A&A 559, A96 (2013)
DOI: 10.1051/0004-6361/201322494
© ESO 2013

Astronomy & Astrophysics

# Automated data reduction workflows for astronomy

## The ESO Reflex environment

W. Freudling, M. Romaniello, D. M. Bramich, P. Ballester, V. Forchi, C. E. García-Dabló, S. Moehler, and M. J. Neeser

European Southern Observatory, Karl-Schwarzschild-Str. 2, 85748 Garching, Germany
e-mail: wfreudli@eso.org

Received 16 August 2013 / Accepted 14 October 2013

**ABSTRACT**

*Context.* Data from complex modern astronomical instruments often consist of a large number of different science and calibration files, and their reduction requires a variety of software tools. The execution chain of the tools represents a complex workflow that needs to be tuned and supervised, often by individual researchers that are not necessarily experts for any specific instrument.
*Aims.* The efficiency of data reduction can be improved by using automatic workflows to organise data and execute a sequence of data reduction steps. To realize such efficiency gains, we designed a system that allows intuitive representation, execution and modification of the data reduction workflow, and has facilities for inspection and interaction with the data.
*Methods.* The European Southern Observatory (ESO) has developed Reflex, an environment to automate data reduction workflows. Reflex is implemented as a package of customized components for the Kepler workflow engine. Kepler provides the graphical user interface to create an executable flowchart-like representation of the data reduction process. Key features of Reflex are a rule-based data organiser, infrastructure to re-use results, thorough book-keeping, data progeny tracking, interactive user interfaces, and a novel concept to exploit information created during data organisation for the workflow execution.
*Results.* Automated workflows can greatly increase the efficiency of astronomical data reduction. In Reflex, workflows can be run non-interactively as a first step. Subsequent optimization can then be carried out while transparently re-using all unchanged intermediate products. We found that such workflows enable the reduction of complex data by non-expert users and minimizes mistakes due to book-keeping errors.
*Conclusions.* Reflex includes novel concepts to increase the efficiency of astronomical data processing. While Reflex is a specific implementation of astronomical scientific workflows within the Kepler workflow engine, the overall design choices and methods can also be applied to other environments for running automated science workflows.

**Key words.** methods: data analysis – astronomical databases: miscellaneous – techniques: miscellaneous – virtual observatory tools

## 1. Introduction

Astronomical observations produce data streams that record the signal of targets and carry associated metadata that include observational parameters and a host of associated information. Apart from the intended signal, such raw data include signatures of the atmosphere and the instrument, as well as noise from various sources. Before any scientific analysis of the data, a process called "data reduction" is used to remove the instrumental signature and contaminant sources and, for ground based observations, remove atmospheric effects. Only then, can the signal of the target source be extracted. In general, data reduction also includes a noise model and error propagation calculations to estimate uncertainties in the extracted signal.

In recent years, astronomical data reduction and analysis has become increasingly complex. The data from modern instruments can now comprise dozens of different data types that include both science and calibration data. For example, the reduction of data from ESO's X-shooter instrument uses almost 100 different data types for its three simultaneously working arms. The reduction of such data in general includes a large number of complex high-level algorithms and methods. The data types and methods are interdependent in a complex web of relations. It is therefore increasingly difficult for an individual researcher to understand, execute and optimize a data reduction cascade for complex instruments. This situation has led to the appearance of specialized, highly integrated data reduction pipelines that are written by specialists and can reduce data without supervision (e.g. Biretta et al. 1994; Jung et al. 2004; Tucker et al. 2006; Schmithuesen et al. 2007; Cavanagh et al. 2008; Tsapras et al. 2009). For efficient large scale data reduction, such pipelines often run in custom-made environments. For example, ESO employs a system for quality control that automatically associates calibration data and processes them as soon as they arrive from the telescopes. The results are then stored in its data archive. Other examples of such event driven data reduction environments are NOAO's High-Performance Pipeline (Scott et al. 2007), STScI's OPUS system (Rose et al. 1995), and the Astro-WISE pipeline (McFarland et al. 2013).

Automatic pipelines work best for data from long-term projects that use stable instrumentation, aim for a well-defined set of similar targets observed at similar signal-to-noise ratio, and in situations where the impact of ambient conditions is relatively small and highly predictable. These conditions are often met, for example, in space-based telescopes. However, the situation is often different for the reduction of data from ground-based observatories. The reasons for this include the complexity of general purpose instruments that are now routinely employed, the rapid upgrade pace necessary to exploit advances in technology and science goals, and the variety of effects imposed by varying atmospheric conditions. In many cases, supervision and interaction with the data reduction process is, therefore, still essential to obtain sufficiently high quality results even from fairly routine observations.

The general concept of astronomical data reduction that does not employ a fully integrated pipeline has not substantially changed in the past decades. Researchers organise their data, and use a mixture of general purpose and highly specialized tools, inspecting the results of each step. Such tools are available in environments such as MIDAS (Banse et al. 1983), IRAF (Tody 1993) and IDL[1], or as stand-alone programs. What has changed is the number, complexity and interdependence of steps needed to accomplish the data reduction. In this situation, the efficiency of the data reduction process can be vastly improved by automating the previously manual workflow of organising data, running individual steps, and transferring results to subsequent steps, while still using the same routines to carry out individual reduction steps.

The most commonly used approach to automate a data reduction workflow by individual researchers is to employ a scripting language such as Python (e.g. Nastasi et al. 2013). This approach works well with a relatively small number of reduction steps, and in situations where the data organisation and book-keeping are fairly simple. In more complex situations, such scripts are themselves complex programs that cannot easily be modified.

In this paper, we describe the usage of a general workflow engine to automate the data reduction workflow for astronomical observations. While this approach is relatively new for the field of astronomy (e.g. Ballester et al. 2011; Schaaff et al. 2012), it has been widely used in other fields of science including biology and chemistry (Abhishek & Sekharb 2007), meteorology (Barseghian et al. 2009) and economics (Ludäscher et al. 2005). For that reason, we discuss in detail the methods and functionalities that are necessary to use such a system for astronomical data reduction, and present ESO's new "Recipe flexible execution workbench" (Reflex) environment as a specific implementation of such a system.

The structure of the paper is as follows. In Sect. 2, we describe the main principles and architecture of our design independent of a particular implementation. In Sect. 3, we discuss how these principles can be implemented in a specific workflow application, using our Reflex implementation as an example. Finally, in Sect. 4 we conclude with a discussion of the impact of performing data reduction in this way.

## 2. Architecture of astronomy data reduction workflows

### 2.1. Data organisation

Astronomical data consist of collections of files that include both the recorded signal from extraterrestrial sources, and metadata such as instrumental, ambient, and atmospheric data. Such a collection of files is the raw output from one or several observing runs, and consists of "science files" that contain the primary science observations to be analyzed.

In addition, it might include files that are not directly related to the current observations, such as calibration files that are routinely collected for a given instrument. Hereafter, we refer to the input files for the data processing as "raw files", as opposed to files that are created during the data processing and that we will refer to as "products". We use the term "calibration file" for any raw file that is not a science file.

In order to discuss data reduction in general terms, we introduce the following terminology. The goal of data reduction is to process sets of files, which we refer to as the *target*s of a data reduction workflow. The result of this processing is to create a *target product*. In most cases, the *targets* of a data reduction workflow will be the science files, and the *target product* is then the science data product to be used for scientific analysis. The *target* files can be naturally grouped into sets that are reduced together. Such a group of *target* files, together with other files needed to process them, is referred to as a *data set*. A *data set* is "complete" when it contains all necessary files to reduce the *target*s, and "incomplete" if some of those files are missing.

"Data organisation" is the process of selecting *data set*s from a larger collection of files, and recording information on the type of files and the reasons for selecting them. This larger collection of files might be the result of a pre-selection process that assures that low quality or defective data are not considered at this stage. Organising data is a complex and time-intensive procedure that is typically among the first tasks of a data reduction workflow (e.g. Scodeggio et al. 2005). Hereafter, we will refer to the whole data reduction workflow including data organisation simply as a "workflow", whereas we will use the term "data processing workflow" for the processing of data that follows the data organisation.

The first step in data organisation is to classify files, i.e. to determine the data content of each file from its metadata. The goal of classification is to assign a *category* to each file. An example of such a *category* is "flatfield for filter I". The next step is to identify the *target*s, and group them into *data set*s that are incomplete at this stage. Subsequently, calibrations are added to the *data set*s. Calibration files for each *data set* are selected by analysing the metadata of the *target*s and that of other available files that potentially qualify for inclusion in a *data set*.

