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

OCA User Manual

VLT-MAN-ESO-19000-4932

Issue 2.0
20/02/2013
24 pages

	Name	Date	Signature
Prepared:	S. Zampieri, V. Forchì	20/02/'13	
Approved:	T. Bierwirth	22/02/13	
Released:	M. Peron	22/02/13	

CHANGE RECORD

Issue	Date	Affected Paragraphs(s)	Reason/Initiation/Remarks
1.0	18/01/2010	All	First version
2.0	20/02/2013	All	Added feedback from reviewers, added new OCA features

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Purpose of the document

This document describes the OCA (Organization, Classification and Association) language, which is used at ESO to define data organization and association rules for the following applications:

- Pipeline processing
- Preparation of data packages
- Calibration selection for archive requests
- Data organization in Reflex
- Etc.

1.2 Scope of this document

The OCA language is used by various applications in different operational scenarios. This document focuses mainly on the generic (application-independent) aspects of the language and does not provide a detailed description of the individual OCA-based applications.
An overview of the main OCA applications is however provided in Chapter 4.

1.3 Definitions, acronyms and abbreviations

1.3.1 Definitions

A **domain-specific programming language** (**domain-specific language**, **DSL**) is a programming language designed for, and intended to be useful for, a specific kind of task. This is in contrast to a general-purpose programming language, such as C or general-purpose modeling languages like UML.

1.3.2 Acronyms

OCA	Organization Classification Association
DO	Data Organizer
DFO	Data Flow Operations
DFS	Data Flow System
AB	Association Block
RB	Reduction Block
FITS	Flexible Image Transport System
DSL	Domain Specific Language
PI	Principal Investigator

1.4 Applicable documents

The following documents of the exact issue shown form a part of this document to the extent specified herein. In the event of conflict between the documents referenced herein and the contents of this document, the contents of this document shall be considered as superseding requirement.

No	Document Title	Reference

1.5 Reference documents

The following documents contain additional information and are referenced in the text:

No	Document Title	Reference
RD1.	OCA User Requirements Document	
RD2.	C Preprocessor	http://en.wikipedia.org/wiki/C_preprocessor
RD3.	GCC online documentation	http://gcc.gnu.org/onlinedocs/

2. OVERVIEW

OCA is a domain specific language (DSL) for **classifying**, **organizing** and **associating** astronomical data based on their meta-data (FITS keywords). The OCA interpreter is embedded in various applications (DataOrganizer, ABbuilder, CalSelector, etc.) used to organize data for pipeline processing or for preparing PI data packages. The OCA rules are normally stored in instrument-specific configuration files which are parsed and interpreted by the application at run-time.
A typical OCA application operates on two sets of data:

- **Raw/Input Files**. The files to be classified and organized for pipeline processing or data packing.
- **Calibration Files**. The files which can be associated to the raw files.

From the OCA point of view, a **file** is a *collection of properties* (keyword/value pairs) which refer to an astronomical file, usually in FITS format. An example OCA file is provided hereafter:

```
FILENAME = "/data/raw/file.fits"
DPR.CATG = "CALIB"
DPR.TYPE = "BIAS"
MJD-OBS  = 53847.61567130
```

Normally an OCA file contains a subset of the FITS header keywords, plus additional keywords generated by the OCA application. Although the OCA language is targeted to astronomical applications and data, its design is quite generic and it would be also possible to use it in a completely different context.

The OCA grammar was designed having in mind the data acquisition and reduction process.
Calibrations and observations are acquired on the mountain with template files that specify the sequence of operations to acquire a given type of frame. Those frames are uniquely identified by a set of a few FITS keywords.
Every type of frame has a corresponding pipeline recipe, capable of reducing such frames and to generate master calibrations and other products.

The statements of the OCA language are called *rules*. There are three types of rules:

1. **Classification Rules**. To *classify* the input files, i.e. define new properties based on existing ones.
2. **Organization Rules**. To *organize* an input data set into homogeneous groups and trigger an action (association) to be applied to each group.
3. **Association/Action Rules**. To *associate* files and other information (recipe, products) to an input data set (a group created by an organization rule).

A generic OCA application normally implements the following sequence of operations[1]:

1. Parse OCA rules
2. Load and classify input files
3. Load and classify calibration files
4. Organize input files into homogeneous groups (association blocks)
5. Add calibration files and other information to association blocks
6. Save the results in an application-specific format (e.g. Reduction Block file or Association Block file)

The remainder of this document is organized as follows:

- Chapter 3 describes the OCA static structure (syntax) and how rules are evaluated at run-time by the OCA interpreter

[1] Specific OCA applications can implement a different workflow. For example the Archive Data Organizer implements only the classification step.

- Chapter 4 gives an overview of the main OCA applications
- Chapter 5 describes how the C preprocessor can be used to organize large sets of OCA rules in smaller and more manageable units
- The remaining chapters (appendixes) provide some examples of OCA rules, Reduction Blocks, etc.

3. OCA LANGUAGE

The syntax and semantics of the OCA language are described in the following sub-chapters:

- 3.1 File structure
- 3.2 Lexical conventions and syntax
- 3.3 OCA Rules

3.1 File structure

As shown in Table 1, an OCA script is made of three sections, one for each type of OCA rule:

- Classification rules;
- Organization/Grouping rules;
- Action/Association rules.

