must be a path to the configuration file to use by the *esorex* program. With a normal system installation this will point to a custom configuration file prepared by *esoreflex* itself. To use a different file change or set the path where the file is located. For example:

```
esoreflex.esorex-config=~/.esorex/esorex.rc
```

8.4.6 esoreflex.esorex-recipe-config

Similarly to the *esoreflex.esorex-config* variable, this must point to the location of the default recipe configuration file to use when invoking *esorex*. Normally this should point to an empty dummy file, but can be adjusted in special cases. For example:

```
esoreflex.esorex-recipe-config=~/.esorex/myrecipe.rc
```

8.4.7 esoreflex.python-command

Similar to the configuration of the *esorex* command, this variable will configure the command that must be invoked to start *python*. By default this is simply set to the string *python*. You can use the full path to the binary and include extra command line options as usual. For example:

```
esoreflex.python-command=/usr/bin/python27
```

8.4.8 esoreflex.python-path

This must contain a colon separated list of additional paths to search for *python* modules. If the *esoreflex.inherit-environment* variable is *TRUE* then the contents of *esoreflex.python-path* will be appended to the *PYTHONPATH* user's environment variable if it exists, otherwise *PYTHONPATH* will be set to the contents of *esoreflex.python-path* as is. In system installations of *esoreflex*, its configuration files will usually set the *esoreflex.python-path* variable to contain search paths to internal *python* modules. You will have to be careful not to remove these, to avoid breaking *esoreflex*. For example:

```
esoreflex.python-path=${esoreflex_base}/esoreflex/python:~/mymodules
```

8.4.9 esoreflex.path

This must be a colon separated list of additional binary search paths that will be added to the *PATH* environment variable. This is particularly useful if you want to be able to invoke a privately installed program or script from a system installation of *esoreflex*. If the *esoreflex.inherit-environment* variable is *TRUE* then the contents of *esoreflex.path* is appended to the user's *PATH* environment variable. Thus the user's binaries will take precedence. Otherwise for *esoreflex.inherit-environment=FALSE* the contents of *esoreflex.path* is prepended to the system PATH variable as returned by the *getconf* command. In which case the binaries found under the paths in *esoreflex.path* will take precedence over the user's ones.

8.4.10 esoreflex.library-path

This must be a colon separated list of additional search paths for shared libraries. When *esoreflex.inherit-environment* is set to *TRUE* then the contents of *esoreflex.library-path* will be appended to the *LD_LIBRARY_PATH* environment variable (*DYLD_LIBRARY_PATH* for BSD derived platforms, such as Apple OS X). This means that the user's shared libraries will take precedence if the *(DY)LD_LIBRARY_PATH* was configured by the user. Otherwise when *esoreflex.inherit-environment=FALSE* the *(DY)LD_LIBRARY_PATH* environment variable is set to the contents of *esoreflex.library-path* as is, completely ignoring any value set in the user's environment. For system installations of *esoreflex*, the configuration files may have this variable set to point to internal shared libraries for certain platforms. Care will have to be taken not to remove these paths so as to avoid breaking *esoreflex*.

8.5 Environment variables

The behaviour of the *esoreflex* launch command can be modified with the following environment variable.

8.5.1 ESOREFLEX_CLEAN_ENVIRONMENT

If this variable is set, it's value must be either *TRUE* or *FALSE*. If set to *TRUE* then all environment variables will be unset before invoking the *esoreflex* binary. Only certain environment variables are set within the clean environment setting before the invocation. This environment variable will also override any setting of *esoreflex.inherit-environment* provided in a configuration file e.g. the system wide *esoreflex.rc*.
The following command can be used to see exactly which environment variables are used and configured by the launch command:

```
esoreflex -debug
```

The variables will be listed in the "Environment used" section.

8.6 Custom esoreflex.rc file use cases

Here we present some examples to show how to setup a configuration file for *esoreflex* to deal with certain common use cases. These will all assume the *bash* or similar shell is used. You will have to adjust the examples for other shell types appropriately.