This cascade of selection criteria naturally maps into a data graph as illustrated in Fig. 1. The links between elements of the graph show the flow of metadata that originates from the raw files. The graph is directed, i.e. links between elements have a direction to distinguish between incoming and outgoing information. The nodes of the graph define necessary procedural steps in the assembly of a *data set*, and we refer to them as *action*s. The targets of the workflow connect directly to the root node (action 1) that is therefore called the *target action*.

Each *action* has several incoming files connected to it. Some of those are used to define selection properties of other input files to that *action*. For example, an *action* might specify to select flat field images that use the same filter as the science image. We use the notation that the files that are used to define properties of other files, in our example the science files, are the *trigger* for that *action*, and their links are shown as solid lines in Fig. 1. The *trigger* of the *target action* are the *target*s of the workflow.

All *action*s other than the *target action* have one or several outgoing links that connect them to subsequent *action*s. These outgoing links pass on metadata that are extracted from the input files to the next *actor*. They are therefore called *product*s of an *action*. These *products* do not necessarily correspond to actual physical products produced during data reduction, and the actual physical products created during data reduction do not necessarily appear in the data organisation graph. Instead, the products in the data organisation graph are used as a logical scheme to define the selection of data. For example, for the purpose of data organisation, it is not necessary to define a *target product*, even when the data processing workflow creates one. This is because

[1] IDL is a trademark of Research Systems Inc., registered in the United States.

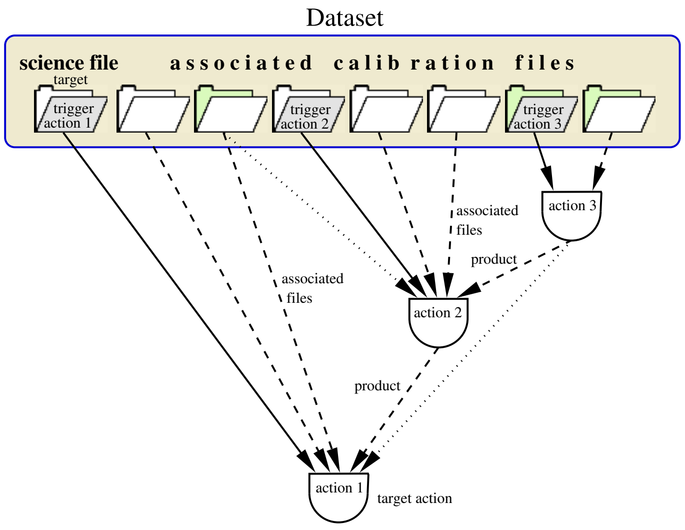


**Fig. 1.** Example of a simple *data set* and its organisation. The *data set* contains all files necessary to produce the science data product of the workflow. This includes the science and associated calibration files. These files are organised using a set of *action*s that are shown as shield-shaped symbols. The target files are directly connected to the target *action* that is the root of the graph. Files that are connected to an *action* with a solid line are the *trigger* for that action. Properties of the *triggers* are used to select associated files for an *action*. The associated files are connected to an *action* with dashed or dotted lines. To highlight files that are connected to more than one *action*, a dashed line is used for one of these connections, and a dotted line for the other one. The *purpose* of a file is the connection between the file and the *target action*. Symbols with tinted background indicate files that have multiple *purpose*s, i.e. there are multiple paths from the file to the *target action*.

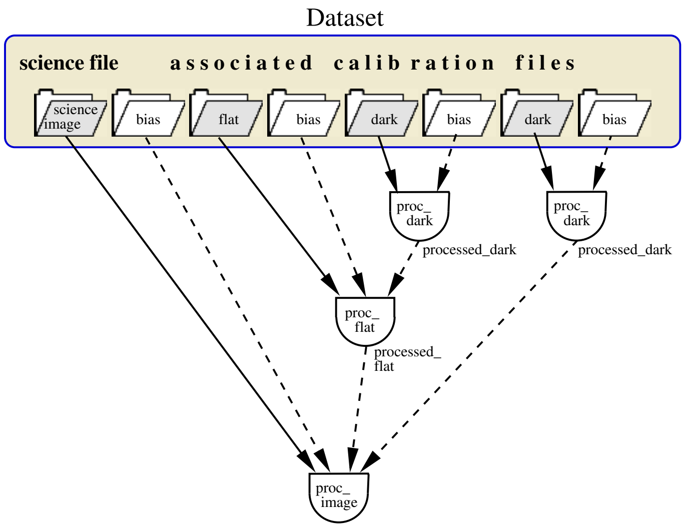


**Fig. 2.** Data graph for a data set to process images as described in the text. The symbols used are the same as in Fig. 1. In the case shown here, each file has a unique *purpose*, and therefore no dotted lines are used. The *action* "proc_dark" is used to select different darks for the flat frame and the science image. Therefore, it appears twice in the graph.

the nature and properties of the target product have no impact on the data selection.

Each raw file is the origin of at least one path along the direction of the links that lead to the *target action*. This reflects the fact that *data set*s only include raw files that are needed to process the targets of the workflow. A path runs either directly from the files to the target action, or passes through other actions on its way. We refer to such a path as one of the *purpose*s of a file. The *purpose*s of a file are important information for the data processing (see Sect. 2.3).

In Fig. 2, we show the data graph for a specific example with the same symbols used in Fig. 1. The simple example is an image that needs a bias frame, a flatfield and a dark frame for its processing. The flatfield needs to be taken with the same optical filter as the science frame, whereas the dark frame needs to be taken with the same exposure time as the science frame. Therefore, flatfields and dark frames with these properties must be identified among available files, and one of each must be selected according to criteria such as the closeness of the time of observation to that of the science frame. After this step, more calibration files need to be added to the *data set* that are used to reduce the calibration files. The selection criteria for those files depend on properties of calibration files instead of the *target*s of the workflow. In the current example, the flatfield itself needs a dark frame for its processing, and this dark frame needs to match the exposure time of the flatfield, not that of the science frame. The science frame, flatfield frame and dark frame in turn might all require their own bias frame for reduction. The *actions* in this case are given specific labels, namely "proc_dark", "proc_flat" and "proc_image". Note that the action "proc_dark" is shown twice, reflecting the fact that it is used twice, once to select darks for the flatfield, and a second time to select darks for the image.

We note that the topology of the graph might differ between *data set*s even for the same kind of data. For example, in one *data set* the input dark frames for the science and flat frames might be identical, in another one they might differ. The task of data organisation is to create such a graph for each individual data set.

### 2.2. Design of data organisation and data processing systems

The data organisation discussed in Sect. 2.1 and the data processing that follows the data organisation both describe relations among different *categories* of data. These relations are interdependent, in the sense that a change in the selection of data might require some change in the data processing, and vice versa. The question therefore arises as to whether the best architecture is to derive these relations from a common source, or whether the information recorded in these relations is sufficiently different to warrant their independent implementation.

In general, a specific selection of data does not uniquely specify the data processing sequence. Very different data processing workflows can be constructed to use a given selection of data. Only the most basic data processing follows the data organisation process one-to-one, but this case is rarely used in practice. The data processing part of the workflows are, in general, more complex than the data organisation, and are also more frequently subject to change and optimization during the data reduction process. The *purpose* of a file records an aspect of the selection criteria used to include this file in a *data set*. It is up to the workflow design to decide how this information is used.

For example, a *category* of files (such as a flatfield) might be selected to match the date of the science frames and is, therefore, assigned a corresponding *purpose*. This does not necessarily mean that these flatfields are exclusively used to flatfield the science frames, but the data processing workflow might also use them to flatfield standard star flux calibration data. For spectroscopy, it is not always clear whether the best flatfields for the flux calibrator are those that are taken close in time to the target spectrum or those taken close in time to the flux calibrator. This decision depends on a complex set of circumstances. A workflow might include conditional and/or interactive parts to help the user make that decision.

Another difference between data organisation and data processing is that, while some steps in the data processing are closely related to a specific selection of data, others are completely independent of it. For example, a step that only modifies intermediate products has no impact on the data selection or organisation. Steps that make small adjustments to intermediate products are often added or removed during data reduction. Any system that mixes the data selection and data processing workflows is then, necessarily, much more complex than either of the two components individually. One design goal for a workflow system is to make modification of the data reduction as simple as possible. This is helped by clearly separating the data organisation from the data reduction steps.