OCA Rules

Classification Rules

```
// line comment
if condition then KEYW.ONE = value;
/*block
Comment */
if condition then { KEYW.ONE = value; KEYW.TWO = value }
if condition then
{
  KEYW.ONE = value;
  KEYW.TWO = value
}
Etc.
```

Organization/Grouping Rules

```
// comment
select execute(action) from inputFiles where condition [group by keyword-list];
select execute(action) from inputFiles where condition [group by keyword-list];
Etc.
```

Action/Association Rules

```
action action
{
  minRet = value; maxRet = value;
  select file as alias from data-set where condition;
  minRet = value; maxRet = value;
  select file as alias from data-set where condition;
  Etc.
  recipe name { "option1"; "option2"; Etc. };
  product name { KEYW.ONE = value; KEYW.TWO = value };
  product name { KEYW.ONE = value; KEYW.TWO = value };
  Etc.
}

action action
{
  Etc.
}
```

Table 1 File structure

3.2 Lexical conventions and syntax

3.2.1 Line Structure

Each statement is terminated with a semicolon (';').

Multiple statements (e.g. action rules) can be enclosed in braces ('{', '}'). In this case a semicolon is not needed after the closing bracket.

3.2.2 Comments

Comment operators are used to add notes and documentation to source code. Everything marked as comments is ignored by the OCA interpreter.
The OCA language has two types of comments: line and block.
A line comment starts with two forward slashes (//) and ends at the end of the line.
A block comment starts with a forward slash and asterisk (/*) and ends with an asterisk and forward slash (*/).

3.2.3 Identifiers, FITS keywords and reserved words

Please note that OCA is a case sensitive language: this feature applies to all the three tokens described in this chapter.

Identifiers are symbolic names used to identify and cross-reference **actions**, **recipes** and **products**. Identifiers start with a letter and can include letters, numbers and the underscore ('_') character. Some examples of identifiers are:

```
ACTION_SW_DARK
isaac_img_dark
MASTER_DARK
```

FITS keywords are used in **conditional expressions, assignments and keyword lists**. Keywords shall be written using the short FITS notation[2], like in the following examples:

```
INSTRUME
DPR.CATG
MJD-OBS
DET.CHIPS
DET.WIN1.BINX
```

The following words are reserved and cannot be used as identifier names:

if	**then**	**or**	**and**
like	**execute**	**select**	**from**
where	**group**	**by**	**minRet**
maxRet	**as**	**recipe**	**product**
action	**file**	**inputFile**	**inputFiles**
calibFiles	**rawFiles**	**is**	**undefined**
string	**integer**	**float**	**boolean**
regexp	**between**	**not**	

Table 2 : Reserved words

3.2.4 Data Types

The following data types are supported:

[2] In the long FITS notation all keywords are referred to as they are written in the FITS header, in the short FITS notation hierarchical keywords are modified by removing the leading HIERARCH.ESO., for more detailed information please refer to http://archive.eso.org/cms/tools-documentation/dicd_v5.pdf

- **Character string**. A character string consists of arbitrary text within double quotes. Leading and trailing blanks are not significant. The escape character for double quotes is '\'. The escape sequence for '\' is '\\'.
- **Integer number**. An integer consists of a `+' or `-' sign, followed by one or more digits with no embedded spaces. The leading `+' sign is optional. Leading zeros are permitted, but are not significant.
- **Floating Point Number**. A floating point number is represented by a decimal number followed by an optional exponent, with no embedded spaces. A decimal number consists of a `+' or `-' sign, followed by a sequence of ASCII digits containing a single decimal point (`.'), representing an integer part and a fractional part of the floating point number. The leading `+' sign is optional. At least one of the integer part or fractional part must be present. If the fractional part is present, the decimal point must also be present. If only the integer part is present, the decimal point may be omitted (in this case the number will be considered to be an integer, unless it is followed by an exponent and/or one of the following letters: 'f', 'F', 'd', 'D'). The exponent, if present, consists of the letter 'E' or 'e' followed by an integer.

3.2.5 Operators

The following operators are supported:

Operator	Operand Types	Description	Context	Example
and	Logical, Logical	Logical and	Expressions	`KWD1==2 and KWD2==3`
or	Logical, Logical	Logical or	Expressions	`KWD1==2 or KWD2==3`
==	Numeric, Numeric String, String	Equals. Returns true if both operands are defined and have the same value (beware of rounding errors when comparing floating point numbers).	Expressions	`KWD == 1`
!=	Numeric, Numeric String, String	Not equals	Expressions	`KWD != 1`
?=	Numeric, Numeric String, String	Possibly equals. If both operands are defined it behaves as equals, otherwise it doesn't do anything.	Expressions	`KWD ?= 1`
<	Numeric, Numeric	Less than	Expressions	`KWD < 5`
<=	Numeric, Numeric	Less than or equal	Expressions	`KWD <= 6`
>	Numeric, Numeric	Greater than	Expressions	`KWD > 5`
>=	Numeric, Numeric	Greater than or equal	Expressions	`KWD >= 5`
is undefined	Identifier	Unary operator. Returns true if the operand is undefined.	Expressions	`KWD is undefined`
is TYPE	Identifier	Unary operator. Returns true if the operand is of the given type. TYPE can be one of the following: boolean, integer, float, string.	Expressions	`KWD is integer`
like	String, String	Can be used to perform pattern matching. The right operand is a character string which may contain the wildcard character "%" which matches any string of zero or more characters.	Expressions	`KWD like "%DARK%"`

