WorkflowMaker user guide

For WorkflowMaker version **1.0.4** and later
Windows 10 & 11, Ubuntu-based Linux distributions
64-bit only

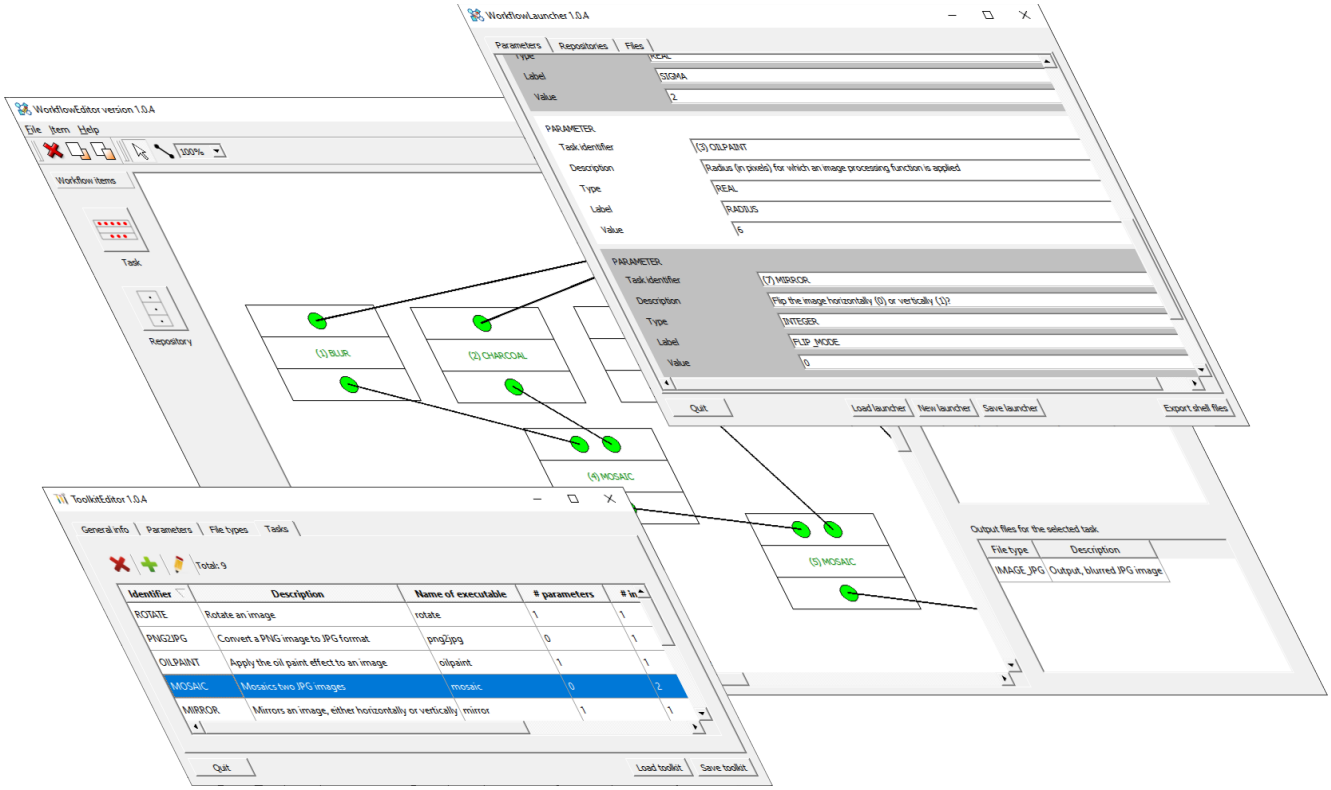

Table of contents

1 Introduction....4
2 WorkflowMaker....5
2.1 The console application. Restrictions....5
2.2 Defining tasks and toolkits....7
2.2.1 The list of file types....7
2.2.2 The list of keyboard parameters....8
2.2.3 The applications (tasks)....9
2.2.4 The toolkit....9
2.3 Design workflows visually....9
2.4 Provide actual data and run the workflow....10
3 The tools....12
3.1 The example....12
3.2 Running the tools....15
3.3 ToolkitEditor....16
3.3.1 The general information....17
3.3.2 The list of keyboard parameters....18
3.3.3 The list of file types....20
3.3.4 The list of tasks....21
3.3.5 Saving the toolkit....25
3.3.6 Loading an existing toolkit....25
3.3.7 Caveats: please *do* read this section very carefully....25
3.4 WorkflowEditor....27
3.4.1 The example workflow....27
3.4.2 The interface of WorkflowEditor....30
3.4.3 Creating, opening or saving workflows....33
3.4.4 Inserting, moving and deleting repositories and tasks....35
3.4.5 Inserting and deleting connections....36
3.4.6 Caveats: please *do* read this section very carefully....39
3.5 WorkflowLauncher....40
3.5.1 The interface....41
3.5.2 Creating a new launcher file....41
3.5.3 Opening an existing launcher file....44
3.5.4 Values for the keyboard parameters....44
3.5.5 Paths for the repositories....46
3.5.6 Give names to files....47
3.5.7 Saving the launcher file....48
3.5.8 Generating the script (shell) files....49
3.5.9 Running the workflow....51
3.5.10 Reusing launcher files....52
3.5.11 Running the example workflow. Results....52