8.7 Using a private installation of esorex

In certain cases you may want to use a private installation of *esorex*, for debugging for example. If using the *esoreflex* tarball package rather than a system installation, the user's environment variables will be inherited. In this case all you need to do is make sure that the custom *esorex* binary is found in the *PATH* environment variable. For example, if you have a custom *esorex* binary installed under *~/myesorex/bin/esorex*, you should preprend the path to the *PATH* variable as indicated below:

```
export PATH="~/myesorex/bin:$PATH"
```

To have this setting persist across login sessions you can add this to your *~/.profile* or *~/.bashrc* file as appropriate. You may also need to similarly deal with the *LD_LIBRARY_PATH* (*DYLD_LIBRARY_PATH* on BSD derived platforms like Apple OS X) if you have shared libraries in non-standard locations.
If a system installation of *esoreflex* was performed then the user's environment variables will not be inherited by default. Thus, the above export method will not work. Instead, a new custom configuration file should be created for *esoreflex*. Here we show how to create a configuration file that will apply to all your login sessions (not to all users). The first thing to do is create a working template configuration with the following command:

```
esoreflex -create-config TRUE
```

This will create a new file in *~/.esoreflex/esoreflex.rc*. Any existing configuration will be backed up to a file with a *.bak* or *.bakN* (where *N* is an integer) extension. You will have to edit this new file in a text editor and change the variable *esoreflex.esorex-command* to the following value:

```
esoreflex.esorex-command=~/myesorex/bin/esorex
```

If shared libraries have been installed in a non-standard location, for example the CPL libraries went into *~/mycpl/lib*, you will also have to let *esoreflex* know about these also. Modify the *esoreflex.library-path* variable by appending the path *~/mycpl/lib* to it:

```
  esoreflex.library-path=~/mycpl/lib
```

If there was already a value set for *esoreflex.library-path* then you likely want to keep that and simply append the new path to the end, as indicated in the following example:

```
  esoreflex.library-path=/opt/local/lib:~/mycpl/lib
```

Once done modifying the configuration, the next invocation of *esoreflex* should now use your custom *esorex* binary. If any sessions of *esoreflex* were already started before creating the custom configuration, you will have to close them and restart *esoreflex*.
As an alternative to creating a configuration file that will be used for all your *esoreflex* sessions, you may want to create a custom configuration file to use just for a particular *esoreflex* invocation. In this case you should run the following command to create a configuration file template in your current working directory:

```
 esoreflex -create-config mycfg.rc
```

You can of course use a different path or even a full path for the file, rather than *mycfg.rc*. The *esoreflex.esorex-command* and *esoreflex.library-path* variables must be modified in the *mycfg.rc* file as indicated before. But you will now have to remember to add the *-config* option to the *esoreflex* launch command whenever you want to use this configuration file. For example:

```
  esoreflex -config mycfg.rc
```

8.8 Using a private installation of python

When you want to use a private or different installation of python, rather than the system one, you can do so in a similar manner as shown previously for *esorex*. If using the *esoreflex* tarball directly, the *python* that is used by *esoreflex* is configured in the user's environment variables. Specifically this is set in the variables *PATH* and *LD_LIBRARY_PATH* (*DYLD_LIBRARY_PATH* on BSD derived platforms like Apple OS X). Details are already shown in the *esorex* section above and not repeated here. The following will instead show how to modify the python that will be used by a system installation of *esoreflex*, which is the more common end user case.
Let us assume your custom *python* installation resides in *~/mypython/bin* and you have some python modules in *~/mypython/modules* that you would like to use. The first step requires creating a custom configuration file template for *esoreflex*. This can be done with the following command:

```
 esoreflex -create-config TRUE
```

A new file will be created in *~/.esoreflex/esoreflex.rc*. Any previous copy will be backed up to a file with a *.bak* or *.bakN* (where *N* is an integer) extension. The *~/.esoreflex/esoreflex.rc* file must be edited in a text editor and the following variables set:

```
 esoreflex.python-command=~/mypython/bin/python
 esoreflex.python-path=${esoreflex_base}/esoreflex/python:~/mypython/modules
```

Take note that any value that was already set in the *esoreflex.python-path* variable should be kept, unless you wanted to replace the internal esoreflex python modules and you know what you are doing. You should just append your own paths to the *esoreflex.python-path* varaible with a colon separating all paths.