		To match the special character "%" one may use "%%".		
regexp	String, String	Extended pattern matching functionality using regular expressions.	Expressions	`KWD regexp "[A-Za-z]+"`
=	Identifier, String Identifier, Numeric	Assignment.	Classification Rules Product Definition Rules.	`DO.CATG = "BIAS"`
+	Numeric, Numeric	Addition	Expressions	`KWD + 3`
-	Numeric, Numeric	Subtraction	Expressions	`KWD1 - KWD2`
*****	Numeric, Numeric	Multiplication	Expressions	`KWD*2`
/	Numeric, Numeric	Division	Expressions	`KWD/2`
%	Numeric, Numeric	Remainder	Expressions	`KWD%10`
between	Numeric, Numeric, Numeric	Returns true if the first operator is greater than the second and lesser than the third	Expressions	`KWD between 5 and 10`
not	Expression	Returns true if the following expression is false	Expressions	`not KWD == 5`

Table 3 : Operators

The order of operation (precedence rules) for OCA operators is listed in Table 4. All operators are evaluated from left to right and are listed in the table from highest to lowest precedence. That is, operators listed first in the table are evaluated before operators listed after.

Operator	Name
x*y, x/y, x%y	Multiplication, Division, Remainder
x+y, x-y	Addition, Subtraction
x==y, x!=y, x?=y, x<y, x>y, x<=y, x>=y, x like y, x is undefined, is type, x regexp y, x between y and z	Equals, Not Equals, Possibly Equals, Less Than, Greater Than, Less Than or Equal, Greater Than or Equal, Like, Is Undefined, Is Type, Regexp, Between
not expression	Logical Not
x and y	Logical And
x or y	Logical Or

Table 4 : Order of evaluation (highest to lowest)

The natural order in which an expression is evaluated, given by the operators' precedence, can be modified through the use of parentheses.
Example:

```
DPR.CATG=="CALIB" and DPR.TYPE like "%FLAT%" and (DPR.TECH=="MOS" or DPR.TECH=="IFU")
```

3.3 OCA Rules

3.3.1 Classification rules

Classification rules are **if statements** used to define new properties (*meta-keywords*) of files based on a conditional expression. The syntax is as follows:

```
if condition then assignment
```

Or

if *condition* **then {** *assignments* **}**

Braces **must** be used to enclose multiple assignments; otherwise they can be omitted in case just one assignment is defined.

The **condition** is a Boolean expression where the following syntax elements can be used:

- FITS keywords (e.g. **DPR.CATG**)
- Literals (strings or numeric values)
- Logical operators (**and or**)
- Comparison operators (e.g. **== >=**)
- Arithmetic operators (e.g. **+ -**)

Example:

```
DPR.CATG == "CALIB" and DPR.TECH == "SPECTRUM" and DPR.TYPE like "%FLAT%" and DPR.TYPE like "%LAMP%"
```

The **assignment** is of the form:

Keyword = Value;

Example:

```
DO.CATG = "SCIENCE";
```

Or

Keyword = Expression;

Example:

```
SECS = (MJD-OBS - 40587) * 86400;
```

An example of classification rules is shown in Table 5:

```
if DPR.CATG == "CALIB" and DPR.TYPE == "BIAS" and DET.CHIPS == 1 then
{
  RAW.TYPE = "BIAS"; DO.CATG = "BIAS_BLUE";
}
if DPR.CATG == "CALIB" and DPR.TYPE == "BIAS" and DET.CHIPS == 2 then
{
  RAW.TYPE = "BIAS"; DO.CATG = "BIAS_RED";
}
if DPR.CATG == "CALIB" and DPR.TYPE == "FLAT" and DET.CHIPS == 1 then
{
  RAW.TYPE = "FLAT"; DO.CATG = "FLAT_BLUE";
}
if DPR.CATG == "SCIENCE" and DPR.TYPE like "OBJECT%" then DO.CATG = "SCIENCE";
```

Table 5 Example of classification rules

Note: if the classification tries to overwrite a real FITS keyword, this is instead saved as a metakeyword, and its value will be ignored by the organization and association rules. Anyway custom application may still take advantage of this.

3.3.1.1 Evaluation

Given a set of classification rules and a set of files to classify, **the OCA interpreter applies all rules to all files**, one file at a time. In this context a *file* is a set of meta-data, typically corresponding to a FITS header. For each file the rules are evaluated sequentially in the same order as they appear in the configuration file. For each file and for each classification rule, the conditional part of the rule is evaluated with the actual keyword

values of the file. If the condition evaluates to true, the corresponding assignments are performed and new (meta-) keywords are added to the file. The new meta-data become immediately available for evaluation of subsequent rules.

3.3.1.2 Note on the order of classification rules

Depending on the file properties and on the order in which classification rules appear in the configuration file, it may happen that a classification rule overwrites some meta-data defined by a previous rule. Normally users want to write rules with more restrictive conditions after less restrictive ones to avoid overwriting meta-data by mistake.