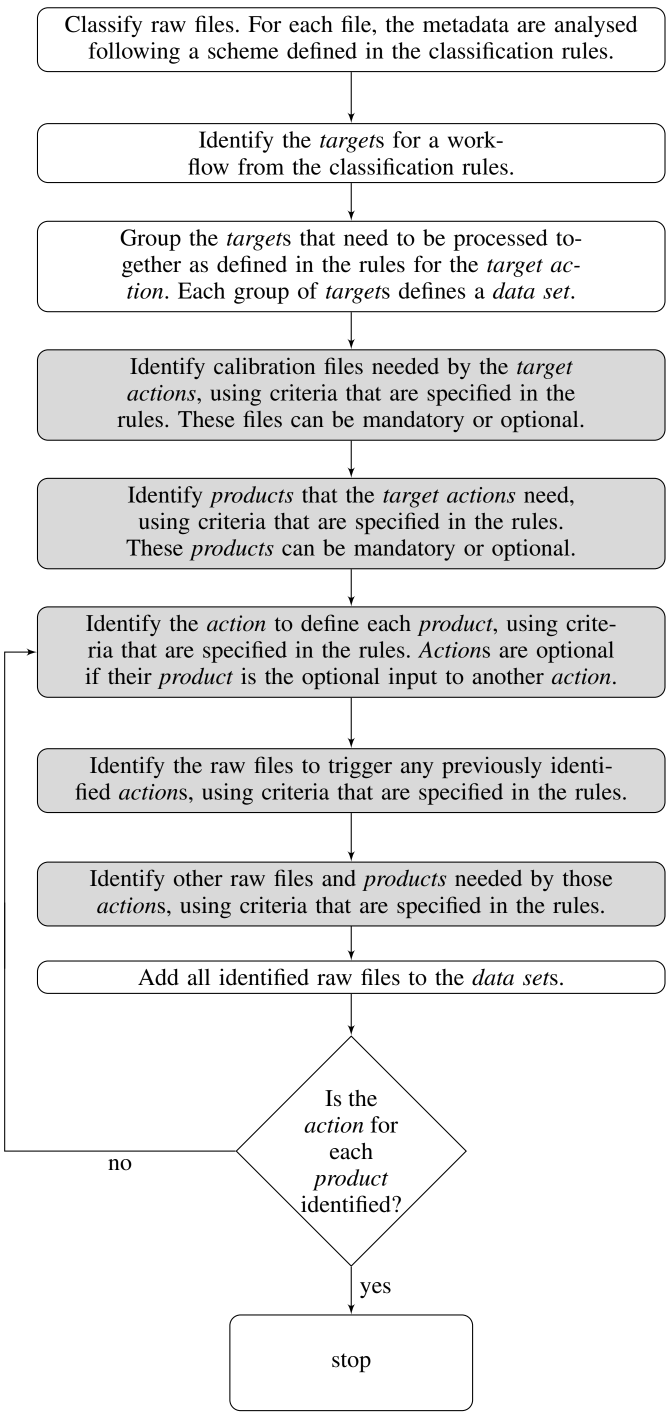
We therefore advocate a design that not only separates the implementation of the two steps, but also uses a different methodology to define the two tasks. Each of them should be geared towards the specific needs of each step. The data organisation is usually closely related to the instrument properties, the observing strategy and the calibration plan. The strategy for data organisation therefore rarely changes after the observations have taken place. Interactivity in that part will create overheads that do not outweigh the expected benefits. On the other hand, the data processing is, in general, highly interactive and experimental, and the final strategy is rarely known at the time of observation. The best values for data reduction parameters and even the chosen strategy might depend on the properties of individual *data set*s.

An efficient way to implement a data organisation is, therefore, a rule-based system that can accommodate complex, instrument-specific rules and can be run to organise either locally stored data or data extracted from an archive repository using pre-defined rules. The syntax of the rules must be able to describe the method of creating data graphs such as the ones discussed in Sect. 2.1. Such data organisation is particularly efficient if it is carried out by raw-data archives that have any potentially useful calibration file available for retrieval. For example, ESO offers an archive service "calselector"[2] that selects, organises and provides access to data in a manner similar to the one described above.

In contrast, data processing after the data organisation benefits from interactive, graphical and dynamic elements. An efficient way to provide this is to use a workflow application that allows the implementation of workflows that can be easily modified for experimentation and optimization during data processing. It is important that these interactive features can be turned off once a workflow has been tuned and optimized, in order to allow time-intensive processing to be carried out in a batch mode.

### 2.3. Functionalities of a rule-based data organiser

A software program that uses rules to organise data as advocated above can produce the data graph discussed in Sect. 2.1 by a set of steps shown in the flow chart of Fig. 3. The output of the data organisation is a list of *data set*s. A *data set* is marked as "complete" if there are files that satisfy the criteria used in steps shown in shaded boxes in Fig. 3. It is marked as "incomplete" if any one



Classify raw files. For each file, the metadata are analysed following a scheme defined in the classification rules.
Identify the *target*s for a workflow from the classification rules.
Group the *target*s that need to be processed together as defined in the rules for the *target action*. Each group of *target*s defines a *data set*.
Identify calibration files needed by the *target actions*, using criteria that are specified in the rules. These files can be mandatory or optional.
Identify *products* that the *target actions* need, using criteria that are specified in the rules. These *products* can be mandatory or optional.
Identify the *action* to define each *product*, using criteria that are specified in the rules. *Action*s are optional if their *product* is the optional input to another *action*.
Identify the raw files to trigger any previously identified *action*s, using criteria that are specified in the rules.
Identify other raw files and *products* needed by those *action*s, using criteria that are specified in the rules.
Add all identified raw files to the *data set*s.
Is the *action* for each *product* identified?
no
yes
stop

**Fig. 3.** High-level flow chart of a data organiser. If any step in a shaded box fails for any given *data set*, then this *data set* is marked as "incomplete".

of those criteria are not satisfied by any existing file. Each file in the output *data sets* is described by the file name, the *category* of the file as defined in the rules, and the *purpose* of the file. The *purpose* of the file is recorded as the concatenation of the names of the *action*s that link the file to the *target action*. In the example discussed in Sect. 2.1, the flatfield is selected based on properties (in this case the filter) of the science frame. The selection rules are defined in the *action* called "proc_science".Then the bias frame is selected for this flatfield based on properties of the flatfield (e.g. observing date or read-out mode). This selection is defined in the action "proc_flat". The *purpose* of this bias frame,

[2] http://www.eso.org/sci/archive/calselectorInfo.html

as well as the flatfield, is then "proc_science:proc_flat", while the bias frame that matches the properties of the science frame, as well as the science frames themselves, have the *purpose* "proc_science". The other biases in this example have the purpose "proc_science:proc_flat:proc_dark" and "proc_dark". The different biases have different *purpose*s so that the workflow can process them separately. A given file might have several different *purpose*s if it is selected multiple times by the rules (see Fig. 1). An example of this is when the same bias frame matches the selection rules for both the flatfield and the science frames.

### 2.4. Data processing workflows

There are different ways to carry out the task of reducing astronomical data, even when the applications used for individual reduction steps are fixed. One approach is to sort data by *category*, and process each *category* in sequence. For example, one might start by processing all the bias frames for all *data set*s as the very first step, then proceed to subtract combined biases from all relevant data, and continue with, for example, producing flatfields.

A different approach is to fully process a single *data set*, performing all necessary steps to see the final result for the first *data set* in the shortest possible time. Each intermediate product, such as a combined bias, is produced only when it is needed.

The former approach has the advantage that it simplifies book-keeping, in that the only necessary initial sorting is by file type. Operations of the same kind are all performed together. The parameters for every task are optimized by repeatedly inspecting results. Once a good set of parameters is found, it is applied to all files of the same kind. This approach is efficient in the sense that identical operations are carried out only once, while the effort for bookkeeping is minimal. It is, therefore, often used when workflows are manually executed by scientists that call individual steps in sequence and book-keeping is carried out ad-hoc without software tools.

The advantage of the latter approach is that it allows for easier inspection of the impact of any change in parameters or procedures on the quality of the final *target product* of a workflow. This is particularly important when data reduction strategies are still experimental and being tested. This approach also delivers the results faster in that it only executes the steps that are needed for a given *data set* and thereby more quickly produces the *target product* for the first *data set*.