List of figures

Figure 1: What do the tools included in the example toolkit do?....14
Figure 2: Running the WorkflowMaker tools on a Windows computer....16
Figure 3: ToolkitEditor: the Graphical User Interface (GUI)....17
Figure 4: ToolkitEditor: the "General info" tab....18
Figure 5: ToolkitEditor: the "Parameters" tab....19
Figure 6: ToolkitEditor: adding a new parameter....19
Figure 7: ToolkitEditor: warning about deleting a parameter already in use....20
Figure 8: ToolkitEditor: the "File types" tab....20
Figure 9: ToolkitEditor: adding a new file type....21
Figure 10: ToolkitEditor: warning about deleting a file type already in use....22
Figure 11: ToolkitEditor: the "Tasks" tab....22
Figure 12: ToolkitEditor: adding or editing a task....23
Figure 13: ToolkitEditor: managing the parameters in a task....24
Figure 14: ToolkitEditor: managing the input / output files in a task....24
Figure 15: WorkflowEditor: the visual representation of the example workflow....29
Figure 16: WorkflowEditor: the Graphical User Interface (GUI)....30
Figure 17: WorkflowEditor: the menus and the toolbar....31
Figure 18: WorkflowEditor: the item selector....31
Figure 19: WorkflowEditor: representation of a repository and a task....32
Figure 20: WorkflowEditor: the task selector....32
Figure 21: WorkflowEditor: the input / output files component....33
Figure 22: WorkflowEditor: metadata for a new workflow....34
Figure 23: WorkflowEditor: asking for the toolkit on which a workflow relies....34
Figure 24: WorkflowEditor: warnings on incomplete workflow when trying to save it....35
Figure 25: WorkflowEditor: coloring of slots when initiating a connection....38
Figure 26: WorkflowLauncher: the Graphical User Interface (GUI)....42
Figure 27: WorkflowLauncher: metadata for a new launcher file....42
Figure 28: WorkflowEditor: finding the toolkit on which the launcher workflow relies....43
Figure 29: WorkflowLauncher: the "Parameters" tab just after creating a new launcher file....43
Figure 30: WorkflowLauncher: asking for the underlying workflow for an existing launcher....44
Figure 31: WorkflowLauncher: typing the keyboard parameters....45
Figure 32: WorkflowLauncher: assigning actual path to repositories....46
Figure 33: WorkflowLauncher: giving names to input and output files....48
Figure 34: WorkflowLauncher: saving the launcher before creating the scripts....49
Figure 35: WorkflowLauncher: options to generate the script (shell) files....49
Figure 36: WorkflowLauncher: example of an automatically generated options file....50
Figure 37: WorkflowLauncher: the Windows shell file for the example in Figure 15....51
Figure 38: WorkflowLauncher: the unique input image for the example workflow....53
Figure 39: WorkflowLauncher: the result of running the example workflow....53

1 Introduction

This is the user guide for the WorkflowMaker software suite.

Briefly, WorkflowMaker is a tool that allows its users to visually design workflows that integrate console applications developed by themselves.

Section 2 of this document explains the steps to follow to adapt user console applications so that they are compatible with WorkflowMaker. It also explains what needs to be done to design (and execute) workflows based on these applications. This section is rather formal, as it explains the model on which WorkflowMaker relies to carry out its task. The concepts of toolkit, workflow, and launcher are defined, including, for each of them, a description of their components.

This formal description is complemented in Section 3 with a practical explanation of the tools incorporated in this software suite, namely ToolkitEditor, WorkflowEditor, and WorkflowLauncher. To make this description more understandable, an example use case is presented, which serves as the basis for all the explanations related to these tools.

It is highly recommended that a user who wishes to master WorkflowMaker to design their own workflows carefully read section 2 of this document, as it lays the necessary conceptual foundation to understand the functioning of this software suite. From there, section 3 shows, with numerous examples, how to proceed to design toolkits, workflows based on them, and finally, launchers that allow the execution of those workflows with user datasets.

2 WorkflowMaker

WorkflowMaker is a set of tools targeted at helping developers to integrate their **console** applications into more complex workflows in a visual way.