An example set of files with some properties is given in Table 6:

	DPR.CATG	**DPR.TYPE**	**DET.CHIPS**	**MJD-OBS**
File 1	CALIB	BIAS	1	53847.01567130
File 2	CALIB	BIAS	1	53847.11567130
File 3	CALIB	BIAS	2	53847.21567130
File 4	CALIB	BIAS	2	53847.31567130
File 5	CALIB	FLAT	1	53847.41567130
File 6	CALIB	FLAT	1	53847.51567130
File 7	SCIENCE	OBJECT	1	53847.61567130
File 8	SCIENCE	OBJECT,GALAXY	1	53847.71567130
File 9	SCIENCE	OBJ,GALAXY	1	53847.81567130

Table 6 Set of files

The result of applying the classification rules in Table 5 to the files in Table 6 is shown in Table 7:

	RAW.TYPE	**DO.CATG**
File 1	BIAS	BIAS_BLUE
File 2	BIAS	BIAS_BLUE
File 3	BIAS	BIAS_RED
File 4	BIAS	BIAS_RED
File 5	FLAT	FLAT_BLUE
File 6	FLAT	FLAT_RED
File 7	Undefined	SCIENCE
File 8	Undefined	SCIENCE
File 9	Undefined	SCIENCE

Table 7 : Result of classification

3.3.2 Organization/Grouping rules

Organization rules can be used to 1) organize a data set into homogeneous groups and 2) invoke the association rules [3.3.3] for each group. Organization rules are written as follows:

```
[minRet = value;]
select execute(action) from data-set where condition [group by keyword-list];
```

Value is an integer defining the minimum number of files the group shall contain.
Action is an identifier referring to an *action block* (see section 3.3.3) where the relevant association rules are defined.
Data-set is an identifier referring to the collection of files on which the rule should operate. Currently the only allowed value for *data-set* is **inputFiles**.
Condition is a Boolean expression as defined in section 3.3.1.
Keyword-list is a comma separated list of keywords, as in:

```
KEYW.ONE, KEYW.TWO, KEYW.THREE
```

3.3.2.1 Evaluation

Given a collection of files and a set of organization rules, each rule selects all files for which the condition evaluates to true. If a **group by** clause is present, the selected files are divided in sub-groups according to the value of the group-by keywords; i.e. each group contains files which satisfy the **where** clause and have the same values of all keywords defined in the **group by** clause.

Example:
Given the organization rule:

```
select execute(action) from inputFiles where DPR.CATG == "CALIB"
group by DPR.TYPE, DET.CHIPS
```

And the set of files in Table 6, the following groups are created:

Group	Files	DPR.CATG	DPR.TYPE	DET.CHIPS
Group 1	**File 1**	CALIB	BIAS	1
	File 2			
Group 2	**File 3**	CALIB	BIAS	2
	File 4			
Group 3	**File 5**	CALIB	FLAT	1
	File 6			

Table 8 Groups created by grouping rule

For each group defined by the grouping rule, the OCA interpreter performs the following steps:

1. Determine the **Exemplar (or Representative) Frame**[3], which is the "oldest" file in the group, i.e. the one with smallest MJD-OBS value
2. Evaluate the corresponding **action** rule, i.e. the one referenced by name in the **select execute(**action**)** clause. Association rules are described in next section [3.3.3].

3.3.3 Association/Action rules

Association rules are used to associate the following information to a group of files defined by a grouping rule:

1. **Files**. In a typical pipeline processing scenario, some reference files are needed to process a set of raw data. These files are selected from a *calibration database* using association rules which define the criteria to select these files.
2. **Recipe and recipe parameters**. The name and parameters of the pipeline recipe which should be used to process the input data.
3. **Products and products meta-data**. The name and properties of the products which will be generated by the execution of the pipeline recipe on this block of data. Also called **virtual products** or **future products** because these products don't exist at the time of data association.

An action block is opened by the reserved word **action** followed by an identifier (name) and an open brace "{", and is closed by the close brace "}".

```
action name
{
  File association rules
```

[3] The properties of the exemplar frame can be referenced in association rules using the prefix **inputFile**, see example on 3.3.3.1.

```
                    Recipe definition rule
                    Product definition rules
                }
```

An action block contains zero or more **file association rules**, exactly one **recipe definition rule** and zero or more **product definition rules**. The syntax of these rules is described in sections [3.3.3.1], [3.3.3.2] and [3.3.3.3].

3.3.3.1 File Association Rules

File association rules define the criteria to associate (reference) files to a set (group) of input files. The syntax of association rules is as follows:

```
                    [minRet = value;] [maxRet = value;]
                    select file as alias from data-source where condition;
```

Where the `minRet` and `maxRet` assignments are optional; the default value for both `minRet` and `maxRet` is 1.

Association rules are made of the following components:

- **Cardinality**. Defines the minimum (`minRet`) and maximum (`maxRet`) number of files to be returned by an association rule. If the rule selects less than `minRet` files, the calling application should raise an error. If more than `maxRet` files are selected, only the `maxRet` files which are **closest in time**[4] to the input file should be returned to the caller. By default an association rule should return exactly one file (`minRet=1; maxRet=1`). The default cardinality can be overridden by setting `minRet` and `maxRet` to user defined values, just before the select statement (see syntax below); this value applies only to the first select clause after the statement. It is not possible to select an unlimited number of files: this limitation can be overcome specifying a very large `maxRet`.
- **Alias**. The alias is used to attach a user defined label to the results returned by an association rule. This feature is mainly used for logging purposes to track the execution of association rules, please note that this is **not** what ends up as DO.CATG.
- **Data-source**. Associated files are selected among the set of files identified by this tag. Data-source is just a symbolic name which can refer to different physical implementations (database table, directory structure, etc.) depending on the context. Valid data sources are `rawFiles`, `inputFiles` and `calibFiles`: their meaning is application dependent.
- **Condition**. The condition is a Boolean expression as defined in section 3.3.1, with one important difference. In this context it is possible to express the condition in terms of properties both of the input files and of the associated files. When referring to input files[5], the prefix **inputFile.** must be used, as shown in the following example:

```
select file as MASTER_BIAS from calibFiles where PRO.CATG=="MASTER_BIAS_UVB"
and inputFile.DET.WIN1.BINX==DET.WIN1.BINX and inputFile.DET.WIN1.BINY==DET.WIN1.BINY
```

The condition `inputFile.X==X` is satisfied by all files which have the property X defined and with the same value as property X of the input file.