One the modifications are complete you should stop any *esoreflex* sessions that are already running and restart *esoreflex*. All future invocations of *esoreflex* should now use the custom python installation with your own modules.
If you wanted to only use the custom python only for certain invocations of esoreflex, you will want to create a custom configuration file other than in *~/.esoreflex/esoreflex.rc*. For example, *mycfg.rc* in the current working directory as follows:

```
esoreflex -create-config mycfg.rc
```

This file should be modified as indicated before and *esoreflex* must be invoked with the *-config* option as follows:

```
esoreflex -config mycfg.rc
```

You can change the name of *mycfg.rc* to any value you want, so long as it is not *~/.esoreflex/esoreflex.rc*, which would be automatically read by every *esoreflex* invocation. You can also use full paths to the configuration files for the *-config* and *-create-config* options.

8.9 Using the user's environment

When dealing with a system installation of *esoreflex*, you may want to customise various aspects of how *esorex* or *python* is executed using your user environment variables that you have set up. By default a system installation of *esoreflex* will not inherit the user's environment variables i.e. they will all be unset, except for a select few, such as the *PATH*, *HOME*, *LANG*, *LOGNAME*, *HOSTNAME* and *DISPLAY* (Note: this list may not be exhaustive. Use *esoreflex -debug* to see an exact listing). To force *esoreflex* you use the environment as has been set by you, you will need to create a modified configuration file.
Start by running the following command to create a new configuration template in *~/.esoreflex/esoreflex.rc*:

```
esoreflex -create-config TRUE
```

Any existing *~/.esoreflex/esoreflex.rc* file will be backed up to *~/.esoreflex/esoreflex.rc.bak* with a possible integer suffix.
You will then want to edit the new file in a text editor. Change the *esoreflex.inherit-environment* variable and set it to TRUE, as shown below:

```
esoreflex.inherit-environment=TRUE
```

When done you should stop all existing *esoreflex* sessions and restart them. All environment variables that you normally see in your terminal session should now also be used by future invocations of *esoreflex*. To confirm this you can start *esoreflex* with the following command and inspect the “Environment used” section:

```
esoreflex -debug
```

8.10 Customising the esorex command used by esoreflex

To customise the *esorex* command that will be used by *esoreflex*, you will need to create a new configuration file. As indicated in previous sections, the template file should be created with the following command:

```
esoreflex -create-config TRUE
```

This will create a new file called *~/.esoreflex/esoreflex.rc* and any existing file in that location will be backed up to *~/.esoreflex/esoreflex.rc.bak* with a possible integer suffix. The *~/.esoreflex/esoreflex.rc* file will be used for all your invocations of *esoreflex*, unless using the *-config* option. It is possible to create a configuration file that will only be used for certain invocations of *esoreflex*. For example, if you want a custom configuration file called *~/my_esoreflex_cfg.rc* in your home directory, you should run the following command:

```
  esoreflex -create-config ~/my_esoreflex_cfg.rc
```

In either case, the new file must be modified in a text editor. Assume you want to change the scheduling priority of *esorex* with the *nice* command to 10, you need to set the *esoreflex.esorex-command* variable as follows:

```
 esoreflex.esorex-command=nice -n 10 esorex
```

You can change the command to whatever you like. As another example, to use the *likwid* tool to force *esorex* to use only certain processors, you may want to set the *esoreflex.esorex-command* variable as follows:

```
 esoreflex.esorex-command=likwid-pin -c N:0-3 esorex
```

You may also need to update the *esoreflex.path* variable in the new configuration file, to indicate the location of the *likwid-pin* binary as follows:

```
 esoreflex.path=/usr/local/bin
```

Adjust the path appropriately for your platform.
Once all the modifications are complete, you should stop any running esoreflex sessions and restart them. In the case that the configuration file was saved to *~/my_esoreflex_cfg.rc*, you should start esoreflex as follows:

```
  esoreflex -config ~/my_esoreflex_cfg.rc
```

9. TROUBLESHOOTING

9.1 Debug mode

By default, Reflex displays some information on the console: it is possible to change the level of logging for debugging purposes by editing the file `kepler-2.4/resources/log4j.properties` in the Reflex distribution.
You can increase the logging level by adding new lines at the end of the file (the default logging level is WARN, defined at the beginning of the file).