The advantages of both of these approaches can be combined with the following design. As in the second approach, data are processed one *data set* at a time. Data reduction steps that need to be executed several times with different input files from the same *data set* are carried out in succession. For example, the step to combine bias frames is executed for the biases to be applied to the science frame, and immediately afterwards the bias frames to debias the flatflields are processed, and so on. The inputs and outputs of each individual data reduction step are stored in a database for re-use later. Whenever a reduction step is called, this database is checked for previous calls to the reduction step with the same input files and parameters. If such a previous call exists, then the reduction step is not executed and instead the previous results are re-used. We call the feature to re-use products created by previous executions of a procedure the "lazy mode". There might be cases when such a re-use of products is not desired. A lazy mode should, therefore, always be an option of each individual step in a workflow. An example for the efficiency gain from using this mode is a set of combined biases that are used by the science and flatfield frames of a *data set*, and in addition

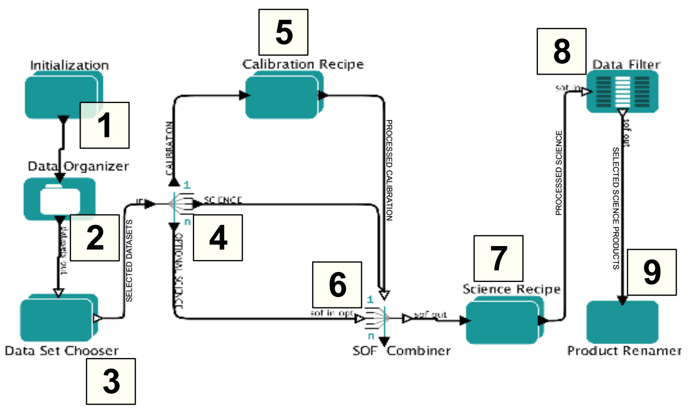


**Fig. 4.** Example of a basic Reflex workflow. The figure uses the graphical elements of a Kepler workflow (Sect. 3). The lines indicate the flow of files and are labelled by their contents. The "optional science" files are files that are used to process the science data, but the processing can proceed even if they are not available (see Sect. 3.3.3). The workflow includes two data processing steps, one for calibration and one for science processing (labelled $\boxed{5}$ and $\boxed{7}$, respectively). The elements of the workflow are: an initialization $\boxed{1}$ that sends the input directories to the data organiser, the data organiser $\boxed{2}$, a data set chooser $\boxed{3}$ that allows interactive selection of a *data set*, the file router $\boxed{4}$ that directs different *categories* of files to their destinations, a SOFCombiner $\boxed{6}$ that bundles the input for the science step, and a data filter and product renamer ($\boxed{8}$ and $\boxed{9}$, respectively) that organise the output products from the workflow.

by the calibration frame of another *data set*. The combination of biases is carried out only once, and is used in three different places. One advantage of our approach is that it is as efficient as the first of the above approaches, but produces the science results quickly and provides the user experience of the second approach. Another advantage is that subsequent runs of the workflow can use this database of intermediate products to redo the reduction with changed parameters in a very efficient manner. If a parameter or input file for any step changes, then the result for this step will change. The change in one of the intermediate products might require the re-execution of some but not all of the subsequent steps. The database can be used to automatically identify products that can be re-used from previous runs, and the steps that need to be repeated.

The implementation of this workflow design requires three levels of grouping of data. A schematic diagram of such a workflow is shown in Fig. 4. The highest level of grouping are the *data set*s, as discussed in Sect. 2.1. This task is carried out by a data organiser (step $\boxed{2}$ in Fig. 4). Subsequently, the files in each *data set* are sorted by *category* and directed to the reduction steps that need this particular *category* of files, a step that is performed by a file router (step $\boxed{4}$). This is the level that describes the data reduction strategy and is shown in the design of a workflow. Each reduction step might be called repeatedly with different input files. For that purpose, a third level of grouping is needed to group files that are processed together with separate calls of the reduction step. This is part of the functionality of the reductions steps $\boxed{5}$ and $\boxed{7}$.

## 3. Implementation

While the principles discussed in this paper do not depend on a specific software implementation, it is useful to discuss them in the context of, and with the terminology used, in a specific environment. Several software environments to design and execute workflows exist (e.g. Curcin & Ghanem 2008). For the Reflex

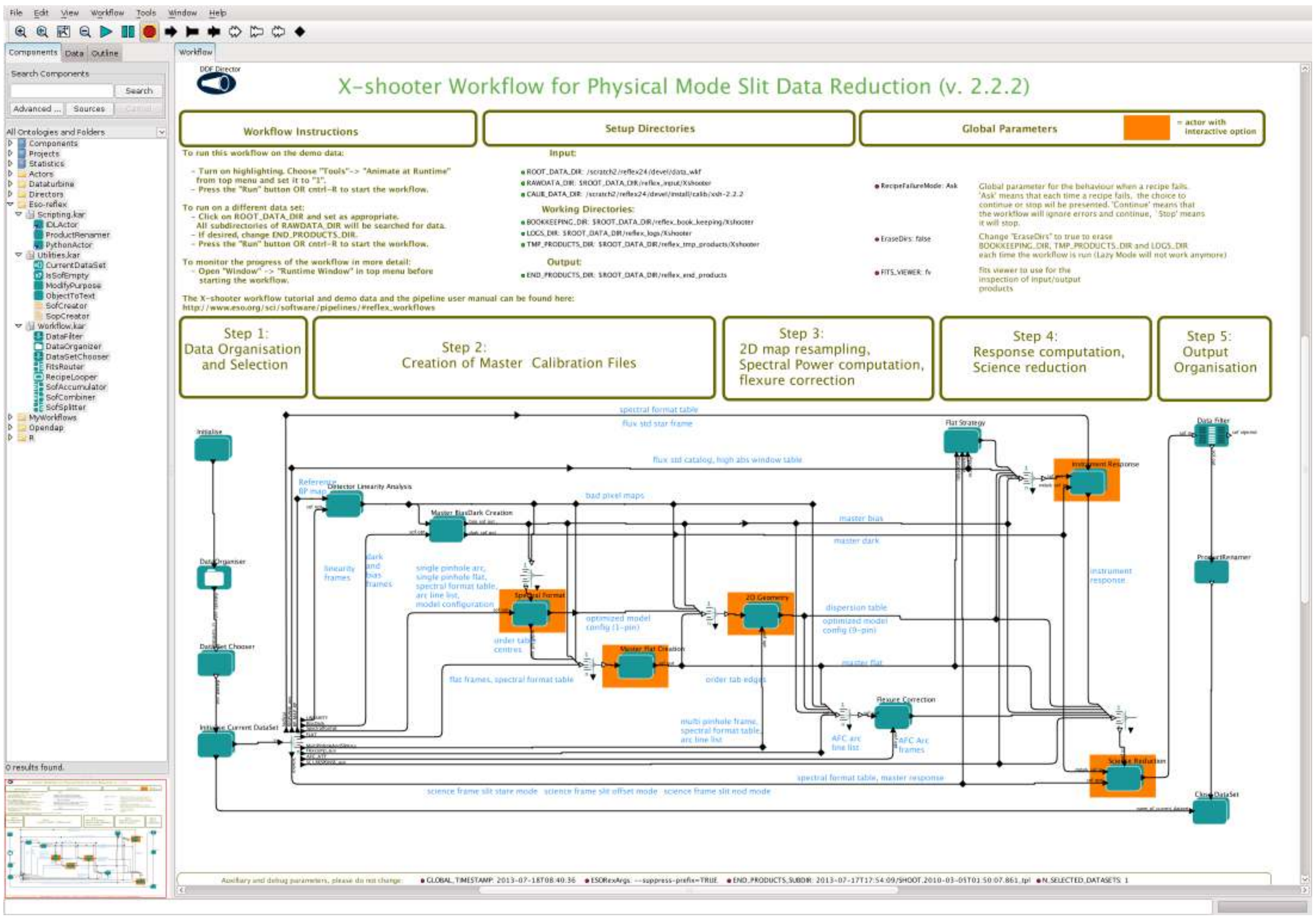

X-shooter Workflow for Physical Mode Slit Data Reduction (v. 2.2.2)
DDF Director
Workflow Instructions
Setup Directories
Global Parameters
Step 1: Data Organisation and Selection
Step 2: Creation of Master Calibration Files
Step 3: 2D map resampling, Spectral Power computation, flexure correction
Step 4: Response computation, Science reduction
Step 5: Output Organisation
Initialise
Data Organiser
Data Set Chooser
Initialise Current DataSet
Detector Linearity Analysis
Master Bias/Dark Creation
Spectral Format
2D Geometry
Master Flat Creation
Flat Strategy
Instrument Response
Flexure Correction
Science Reduction
Data Filter
ProductRenamer
Close DataSet