[4] The *closest in time* rule is evaluated using the **Modified MJD-OBS** value of the input files as a reference. The **Modified MJD-OBS** value is computed as follows:
$MJD\text{-}OBS_{modified} = MJD\text{-}OBS_{examplar} + (\Sigma_{input\_files}\ EXPTIME) / 2$

[5] The *input files* are actually represented by just one file, which is the **exemplar frame** defined in 3.3.2.1

3.3.3.2 Recipe Definition Rule

This rule is used to define the name and optionally the parameters of the pipeline recipe which should be used to process the data block consisting of input files plus associated files. The recipe definition rule can be written in the following format:

```
recipe name;
```

Or

```
recipe name { parameters };
```

Where *parameters* is a semicolon separated list of character strings, like in the following example:

```
recipe xsh_scired_slit_offset {
 "--rectify-bin-lambda=0.015"; "--rectify-bin-slit=0.16";
 "--extract-method=LOCALIZATION"; "--mergeord-method=0";
}
```

Note that it's up to the tool whether to use the recipe parameters or not, depending on its purpose.

3.3.3.3 Product Definition Rules

These rules are used to define the *properties* of the products which will be generated by a successful execution of the pipeline recipe indicated by the recipe definition rule [3.3.3.2]. At the time of data organization these products don't exist yet, therefore they are called **virtual products** or **future products**. The purpose of declaring in advance which products will be generated by a pipeline execution is to include them (at run-time) in the reference files database (*calibFiles*) even if the corresponding data file does not exist yet, in order to make virtual products available for subsequent (*second level*) associations. Not all of the OCA applications support the virtual products feature, in particular this feature is not needed by the CalSelector and in general by applications only interested in data packing and not in data reduction.
The syntax of the product definition rule is as follows:

```
product name { assignments };
```

Where *name* is just a label and *assignments* is a list of meta-keyword definitions which define some important properties of the product, as in the following examples:

```
product MASTER_BIAS { PRO.CATG = "MASTER_BIAS"; }
product MASTER_FLAT { PRO.CATG = "MASTER_FLAT"; }
```

At run-time, a product definition rule creates a new OCA file in memory which inherits all of the properties of the input (exemplar) frame that triggered this rule, and contains in addition the new properties defined by the rule (most notably PRO.CATG which is the ESO product category): this (virtual) file can be then used by other rules to generate new associations.

4. OCA APPLICATIONS

This chapter gives an overview of some ESO data organization tools based on the OCA language, please refer to the application-specific documentation for a more detailed description.
Some applications use dedicated meta-keywords, that might make the rules not compatible across different applications.
This is not a comprehensive description of the applications' behavior: for more information please refer to the specific user manual.

4.1 ABbuilder

The ABbuilder organizes input data into Association Blocks (an example AB is provided in chapter 8) using a set of instrument specific OCA rules. The ABbuilder performs the following operations:

1. Load instrument-specific rules. If a syntax error is detected during this phase, ABbuilder reports a message indicating where the syntax error occurred, and the application terminates.
2. Load and classify all *.fits, *.tfits and *.hdr files in the **raw** directory (including subdirectories).
3. Load and classify all *.fits, *.tfits and *.hdr files in the **calib** directory (including subdirectories).
4. If the **virtual** product directory is specified (option `--vcal-dir`), load all virtual products (header files) in this directory.
5. Organize the raw data into Association Blocks using the OCA organization and grouping rules.
6. Apply the association rules to add the following information to each association block:
    - Associated calibrations
    - Recipe
    - Virtual products
7. Save the association blocks in the AB directory
8. If the virtual product directory is specified (option `--vcal-dir`), save virtual products in this directory as header files.

Certain OCA meta-keywords have special meaning to the ABbuilder:

- **PACK.DIR**. The subdirectory of a file in a service mode data package. This meta-keyword is saved in the Association Block file and then used by the QC Data Packer to prepare the service mode package.
- **PRO.EXT**. The file extension of a virtual product. This meta-keyword is defined in the product definition rules, for example:
  ```
  product mbias { PRO.CATG="MASTER_BIAS"; PRO.EXT="0001.fits"; }
  ```

The ABBuilder has some conventions on alias names in product selection: if it contains MASSOC or RASSOC the selected files are listed in the AB respectively as MASSOC or RASSOC.

4.2 Data Organiser

The Data Organiser receives raw data files from one or more instruments and generates the appropriate Reduction Blocks (an example RB is provided in chapter 7) for scheduling. Whilst doing this for a given instrument, it refers to an instrument-specific rule file and a set of instrument-specific calibration files.