To do it, users must follow the steps below:

- Design and implement their console applications respecting a very reduced set of restrictions (section 2.1).
- Define, in a formal way, the inputs and outputs of the set of applications that will be included in a so-called *toolkit* (section 2.2).
- Design a workflow visually (section 2.3).
- For that workflow, provide the values of the parameters and names of the input / output files required to execute it and execute it.

2.1 The console application. Restrictions

WorkflowMaker is targeted at creating and executing workflows made exclusively of console applications – also known as command line applications. Applications with a Graphical User Interface (GUI) are not supported.

Console applications, typically, accept two kind of inputs: *keyboard* parameters – that is, data input by the user when prompted to provide with some value(s) – and files. Regarding to outputs, these may be either some information printed on the screen or, once more, files.

For a console application to be integrable in a WorkflowMaker workflow, it must however, respect the following restrictions:

- All inputs, including the so-called keyboard parameters, must be files.
- All outputs must be files too.
- The keyboard parameters as well as the names of the input and output files must be defined in a single file, the *options file*.
- An options file must be written respecting a very simple format, consisting of lines made of pairs of labels and values (as, for instance, “RADIUS = 3.5”, without the quotes.).
- The application must be able to read this options file to retrieve the values of the keyboard parameters as well as the names of the input and output files.
- The application must be built as an executable file, that is, the operating system must be able to execute it without the intervention of any kind of middleware, such as Python interpreters or Java virtual machines.
- When the console application is run, it must accept a single command line parameter, defining the name of the options file mentioned above.
- They must return a status code stating whether it ended successfully (return code 0) or not (any other value).

The restrictions above are targeted at making possible a high degree of automation when a workflow is executed. This is the reason why the typical keyboard inputs must be stored in a file or that even the outputs printed to the screen must also be saved in this way. Users will not be bothered with questions or information once a workflow is started; all outputs may be checked at the end of the execution.

Another reason to set these restrictions is the need to *normalize* the way applications are integrated. Standardizing the way they obtain and produce their data (files) and the way they are run (command line with a single parameter) makes integrating any application possible, since the mechanism to do it is clearly defined.

The restriction concerning the use of executable files (instead of, for instance, Java bytecode or Python sources) is a limitation of the current version of WorkflowMaker. It is foreseen that future version of this software suite will remove this restriction.

For instance, an image processing console application applying the charcoal effect to an input image would need the name of the input and output files as well as the radius of the charcoal operator to be able to perform its task. Therefore, assuming that such application is called “charcoal”, it should be possible to invoke it like this:

```
charcoal name_of_the_options_file.op
```

that is, called with a single command line parameter, the name of the options file. The following is an example of how such options file would look like:

```
INPUT_FILENAME  = C:/data/PNG2JPG_10_0.jpg
OUTPUT_FILENAME = C:/data/OILPAINT_3_0.jpg
RADIUS          = 6
```

Here, the names of both the input and output files are defined in the first two lines (labels “INPUT_FILENAME” and “OUTPUT_FILENAME”). Moreover, the value of the radius parameter (label “RADIUS”) is also given.

Thus, the developer must be prepare its code so it is able to read this kind of label + equal sign + value files.

Note that developers are free to choose the labels themselves. For instance, the text “NAME_OF_INPUT_FILE” could have been used instead of “INPUT_FILENAME”. It is the task of the developers to look for the right labels, since they are the ones who decide which ones to use.

It is worth to say that these option files will be automatically created by WorkflowLauncher (see sections 2.4 and 3.5). Developers are responsible only of reading the right set of labels and values from the option file that will be passed as a parameter when calling some application.

Then, assuming that the application “charcoal” is an executable file that is invoked as shown above, that it is able to read the example options file shown, that the rest of its inputs are only files, that it will produce only files as output and that it will return a status code to signal whether it finished successfully or not, then this application may be integrated using WorkflowMaker.

Note that, for the sake of terseness, in the sections to follow the words “console application”, “application” or “task” will be used indistinctly to refer to console applications.

2.2 Defining tasks and toolkits

Once that it is guaranteed that all the applications that are going to be included in a toolkit comply with the restrictions described in section 2.1, it is possible to characterize these in a formal way. Doing this implies following a series of steps. Sections 2.2.1 to 2.2.4 describe these.

Note that these sections describe the rationale behind the characterization of the tasks. The idea is let users understand the reason why some things are requested by the tool (ToolkitEditor, section 3.3). The reader should not worry, therefore, about the details concerning how to provide this information. ToolkitEditor will take care of this.

2.2.1 The list of file types

The set of applications that will be included in a toolkit will read and produce different kinds of files. Examples of these could be images, GPS positions, sensor characterization data, datum definitions, or camera calibration checkerboards. The previous list is just an example and, obviously, incomplete; moreover, different toolkits will require different file types.