**Fig. 5.** Kepler user interface loaded with the Reflex workflow for ESO's X-shooter instrument. The top section defines the input directories and user preferences. It is usually sufficient to specify the raw data directory to run the workflow on a new data set. The execution of a workflow is started with the run button in the top left panel. The workflow includes 8 recipe executers that run the recipes necessary to reduce X-shooter data. Actors with an orange background include interactive steps that can display the result of the recipe, and allow for the optimization recipe settings. The workflow includes an specially implemented actor called "Flat Strategy" that is specific to the X-shooter workflow. This actor allows the user to select a flatfielding strategy. Depending on the chosen strategy, files will be routed differently.

project, we evaluated and partially implemented some of the concepts discussed in this paper in several open source workflow engines. In the end, we decided to use the Kepler workflow application (Altintas et al. 2004) to implement Reflex, because of its large suite of available components, and its robust support for conditional branching, looping, and progress monitoring. In this section, we introduce the terminology and summarize the most important features of Kepler. For more details, see the Kepler User Manual[3].

## 3.1. The Kepler workflow engine

A workflow application is a software system designed to run sequences of stand-alone programs, where the programs depend on the results of each other. Components of a workflow are representations of these programs, as well as elements that manage the results and communication between them. In Kepler, components of the workflow are called "actors". In the graphical interface, actors are represented by green boxes (see Fig. 5). Actors have named input and output "ports" to communicate with each other. The communication is implemented by exchanging objects called "tokens" that travel along connections between the output port of one actor to the input port of another actor. These connections are called "relations" and are represented by lines. Output ports emit tokens after the execution of an actor is finished, and input ports consume tokens when execution of the actor starts. The availability of tokens is a crucial factor in determining the sequence of triggering the actors.

The *relation*s between actors themselves do not define the temporal sequence of the execution of actors. A scheduler is required to trigger the execution of each actor. A scheduler in a Kepler workflow is called a "director". The terminology of Kepler follows the metaphor of film making, where a director instructs actors that carry out their parts. Reflex uses the "Dynamic Data Flow" (DDF) director that allows the workflow execution to depend on the results of actors and supports looping and iterating. The basic algorithm used by the DDF director is to repeatedly scan all actors and identify those that can be executed because all of the necessary input is available. It then selects one of the actors for execution based on minimizing unused tokens and memory. The details are extensively discussed in Zhou (2004). It should be noted that an actor of a workflow can itself

[3] `https://code.kepler-project.org/code/kepler-docs/trunk/outreach/documentation/shipping/2.4/UserManual.pdf`

be a sub-workflow. Such “composite actors” might include their own directors.

The Kepler workflow application provides a graphical interface to create, edit and execute workflows. A large number of general purpose actors are bundled with the environment. There are several ways to monitor the progress of a workflow, pause it, or stop it.

### 3.2. The reflex environment

We have produced the software package Reflex to implement the design discussed in this paper using Kepler workflows. Reflex consists of a collection of actors that support the execution of astronomical applications. A shared characteristic of commonly used astronomical applications is that they read data and metadata from FITS files, are configurable with parameters, and produce output FITS files called products. Reflex supports any application of this kind that can be started from a command line. Hereafter, we refer to such applications as “recipes”. The primary task of Reflex is to route the necessary input files to the recipes. This includes both files in a *data set* and files created during execution of the data processing workflow. In addition, Reflex is able to create and send lists of parameters to recipes.

To achieve these tasks, Reflex uses two kinds of objects called “set of files” (hereafter SOF) and “set of parameters” (hereafter SOP). These objects are used as tokens in a workflow. A SOF contains a list of files. The record for each file consists of the file name, the checksum, the *category* and a list of *purpose*s for that file. A SOP contains a list of parameters, and the record for each parameter consists of its name and value. Reflex actors use and process these objects.

The construction of an input SOF, to be fed to a recipe, needs to consider the *category* and the *purpose* of a file. For every file in a *data set*, these file properties are determined during file organisation according to the pre-defined rules. For products, these properties have to be determined during the execution of the workflow. It is important to note that these two file properties are handled differently. Every recipe needs to be aware of the file *category* of its input files. For example, a recipe that combines flatfields might use dome flat exposures and bias frames as input files, and these files need to be identified to the recipe. The mechanism to identify files to the recipes is different in different environments. For example, IRAF uses different input parameters for different file types, whereas ESO’s CPL recipes (McKay et al. 2004) use text files with file tags to identify the file types. In both cases, Reflex uses the *category* to identify these file types. Reflex workflows therefore need to explicitly use the exact names known to the recipes for its *categories*. The data organisation rules have to generate these exact names. In contrast, recipes are oblivious to the *purpose* of a file. The recipe to combine flat field frames does not need to know how and where the combined flatfields will be used. Therefore, the processing of the *purpose* is completely handled by workflow actors.

As discussed in Sect. 2.3, a *purpose* is a concatenation of *action*s used to organise the input data. The name of an *action* is arbitrary and, therefore, it is never used explicitly in the workflow. Instead, Reflex uses the overriding principle that recipes receive files of the same, but arbitrary *purpose*. Actors compare and manipulate, but never decode the *purpose*. There are three standard operations on the *purpose*, they are called *pass-through*, *set-to-universal* and *trim*. The operation *pass-through* simply reads the *purpose* of a file, and passes it on without any modification. The operation *set-to-universal* replaces an existing *purpose* with a new one with the protected name *universal*. The *universal purpose* is a wildcard that may be substituted by any other defined *purpose* depending on the circumstances. The operation *trim* modifies a *purpose* by removing the last *action* from a *purpose* that consists of at least two concatenated *action*s, and sets a *purpose* that consists of a single *action* to *universal*. Workflows are designed so that the input to any recipe consists of files with a single identical *purpose*. These operations are sufficient to design workflows that collect all necessary input files for each recipe, by selecting all files with identical *purpose*s to be processed.

The usage of these operations can best be explained with examples. The most commonly used operation is *trim*. In Sect. 2.3, we already used the example of bias frames with the *purpose* “proc_science:proc_flat” and “proc_science”, and flatfield and science frames with the *purpose* “proc_science”. When the flatfield is to be processed, the workflow selects the file with the *category* “flat”, and all files with identical *purpose*. In our example, these are the bias and flatfield files with *purpose* “proc_science:proc_flatfield”. The output product, i.e. the processed flatfield, should be assigned the *trim*med input *purpose*. In our case, the *purpose* “proc_science:proc_flat” is reduced to “proc_science”. In a subsequent step, a science recipe collects all the files with purpose “proc_science” for the input. This will include the processed flatfield file, the bias frame selected to match the properties of the science frame, and the science frame itself.

The operation *pass-through* is used for recipes that only use intermediate products as inputs. If such a recipe is needed in the chain (e.g. to smooth the flatfields in the above example), this recipe should pass-through the *purpose* of its input file to the product file, so that the *purpose* of the smoothed flatfields is still “proc_science”. In general, any recipe that has no impact on the data selection should pass-through the *purpose* of the input files.

Finally, the *set-to-universal* operation can be used for files with a unique *category* that can be processed independently of their usage in the workflow. For example, a bad pixel map that is used by many different recipes in a workflow can be given the *purpose* “universal” to simplify the workflow design.

These three operations allow an efficient and elegant assembly of input files for recipes. Different operations might be necessary under special circumstances and a flexible system will allow these to be implemented. An important design principle for any operation on the *purpose* of a file is that it should never explicitly use the name of the *purpose*. The names assigned by the rules are arbitrary, and a change of those names should not impact the workflow execution.

### 3.3. Reflex actors

In order to implement these principles, Reflex provides 17 essential actors. A complete list of actors is given in Appendix A and are described in detail by Forchi (2012). The actors can be grouped into the data organiser, actors to process and direct tokens, actors to execute data reduction recipes written in one of several supported languages, and actors that provide interactive steps in a workflow. In this section, we discuss these features and options to illustrate how the principles laid out above can be implemented in concrete software modules.