Certain OCA meta-keywords have special meaning to the Data Organiser:

- **DO.CATG.** The Data Organiser Category of a file. If the rules for an instrument dictate that a given file should be processed by the instrument's pipeline, this meta-keyword must be set by that instrument's classification rules. DO.CATG is a required component of the Reduction Block file format - if no value is set for it, the Data Organiser will continue to work, but subsequent stages of the pipeline are likely to malfunction. The original Data Organiser sets the DO.CATG of a template file

set to the DO.CATG of its final frame. For backward compatibility, OCA-based rules can duplicate this behavior by referring to the LF.DO.CATG meta-keyword (see next bullet).

- **LF.DO.CATG.** The Data Organiser Category of the last file in a template. This keyword is set by the Data Organiser immediately after the classification step of template rule evaluation. The value used is the DO.CATG of the last frame in the template; the last frame is considered to be the one with the latest MJD-OBS. Rule files must not set LF.DO.CATG themselves, because its value will be overwritten by the Data Organiser. The rules should only refer to this keyword during the organisation and association steps of rule evaluation.

The Data Organiser makes use of the following FITS keywords for all instruments:

- **MJD-OBS**. The exposure date of the frame in question. This is used to sort an instrument's frames into chronological order.
- **TPL.START**. The exposure date of the first frame in a VLT DFS template. Collation of a new template begins whenever this value changes.
- **TPL.NEXP**. The total number of exposures in a template. Basic error checking is performed to ensure that this value is sane, but it is mostly used in logging output.
- **TPL.EXPNO**. The exposure number of the current frame in a template. Basic error checking is performed to ensure that this value is sane, but it is mostly used in logging output.

Some instruments require special-case processing, which is controlled using instrument-specific FITS keywords:

- **DET.CHIP1.Y**. The CCD ID of a FORS1/FORS2 frame. FORS1 and FORS2 frames are generated separately for each CCD chip. This keyword is used to determine which template collator and rule evaluator tasks should process a given frame.
- **OCS.CON.QUAD**. The CCD ID of a VIMOS frame. VIMOS frames are generated separately for each CCD chip. This keyword is used to determine which template collator and rule evaluator tasks should process a given frame.

4.3 Archive Data Organizer

In the contest of the DataTransfer, the Archive Data Organizer classifies incoming data using a set of OCA rules and generates Transfer Requests containing the following archive-related meta-data:

- TRANSFER.CATG (e.g. “rawfile”).
- TRANSFER.METHOD (e.g. “network”)
- TRANSFER.PRIORITY (e.g. 40)
- COMPRESSION.METHOD (e.g. “unixcompress”)

These meta-keywords define how a file should be transferred from the Observatory to the main archive in Garching.

4.4 CalSelector

The CalSelector is an archive service used to associate calibration data (raw and reduced) to science data. The main difference between the CalSelector and other data organization tools like ABbuilder or DataOrganiser is that the CalSelector operates on the whole ESO archive, which imposes some constraints on how the OCA rules can be evaluated. Since the FITS keywords of archived files are kept a relational database (more specifically in *instrument specific tables*), the CalSelector applies the OCA rules directly on the database after transforming them to suitable SQL statements.

4.5 Gasgano

Gasgano (http://www.eso.org/sci/software/gasgano/) is a tool that help the users of the ESO archive to mange, organize and reduce astronomic data: since version 2.4 Gasgano uses the OCA rules to classify the FITS files the user provides. In this case only the classification part of the rules is relevant to the application, organization and association rules are simply ignored.

5. USAGE OF C PREPROCESSOR DIRECTIVES IN OCA RULES

Sometimes OCA rules can be a bit tedious to write, like in the following example:

```
select file as MASTER_BIAS_VIS from calibFiles where PRO.CATG=="MASTER_BIAS_VIS"
and inputFile.DET.WIN1.BINX==DET.WIN1.BINX and inputFile.DET.WIN1.BINY==DET.WIN1.BINY
and inputFile.DET.WIN1.NX==DET.WIN1.NX and inputFile.DET.WIN1.NY==DET.WIN1.NY
and inputFile.DET.READ.CLOCK==DET.READ.CLOCK and inputFile.DET.CHIP1.ID==DET.CHIP1.ID
and inputFile.SEQ.ARM==SEQ.ARM;
```

Also, the size of a complete set of OCA rules for a complex[6] VLT instrument can be significant (e.g. the XSHOOTER OCA rules are nearly 5000 lines).

The OCA language does not provide any built-in mechanism for organizing large sets of OCA rules into smaller and more manageable units, but fortunately there are other convenient ways for achieving this goal without having to expand the scope of the OCA project. We refer in particular to the usage of the C preprocessor **macro definition (**`#define`**)** and **source file inclusion** (`#include`**)** directives, as shown in the following example:

File macro.oca

```
# define _m(A)                        inputFile.A==A
# define MATCH1(A)                    _m(A)
# define MATCH2(A,B)                  _m(A) and _m(B)
# define MATCH3(A,B,C)                _m(A) and MATCH2(B,C)
# define MATCH4(A,B,C,D)              _m(A) and MATCH3(B,C,D)
# define MATCH5(A,B,C,D,E)            _m(A) and MATCH4(B,C,D,E)
# define MATCH6(A,B,C,D,E,F)          _m(A) and MATCH5(B,C,D,E,F)
# define MATCH7(A,B,C,D,E,F,G)        _m(A) and MATCH6(B,C,D,E,F,G)

# define BINX     DET.WIN1.BINX
# define BINY     DET.WIN1.BINY
# define NX       DET.WIN1.NX
# define NY       DET.WIN1.NY
# define CLOCK    DET.READ.CLOCK
# define CHIP     DET.CHIP1.ID
# define ARM      DET.SEQ.ARM
```