Log4j, the logging library used by Reflex, defines the following logging levels, starting from the most verbose: `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR`, `FATAL`.

If you want to put all Reflex actors in debug mode change the first line as follows:
```
log4j.logger.org.eso=DEBUG, CONSOLE
```

If you want, for example, to raise the logging level of the RecipeExecuter to `DEBUG`, then you have to add the following line:
```
log4j.logger.org.eso.RecipeExecuter=DEBUG, CONSOLE
```

Note that the most restrictive condition is applied.

9.1.1 Logging to file

By default Reflex logs only to the console, so that if you want the logs to be saved also to a file, then you have to modify the first line of the configuration file (see 9.1) from
```
log4j.rootLogger=WARN, CONSOLE
```
to
```
log4j.rootLogger=WARN, CONSOLE, R
```
The log filename is defined in the section `LOGGING TO FILE` of the same configuration file.

9.2 Reflex hangs

In the past some users have experienced a Reflex hang when they were browsing the ESO workflows: the origin of this issue is unclear, but it can be solved by increasing the memory assigned to Reflex.
To do so, open the menu entry “Tools->JVM Memory Settings” and increase the value of Max Memory.
The installation script provides a simple tool (`esoreflex_set_memory`) to change it from the command line.

9.3 SQL error messages

If you run into error messages about a missing table or column, it’s very likely that you are using a bookkeeping directory created by an old version of Reflex. The solution is to remove it (or move it, if you still need to run said old version): the default location is `~/reflex_data/reflex_book_keeping`.

9.4 Strange behavior of actor String Costant

The actor *String Constant* comes from Kepler, and it has a peculiar behavior when it’s value is set to a parameter (i.e. something starting with a $ sign): if you close the window using the *Commit* button, then the value is stored as input. If, on the other hand, you press *Enter*, then Kepler tries to resolve the value of the parameter and replaces it in the actor. The effect is that, if afterwards you change the value of the parameter, the actor will not pick it up.

APPENDIX A: SIMPLIFIED SOP FORMAT

Some actors (i.e. the SopCreator) use a simplified version of the SoP format in order to ease interaction with the user and with custom scripts. These formats are described here.

- SoP: RECIPE1:PAR1=VAL1,RECIPE2:PAR2=VAL2…
    - Example: uves_cal_mdark:process_chip=both,uves_cal_mflat:backsub.mmethod=median

APPENDIX B: example.py

```
#!/usr/bin/env python

# import reflex module
import reflex

import sys

if __name__ == '__main__':

    # create an option parser
    parser = reflex.ReflexIOParser()

    # define inputs (Note: you must define at least long option)
    parser.add_input("-i", "--input1")
    parser.add_input("-j", "--input2")

    # define outputs (Note: you must define at least long option)
    parser.add_output("-o", "--output1")
    parser.add_output("-p", "--output2")

    # get inputs from the command line
    inputs = parser.get_inputs()
    # get output variables
    outputs = parser.get_outputs()

    # read inputs and assign outputs
    if hasattr(inputs, "input1"):
        outputs.output1 = inputs.input1
    else:
        outputs.output1 = 'test1'

    if hasattr(inputs, "input2"):
        outputs.output2 = inputs.input2
    else:
        outputs.output2 = 'test2'

    # print outputs
    parser.write_outputs()

    sys.exit()
```

APPENDIX C: SOFTWARE REQUIREMENTS

Reflex requires java 8, update 121 is recommended: in principle every recent Linux distribution should be compatible, but this can not be guaranteed.
Workflows may have specific requirements (e.g. python, astropy, etc.): please refer to the pipeline manual or to the workflow tutorial for more information.