#### 3.3.1. Data organiser and rule syntax

For Reflex, we opted to implement a program DataOrganiser that carries out the organisation of local data fully automatically using a set of user-supplied, human-readable rules. The input of

**Table 1.** OCA rules syntax.

| Rule type | Syntax |
|---|---|
| Classification | `if` *condition* `then { REFLEX.CATG = "`*category*`"; REFLEX.TARGET="[T,F]" }` |
| Organisation | `minRet =` *i*`;` `select execute(`*actionname*`) from inputFiles where` *conditions* `[group by` *keyword-list*`]` |
| Association | `action` *actionname* `{` `minRet =` *i*`; maxRet =` *j*`;` `select files as` *label* `from inputFiles where` *conditions* `[closest by` *keyword*`];` `product` *label* `{ REFLEX.CATG =` *category* `};` `}` |

**Notes.** The table lists a simplified version of the OCA rules syntax appropriate for data organisation in Reflex workflows. The *conditions* define *categories* of FITS files by their header keywords and may include logical and arithmetic expressions. The *label*s in the association rule are used for logging purposes and are usually set to the *category* of the file defined in the rule.

the DataOrganiser is a set of FITS files and the classification rules; the output is a collection of *data set*s. The rules are based on the principles discussed above. It should be re-iterated that, while definitions of *action*s could be used to define a data structure in sufficient detail to allow automatic derivation of a simple data processing workflow, this is not the approach that we adopt here. Instead, the rules are used only to organise the data, while the workflow to reduce the data is not constrained to using the selected data in any particular manner.

The DataOrganiser is the first actor after the initialization in any Reflex workflow. It organises the input FITS files according to the workflow-specific input rules, and the output are *data set*s that are either marked complete or incomplete (see Sect. 2.1). The execution of a Reflex workflow is triggered by sending an input token to the DataOrganiser actor.

The DataOrganiser recognizes rules that use the syntax of a special language called OCA. The OCA language has been developed at ESO (Zampieri et al. 2006) and is designed to describe the Organisation, Classification, and Association of FITS files based on their FITS header keywords (Wells et al. 1981). OCA is used for multiple purposes within ESO's data flow system (Peron 2012), and interpreters are embedded in a number of applications. Therefore, OCA rules to organise data are available for most instruments on ESO's telescopes. The details of the language are described in Zampieri & Forchi (2012). The language has all of the features needed to define rules for data organisation. Here, we summarize a subset of the OCA language that is useful for the data organisation discussed in this paper.

The OCA language recognizes three types of rules. They are:

1. Classification rules. Classification rules define file *categories* based on any logical combination of conditions on FITS keywords. The syntax of the classification rules is given in row 1 of Table 1. The classification defines the keyword REFLEX.CATG as the *category* of the file, and this keyword can be used like any other FITS keyword in the header by other rules. A simple example for the usage of a classification rule is to assign to a file the *category* "bias" if the header keyword "EXPTIME" is set to the value "0". The classification rules also define whether a set of files is the target of the workflow or not.
2. Organisation rules. Organisation rules define *action*s and the groups of files that *trigger* them. The rules define a name for each *action* so that it can be referred to by other rules. The syntax of the organisation rules is given in row 2 of Table 1. The rules include an optional specification of the minimum number of files needed to *trigger* the *action*. This minimum number is used to determine whether a *data set* is complete or not. There is no maximum number because there are no defined criteria to select among files that match the condition. A simple example is to group at least 3 dome flat frames by filter, and *trigger* an *action* called "proc_flat" that combines flatfields.
3. Association rules. Association rules define "associated files", i.e. input files and *product*s that are needed by an *action* in addition to the *trigger*. The syntax of the association rules is given in row 3 of Table 1. There is an unlimited number of "select" statements that define conditions to select files. In addition to the conditions, a "closest by" statement can be used to select those files that have a value for a given keyword that is as close as possible to that of the *trigger*. If there is no "closest by" statement, then the time of observation will be used to select among several files that satisfy the conditions. Each select statement can be preceded by an optional specification of the minimum and maximum number of files needed for each *category*. This mechanism allows to define optional input files that are not essential for a workflow but that will be used if present. The association rules also define names for *categories* of *products* that can be referred to by other rules. A simple example is that the proc_flat *action* needs a combined bias frame and produces a *product* "MasterFlat".

These rules are sufficient to describe the data graphs discussed in Sect. 2.1 and shown in Figs. 1 and 2. In Table 2, we show as a specific example the rules that describe the data organisation for an image that needs a flatfield and a dark frame for its processing, as discussed in Sect. 2.1 and shown in Fig. 2. The first block classifies available files as "science_image", "flat" or "dark" based on the header keyword "TYPE", and as "bias" based on the fact that the value of the header keyword "EXPTIME" is 0. The next three "select" statements define the three *actions* "proc_dark", "proc_flat" and "proc_image", and their *triggers* "flat", "dark" and "science_image". What follows are the association rules that specify that the *action* "proc_dark" needs a bias as input and outputs a product called "processed_dark", the *action* "proc_flat" needs this "processed_dark" and a bias, and outputs a "processed_flat". Finally, the action "proc_image" needs a "bias", the "processed_dark" and the "processed_flat". The association rules also specify that darks are selected to match the exposure time of the "science_image" or the "flat", and flats are selected to match the filter of the "science_image". The application

**Table 2.** Simple example of OCA rules.

Classification rules

```
if TYPE=="OBJECT" then {REFLEX.CATG = "science_image";  REFLEX.TARGET="T";}
if TYPE=="FLAT"  then {REFLEX.CATG = "flat";}
if TYPE=="DARK"  then {REFLEX.CATG = "dark";}
if TYPE=="CALIB" and EXPTIME==0 then {REFLEX.CATG = "bias";}
```

Organisation rules

```
select execute(proc_dark) from inputFiles where REFLEX.CATG=="dark"
select execute(proc_flat) from inputFiles where REFLEX.CATG=="flat"
select execute(proc_image) from inputFiles where REFLEX.CATG=="science_image"
```

Association rules

```
action proc_dark
{
  select files as bias from inputFiles where REFLEX.CATG=="bias" ;
  product processed_dark { REFLEX.CATG="processed_dark";}
}

action proc_flat
{
  select files as bias from inputFiles where REFLEX.CATG=="bias" ;
  select files as processed_dark from inputFiles where REFLEX.CATG=="processed_dark"
                                             and inputFile.EXPTIME==EXPTIME;
  product processed_flat { REFLEX.CATG="processed_flat";}
}

action proc_image
{
  select files as bias from inputFiles where REFLEX.CATG=="bias" ;
  select files as processed_dark from inputFiles where REFLEX.CATG=="processed_dark"
                                                       and inputFile.EXPTIME==EXPTIME;
  select files as processed_flat from inputFiles where REFLEX.CATG=="processed_flat"
                                                       and inputFile.FILTER=FILTER ;
}
```

**Notes.** The table lists an example of a set of OCA rules that can produce the data organisation shown in Fig. 2.

of these rules can lead to a data set organized as shown in Fig. 2. However, the same rules can also lead to a data graph with different topology for a different data set. For example, if both the science image and the flat have the same exposure time, the application of the rules might select the same dark frame for both the flat and the science image. The power of the data organiser is to use such abstract rules to select optimal data sets based on the metadata of the available files.

### 3.3.2. Actors for data processing

The purpose of a data processing workflow is to execute a series of recipes. The recipes can be written in any language, but must accept the basic input, and provide the basic output information, needed to run the workflow. In particular, the recipes must accept FITS files that are categorized, and generate products as FITS files and the information to categorize them. Reflex provides three actors to execute recipes. They are called "PythonActor", "IDLActor" and "RecipeExecuter". The PythonActor is used to run Python scripts that, in turn, can call, for example, shell commands, IRAF tasks via the PyRAF interface (White & Greenfield 2002), or MIDAS programs via the pyMIDAS interface (Hook et al. 2006). The IDLActor is used to run IDL programs, and the RecipeExecuter executes CPL recipes. The basic function of all three actors is to filter and send the files of the input SOF to the recipe, and create and emit the output SOF with the products of the recipe. The *purpose* of the product files is constructed from those of the input files using one of the standard operations described in Sect. 3.2. All CPL recipes can be queried to report their input and output in a well-defined format. This feature is used by Reflex to automatically generate parameter lists and ports. For Python and IDL, simple interfaces are provided that can be added to any program.