File association.oca

```
# include "macro.oca"
select file as MASTER_BIAS_VIS from calibFiles where PRO.CATG=="MASTER_BIAS_VIS"
and MATCH7(BINX,BINY,NX,NY,CLOCK,CHIP,ARM);
```

File all.oca

```
# include "classification.oca"      // not defined in this example
# include "organization.oca"        // not defined in this example
# include "association.oca"
```

The final OCA script is obtained simply by invoking the C preprocessor in order to expand the macro definitions and to resolve the file inclusions. The following command works with the GCC compiler:

```
gcc -x c -P –E all.oca
```

[6] In this context the complexity refers to the number of instrumental modes and setups.

Please note that the use of macros could lead to unexpected and undesired behavior if a macro makes use of one of the values defined in one of the preceding macros, e.g.:

```
#define FILT INS.FILT1.NAME
#define FILT2 INS.FILT2.NAME
```

Leads to `FILT2` being replaced by `INS.INS.FILT1.NAME2.NAME`, which is clearly not the desired behavior: such errors usually result in a syntax error in the rules, but they can also be quite tricky to spot.

For an exhaustive description of the C preprocessor and of the GCC compiler, please refer to [RD2] and [RD3].

6. APPENDIX A – OCA RULES EXAMPLE

```
// subset of rules for ISAAC
// classification
if DPR.CATG like "%CALIB%" and DPR.TECH like "%IMAGE%" and DPR.TECH like "%JITTER%" and DPR.TYPE like
"%STD%" then
{
  DO.CATG = "IM_ZPOINT";
}
if DPR.CATG like "%CALIB%" and DPR.TECH like "%IMAGE%" and DPR.TYPE like "%DARK%" then
{
  DO.CATG = "IM_DARK";
}
if PRO.CATG like "%DETLIN_A%" then
{
  DO.CATG = "DETLIN_A";
}
if PRO.CATG like "%DETLIN_B%" then
{
  DO.CATG = "DETLIN_B";
}
if PRO.CATG like "%DETLIN_C%" then
{
  DO.CATG = "DETLIN_C";
}
if PRO.CATG like "%MASTER_IMG_FLAT%" then
{
  DO.CATG = "MASTER_IMG_FLAT";
}
if PRO.CATG like "%STDSTARS_CATS%" then
{
  DO.CATG = "STDSTARS_CATS";
}

// organization
select execute(isaac_img_dark) from inputFiles where SIG.TEMPLATE == 1 and DO.CATG == "IM_DARK";
select execute(isaac_img_zpoint) from inputFiles where SIG.TEMPLATE == 1 and DO.CATG == "IM_ZPOINT";

// association
action isaac_img_dark
{
recipe isaac_img_dark;
}

action isaac_img_zpoint
{
minRet = 0; maxRet = 1;
select file as STDSTARS_CATS from calibFiles where DO.CATG == "STDSTARS_CATS";
minRet = 0; maxRet = 1;
select file as DETLIN_A from calibFiles where DO.CATG == "DETLIN_A"
   and inputFile.INSTRUME==INSTRUME and inputFile.DET.MODE.NAME==DET.MODE.NAME;
minRet = 0; maxRet = 1;
select file as DETLIN_B from calibFiles where DO.CATG == "DETLIN_B"
   and inputFile.INSTRUME==INSTRUME and inputFile.DET.MODE.NAME==DET.MODE.NAME;
minRet = 0; maxRet = 1;
select file as DETLIN_C from calibFiles where DO.CATG == "DETLIN_C"
   and inputFile.INSTRUME==INSTRUME and inputFile.DET.MODE.NAME==DET.MODE.NAME;
minRet = 0; maxRet = 1;
select file as MASTER_IMG_FLAT from calibFiles where DO.CATG == "MASTER_IMG_FLAT"
   and inputFile.INS.FILT1.ID==INS.FILT1.ID and inputFile.NAXIS1==NAXIS1
   and inputFile.NAXIS2==NAXIS2 and inputFile.INSTRUME==INSTRUME;
recipe isaac_img_zpoint;
}
```

7. APPENDIX B – REDUCTION BLOCK EXAMPLE

```
recipe: sinfo_rec_pupil

instrument: sinfo

/data_sinfo/lists/reduced_do/2006-05-07/r.SINFO.2006-05-07T21:55:08.427_tpl

{
/diskb/data_sinfo/SegRaw/sinfo/sinfo_rec_pupil_set1/SINFO.2006-05-07T21:55:08.427.fits PUPIL_LAMP
/diskb/data_sinfo/SegRaw/sinfo/sinfo_rec_pupil_set1/SINFO.2006-05-07T21:56:15.880.fits PUPIL_LAMP
}

{
/data_sinfo/calibDB/certif/ins/sinfo/cal/SI_PMPM_K_PUP.fits MASTER_BP_MAP
/data_sinfo/calibDB/certif/ins/sinfo/cal/SI_PMFL_K_PUP.fits MASTER_FLAT_LAMP
/data_sinfo/calibDB/certif/ins/sinfo/cal/FIRST_COLUMN.fits FIRST_COL
/data_sinfo/calibDB/certif/ins/sinfo/cal/SI_PWMP_K_PUP.fits WAVE_MAP
/data_sinfo/calibDB/certif/ins/sinfo/cal/SI_PSLD_K.fits SLITLETS_DISTANCE
/data_sinfo/calibDB/certif/ins/sinfo/cal/SI_PWSP_K_PUP.fits SLIT_POS
/data_sinfo/calibDB/certif/ins/sinfo/cal/SI_PDST_K.fits DISTORTION
}
```