### 3.3.3. Actors for file routing

The top level task when designing a data processing workflow is to decide on the cascade of file processing, i.e. the routing of files by *category*. In Reflex, users of a workflow are presented with a visual diagram that shows the directional flow of files with different *categories* to the corresponding data processing actors (see Fig. 4). The *data set*s created by the DataOrganiser are SOFs that contain a full set of file *categories*. An actor is needed to direct the different *categories* of files in a *data set* to the respective data processing actors. In Reflex, this actor is called the FitsRouter. It takes a single *data set* SOF as input, and creates SOFs that contain input files selected by *category* from the *data set*. Different output SOFs are emitted from separate ports, that are connected to data processing actors. For each output port, one, or several, file *categories* sent to this port are explicitly specified by name. The primary use of the FitsRouter is to select the *categories* of raw files in a *data set* that are needed for each data processing actor, whereas products needed as input arrive directly from the data processing actor via dedicated *relation*s.

The routing by *category* assures that a recipe receives all necessary file *categories* at the time that it is executed. If there are files with *categories* that are not needed by the recipes, they can be filtered out by the data processing actor. What remains is the task to select among all of the files of a given *category* those that should be processed together by the recipe. In Reflex, this is implemented as an actor "SOFCombiner" that bundles the different input files for a recipe into a single SOF. The SOFCombiner has two input ports, one is for mandatory files and another one for optional files. Both of them are multiple ports, i.e. several *relation*s can be connected to either input port. The tokens sent via different *relations* to a multiple port are in different *channels* of the port. The SOFCombiner creates a single output SOF that includes input files selected by *purpose*. The selection rule is that only files with a *purpose* that is present at each of the input *channels*, at the mandatory input port, are passed. The desired selection of all files with the appropriate *purpose* is achieved when at least one of the input channels includes only files that are the necessary input for the recipe, typically the *trigger* for the recipe. All other channels can include any file, and the SOFCombiner automatically selects the correct input for the recipe. The algorithm used by the SOFCombiner uses comparison of *purpose*s as the only method and consists of the following two steps.

1. Find *purpose*s that are present at each input *channel* of the mandatory input. A *universal purpose* counts as a match to any other *purpose*.
2. Send all files, both from the mandatory and optional port, that match any *purpose* found in step 1 to the output SOF. Again, a *universal purpose* counts as a match to any other *purpose*.

This simple but powerful algorithm assures that the files of the same *purpose* in the output SOF are necessary and sufficient to run the intended data processing recipe. This fact is then used by a combination of two actors called "SOFSplitter" and "SOFAccumulator". The former splits an input SOF by *purpose* and emits a separate SOF for each *purpose*. The latter collects several SOFs that arrive at the same port and combines them into a single SOF. These actors are used in combination with, and always bracket, a data processing actor (see Fig. 6). The net effect of this combination is that the recipes called by the data processing actor are executed multiple times, and the result is a single SOF that includes all of the products. This SOF can then be used by the next SOFCombiner to select the files for the next data processing step.

The algorithms discussed above are an elegant and efficient way to implement the most common routing needs without repetition of information. In addition, explicit operations on file properties can be used to implement special needs. A common application is conditional routing of files, i.e. workflows in which files are routed differently depending on some data properties or user choices. Kepler provides a large number of general purpose actors to implement conditional and/or iterative branches in a workflow, and stand-alone actors, to manipulate the *category* or *purpose* of a file, are either provided by Reflex or can easily be implemented (e.g. as a Python script). As emphasized earlier, in any manipulation of the *purpose*, the *purpose* should never explicitly be called by name within the workflow to avoid unnecessary dependencies of the workflow on syntax choices in the rules. For example, the case discussed above that a flatfield file is selected to be taken close in time to the science spectrum, but is used to flatfield the flux calibration file can easily be implemented with such a customized *purpose* processing script.

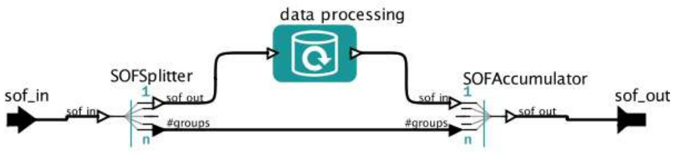
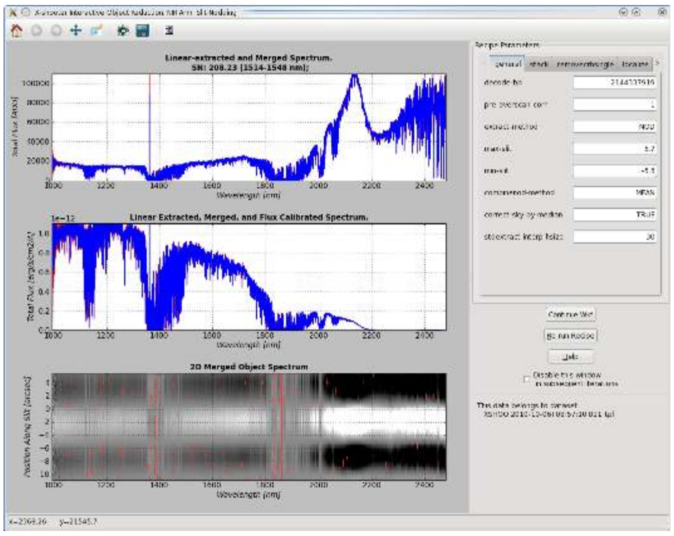


**Fig. 6.** A data processing actor embedded in a SOFSplitter and SOFAccumulator to manage repetitive executions for files with different *purpose*s.


Linear-extracted and Merged Spectrum.
SN: 208.23 [1514-1548 nm];
Total Flux [ADU]
Wavelength [nm]
Linear Extracted, Merged, and Flux Calibrated Spectrum.
Total Flux [erg/s/cm2/A]
Wavelength [nm]
2D Merged Object Spectrum
Position Along Slit [arcsec]
Wavelength [nm]
Recipe Parameters
general
Continue Wkf
Re-run Recipe
Help
Disable this window in subsequent iterations

decode-bp 2144337919
pre-overscan corr 1
extract-method NOD
combine-nod-method MEAN
correct sky by median TRUE
slitextract interp hsize 30

**Fig. 7.** Example of an interactive interface created with the Reflex Python library. The plots on the left hand side can be interactively manipulated to inspect the data products in different ways. The panel on the right hand side allows the user to modify recipes parameters, and re-execute the recipe or continue with the workflow.

### 3.3.4. Interactive actors

One reason why automated workflows are an efficient way of data reduction is that the user can intercept the processing at any stage and interact with the workflow. A major contribution to the interactive user experience comes from the workflow application that provides tools to monitor, pause and modify the workflow itself. Additional tools are needed to provide application-specific ways to inspect and influence the execution of the workflow. Reflex provides several interactive actors, and a Python library to implement actors that can create customized plots and allow recipe parameters to be modified during the execution of a workflow. An example of an interface created with this library is shown in Fig. 7. Ready-to-use interactive actors that have been developed for Reflex include the DataSetChooser to interactively inspect and select *data set*s, the DataFilter to inspect and filter SOFs, and the ProvenanceExplorer to inspect the provenance of a product and its history from repeated runs of a workflow. All interactive actors and features can easily be turned off when starting Reflex, to allow a workflow to be run in batch mode once it has been adapted and optimized.

### *3.4. Modularity of reflex workflows*

Kepler provides an easy way to create modular workflows. A composite actor is an actor that itself contains a workflow, and composite actors can be nested to arbitrary depth. Placing each data processing actor, together with its supporting actors, into a composite actor leads to a clean and intuitive view of the whole data processing workflow.

The layout of a workflow, whether it is modular or not, does not uniquely define a sequence of actor executions. For example, a scheduler might decide to alternate processing of actors contained in different composite actors. However, workflow execution is more intuitive when each composite actor is completed before processing proceeds to other actors. In Kepler, this can be achieved by placing an appropriately configured director into each composite actor.

### *3.5. Book-keeping and product organisation*

The efficiency of the workflow execution relies on rigorous book-keeping that stores relevant information in a database for easy retrieval and processing. During execution of the workflow, the input and output files of each step in the workflow, as well as all parameters used for the processing are stored in the database. For each file, the file name and the checksum are recorded.