8. APPENDIX C – ASSOCIATION BLOCK EXAMPLE

```
# general information
TOOL_VERSION   ABbuilder-1.1.5
CONFIG_VERSION config.createAB_2.4
INSTRUMENT     GIRAFFE
DATE           2007-03-18

DPR_CATG       CALIB
RAW_TYPE       NFLT
RECIPE         gimasterflat
DRS_TYPE       CON
PACK_DIR       FLAT
BATCH_ID       CALIB_2007-03-18

AB_NAME        GIRAF.2007-03-18T22:23:24.749_tpl.ab
AB_EVENT       TPL_A
COMPLETENESS   COMPLETE
PROCESS_STATUS SUCCESSFUL

OBS_PROG_ID    60.A-9022(B)
OBS_MODE       NONE
OBS_ID         200164184

MJD-OBS        54177.93292534
MJD-OBS_MOD    54177.93326910

PROD_ROOT_NAME r.GIRAF.2007-03-18T22:23:24.749_tpl

LOG_NAME       NONE

# pipeline product path
PROD_PATH      ${DFS_PRODUCT}/NFLT/2007-03-18

# raw match key
RAW_MATCH_KEY  INS.SLIT.NAME=Argus
RAW_MATCH_KEY  INS.EXP.MODE=H379.0

# input file(s)
RAWFILE        ${DFO_RAW_DIR}/2007-03-18/GIRAF.2007-03-18T22:23:24.749.fits       NASMYTH_FLAT

# associated raw file(s) (only for SCIENCE; taken from same night only!)
RASSOC         NONE

# product file(s)
PRODUCTS       NONE    NONE

# associated mcalib file(s), used for processing
MCALIB         REAL    ${DFO_CAL_DIR}/2007-03-17/GI_MBIA_070317A_1x1.fits MASTER_BIAS GIRAF.2007-03-
18T13:46:46.237_tpl.ab -0.359
MCALIB         REAL    ${DFO_CAL_DIR}/gen/grating_HR316.tfits GRATING_DATA GEN -9999
MCALIB         REAL    ${DFO_CAL_DIR}/gen/slit_geometry_Argus_H379.0_o15.tfits SLIT_GEOMETRY_SETUP GEN
-9999

# associated mcalib file(s), used for packing
MASSOC         NONE       0

# parameters for processing
PARAM          --bsremove-method=MASTER
PARAM          --sloc-noise=5.

# defined waitfors (for CONDOR)
WAITFOR        NONE


# further associated information
FURTHER_PS     NONE
FURTHER_GIF    NONE

# status of AB
TEXEC          783
```

```
AB_STATUS      - created by 'createAB'              on Tue Mar 27 15:44:28 CEST 2007 by flames1 on
dfo03
AB_STATUS      - processed by 'processAB'    on Tue Mar 27 15:59:00 CEST 2007 by flames1 on dfo03

# ========== RB section starts here ==========
RB_CONTENT     recipe: gimasterflat
RB_CONTENT
RB_CONTENT     instrument: GIRAFFE
RB_CONTENT
RB_CONTENT     ${DFS_PRODUCT}/NFLT/2007-03-18/r.GIRAF.2007-03-18T22:23:24.749_tpl
RB_CONTENT
RB_CONTENT     {
RB_CONTENT     ${DFO_RAW_DIR}/2007-03-18/GIRAF.2007-03-18T22:23:24.749.fits NASMYTH_FLAT
RB_CONTENT     }
RB_CONTENT
RB_CONTENT     {
RB_CONTENT     ${DFO_CAL_DIR}/2007-03-17/GI_MBIA_070317A_1x1.fits MASTER_BIAS
RB_CONTENT     ${DFO_CAL_DIR}/gen/grating_HR316.tfits GRATING_DATA
RB_CONTENT     ${DFO_CAL_DIR}/gen/slit_geometry_Argus_H379.0_o15.tfits SLIT_GEOMETRY_SETUP
RB_CONTENT     }
RB_CONTENT
RB_CONTENT     --bsremove-method=MASTER
RB_CONTENT     --sloc-noise=5.
RB_CONTENT

# ========== SOF section starts here ==========
SOF_CONTENT    ${DFO_RAW_DIR}/2007-03-18/GIRAF.2007-03-18T22:23:24.749.fits NASMYTH_FLAT RAW
SOF_CONTENT    ${DFO_CAL_DIR}/2007-03-17/GI_MBIA_070317A_1x1.fits MASTER_BIAS CALIB
SOF_CONTENT    ${DFO_CAL_DIR}/gen/grating_HR316.tfits GRATING_DATA CALIB
SOF_CONTENT    ${DFO_CAL_DIR}/gen/slit_geometry_Argus_H379.0_o15.tfits SLIT_GEOMETRY_SETUP CALIB
```