The two main uses of the database are the implementation of the lazy mode described in Sect. 2.4, i.e. the keeping track of products for later re-usage, and the organisation of the output files in a user-friendly way. For the lazy mode, checksums and creation dates can be used to detect changes in input files of the same name. The main output of a workflow are the files produced by the final science processing step, i.e. the science data products of a workflow. Intermediate products produced by previous steps are often needed to evaluate the science data products, troubleshoot, or investigate the optimization of the products. For that purpose, each science data product should be associated with the input data and parameters used to generate it. The input files might themselves include products from previous steps, that are associated to the input of that step. At the conclusion of a workflow, all files used and produced during its execution can be organised in a directory tree that can be browsed either with a specialized tool or with a file browser.

## 4. Summary and conclusions

In this paper, we describe how a workflow application can be used to automate an astronomical data processing workflow. We propose a specific design for such a workflow, and present the application Reflex that implements this design within the Kepler workflow engine. The key advantages of automated workflows over alternative methods such as scripting or monolithic data processing programs are the built-in tools for progress monitoring, and the ability to modify the data organisation and data flow efficiently.

The specific advantages of our Reflex implementation include:

1. Selecting and organising the input data is a significant task for any astronomical data reduction. A rule-based data organiser is used to formalize the selection criteria and to fully automate the organisation of data. The automated data organisation can be followed by an interactive step to inspect and modify the chosen *data set*s.
2. Reflex allows users to monitor the progress of data reduction, interact with the process when necessary, and modify the workflow. A graphical user interface can be used to develop and experiment with workflows. At the same time, workflows can be executed in a completely non-interactive batch mode to allow processing of large data sets and/or for computational time-intensive processing.
3. Re-reduction, after a change in input files or parameters, is efficiently carried out by only re-running those steps that are affected by this change. A modern reduction process might use hundreds of files with dozens of different *categories*, and any number of data reduction steps. Changing a single parameter in one of the steps or switching a single input file might trigger a complex cascade of necessary re-runs of steps. Recognizing those steps and re-executing them is fully automated in Reflex.

The execution time of the data organiser strongly depends on the complexity of the rules, the total number of files, and the files in each category. For data from a typical observing run, the first time data organization might take on the order of a minute on a typical desktop workstation. In subsequent runs the lazy mode will reduce this time by a very large factor. The execution of the data processing workflow itself adds a fraction of a second to the stand-alone execution of each recipe. The default memory allocation for Reflex is 1536 MB in addition to the memory requirement of the recipes. This allocation can be reduced for simple workflows if necessary.

So far, Reflex workflows have been developed for the most commonly used instruments on ESO's Very Large Telescope (VLT), namely FORS2, SINFONI, UVES, VIMOS, and X-shooter, as well as the newly commissioned KMOS. They are distributed to users to provide them with a pre-packaged workflow that works out-of-the-box to reduce VLT data. All ESO Reflex workflows are intuitive to understand, as each includes a detailed tutorial and a comprehensive demonstration data set. As such, even novice users can easily modify and experiment with the workflows. Reflex workflows are bundled with the corresponding instrument pipelines and the Reflex environment. The whole package can be installed with a single installation script[4]. ESO expects to develop Reflex workflows for all future VLT instruments.

*Acknowledgements.* The Kepler software is developed and maintained by the cross-project Kepler collaboration, which is led by a team consisting of several of the key institutions that originated the project: UC Davis, UC Santa Barbara, and UC San Diego. We acknowledge useful discussions with Reinhard Hanuschik on some concepts discussed in this paper.

## Appendix A: List of Reflex actors

A complete description of Reflex actors is given by Forchi (2012). Here, we list the standard Reflex actors in alphabetical order.

– *DataFilter*: interactive actor to inspect and select FITS files.
– *DataOrganiser*: implementation of the rule-based data organiser as described in text.
– *DataSetChooser*: interactive actor to inspect files in a *data set*, edit the selection, and select *data set*s to be reduced.
– *FitsRouter*: actor to route files by *category*.
– *IDLActor*: interface to configure and execute IDL scripts.

[4] http://www.eso.org/reflex/

– *IsSofEmpty*: actor that checks whether an SOF contains files. This actor is used to implement different data flows depending on the availability of some data.
– *ObjectToText*: actor to present Reflex tokens in human readable form.
– *ProductRenamer*: actor for renaming FITS files based on keywords of the file.
– *ProvenanceExplorer*: interactive actor to inspect products produced during a current or previous run of the workflow.
– *PythonActor*: interface to configure and execute Python scripts.
– *RecipeExecuter*: interface to configure and execute CPL recipes.
– *RecipeLooper*: actor to implement looping over one or several recipes.
– *SofCreator*: actor to create a Reflex Set of Files (SOF) token from a directory with files.
– *SopCreator*: actor to create Reflex Set of Parameter (SOP) tokens.
– *SOFAccumulator*: actor to create a single SOF out of several input SOFs that arrive in sequence.
– *SOFCombiner*: actor to create a single SOF out of several SOFs that are available simultaneously.
– *SOFSplitter*: actor to split an SOF by file *category*.

## References

Abhishek, T., & Sekharb, A. 2007, Comput. Biol. Chem., 31, 305
Altintas, I., Berkley, C., Jaeger, E., et al. 2004, in Scientific and Statistical Database Management, 16th International Conference, 423
Ballester, P., Bramich, D., Forchi, V., et al. 2011, Astronomical Data Analysis Software and Systems XX, 442, 261
Banse, K., Crane, P., Grosbol, P., et al. 1983, The Messenger, 31, 26
Barseghian, et al. 2009, Ecol. Inform, 5, 3
Biretta, J. A., Baggett, S. M., MacKenty, J. W., Ritchie, C. E., & Sparks, W. B. 1994, in Calibrating Hubble Space Telescope, eds. A. Koratkar, & C. Leitherer, 8
Cavanagh, B., Jenness, T., Economou, F., & Currie, M. J. 2008, Astron. Nachr., 329, 295
Curcin, V., & Ghanem, M. 2008, Biomedical Engineering Conference, CIBEC, Cairo International, 1
Forchi, V. 2012, Reflex User Manual, ESO internal document VLT-MAN-ESO-19000-5037
Hook, R. N., Maisala, S., Oittinen, T., et al. 2006, Astronomical Data Analysis Software and Systems XV, 351, 343
Jung, Y., Ballester, P., Banse, K., et al. 2004, Astronomical Data Analysis Software and Systems (ADASS) XIII, 314, 764
Ludäscher, B., Altintas, I., Berkley, C., et al. 2005, Concurrency and Computation: Practice and Experience, 18, 1039
McFarland, J. P., Verdoes-Kleijn, G., Sikkema, G., et al. 2013, Exp. Astron., 35, 45
McKay, D. J., Ballester, P., Banse, K., et al. 2004, Proc. SPIE, 5493, 444
Nastasi, A., Scodeggio, M., Fassbender, R., et al. 2013, A&A, 550, A9
Peron, M. 2012, Astronomical Data Analysis Software and Systems XXI, 461, 115
Rose, J., Akella, R., Binegar, S., et al. 1995, Astronomical Data Analysis Software and Systems IV, 77, 429
Schaaff, A., Verdes-Montenegro, L., Ruiz, J. E., & Santander-Vela, J. 2012, Astronomical Data Analysis Software and Systems XXI, 461, 875
Schmithuesen, O., Erben, T., Trachternach, C., Bomans, D. J., & Schirmer, M. 2007, Astron. Nachr., 328, 701
Scodeggio, M., Franzetti, P., Garilli, B., et al. 2005, PASP, 117, 1284
Scott, D., Pierfederici, F., Swaters, R. A., Thomas, B., & Valdes, F. G. 2007, Astronomical Data Analysis Software and Systems XVI, 376, 265
Tody, D. 1993, Astronomical Data Analysis Software and Systems II, 52, 173
Tucker, D. L., Kent, S., Richmond, M. W., et al. 2006, Astron. Nachr., 327, 821
Tsapras, Y., Street, R., Horne, K., et al., 2009, Astron. Nachr., 330, 4
Wells, D. C., Greisen, E. W., & Harten, R. H. 1981, A&AS, 44, 363
White, R. L., & Greenfield, P. 2002, The PyRAF Tutorial, http://stsdas.stsci.edu/stsci_python_sphinxdocs_2.13/docs/pyraf_tutorial.pdf
Zampieri, S., Chuzel, O., Ferguson, N., et al. 2006, Astronomical Data Analysis Software and Systems XV, 351, 196
Zampieri, S., Forchi, V., 2012, OCA User manual, ESO internal document VLT-MAN-ESO-19000-4932.
Zhou, G., 2004 Master Thesis, University of California at Berkeley