The important thing here is that a console application will deal with a specific set of file types to do its work. For instance, the example application “charcoal” briefly sketched in section 2.1 above, requires its input file to be an image, not an audio file nor a spreadsheet. Furthermore, its output is also a file of type image.

It is very important to define what are the types of files that an application accepts and produces for each of its inputs and outputs. Making this information explicit will make possible, when designing a workflow visually, checking whether the output of some task is compatible with the input of some other. That is, it will be possible to reject that an output audio file be used as the input to some task that needs a file with a list of coordinates to work.

The first task to define a toolkit is identify the complete list of file types. To describe a single file type is enough to provide:

- A unique identifier (such as “JPG_IMAGE” or “COORDINATE_LIST”).
- A textual description stating the generic purpose of such file type (for instance, “Image in JPG format” or “List of X, Y and Z coordinates in plain text format”).
- A default extension for the file type (such as “.jpg” or “.txt”).

The list of file types will consist of a set of single file type descriptions as the one given above. When defining some task (section 2.2.3), the file type unique identifiers will be used to state what are the types of the input or output files such task reads or produces.

2.2.2 The list of keyboard parameters

When building a toolkit, it will contain applications that, usually, will perform tasks within a specific field of knowledge. For instance, some toolkits could work in the realm of image processing, bundle network adjustment or ground monitoring.

Therefore, the applications in the toolkit will deal with very similar concepts. This, usually, has direct implications on the keyboard parameters that these applications use.

For instance, in the realm of image processing, it is usual to find parameters defining the width or height of an image, the number of channels it includes or the radius in pixels of some operator. It is reasonable to expect that these parameters will be used by several applications in the toolkit.

The idea of the list of parameters is to avoid repetition as much as possible. That is, if three tasks in the toolkit use some parameter named “NUMBER_OF_CHANNELS”, whose type is “integer” and whose description is “Number of channels in the image”, it will be redefined, three times, one per task using it.

Avoiding repetition means, therefore, identifying the unique list of parameters used by the whole set of applications in the toolkit, and then define these only once, so when defining a task later, it will be possible to use a reference to the parameter instead of having to redefine it again.

The fact that different tasks share the definition of a parameter does not imply that, when executing a workflow based on these tasks, the value assigned to that parameter has to be the same for all of them. This reuse only has effects at the moment the task is formally defined; at runtime, a different value can be assigned, if necessary, to that parameter for each of the tasks that require it.

Thus, a single keyboard parameter is defined by means of:

- A **unique** label. This label will be used later on to create the option files described in section 2.1. Developers are free to choose these labels.
- A type (such us floating point, integer, string of characters…).
- A textual description.

The list of keyboard parameters is, therefore, a list including a set of individual parameter descriptions as the one above. Note, as stated above, that no repeated labels may exist in the list.

When defining a task (section 2.2.3), it will include as many parameter *labels* as necessary to build the

list of keyboard parameters it needs to run.

2.2.3 The applications (tasks)

Basically, defining a task consist of describing what are its keyboards parameters and input and output files.

More specifically, this the information needed to fully define an application:

1. The name of the executable file implementing the application.
2. A description of its purpose, so users willing to use it understand what it is made for.
3. The list of labels stating what are its keyboard parameters.
4. A list of input files.
5. A list of output files.

Both the list of input and output files gather the description of a series of individual files. To describe *each* of these single files, it is necessary to include:

- A **unique** *label* to identify the file. Again, developers are free to choose these labels at will.
- A file type identifier, stating what is the type of this file.
- A textual description stating the role of this file in the task.

Note how the use of the “label” and the “file type identifier” makes possible for a task to use, if necessary, several files of the same type. The label is used not only to tell apart this file from the others that might be intervening in the task but also to provide a mechanism to assign a file name when creating the options file described in section 2.1. The file type will be used to state what kind of file it is (An image? A list of coordinates?).

2.2.4 The toolkit

The definition of the toolkit is just the *addition* of all the elements already described plus some general information to describe the toolkit as a whole:

- Identifier and description.
- List of file types.
- List of keyboard parameters
- List of tasks.

To create a toolkit, use the ToolkitEditor. Section 3.3 describes this tool in detail.

2.3 Design workflows visually

A workflow, in the context of WorkflowMaker, is a set repositories, tasks and connections between them.

A repository is a place where files reside. These may be input files and thus be used to feed some tasks or, on the contrary, output ones, storing the results of some other tasks.

Tasks, the console applications, process information, taking some files as inputs (which include the values of the keyboard parameters) to produce some files as outputs.

And the connections, the key part of the design of a workflow, state how information flows between repositories and tasks, tasks and tasks, and tasks a repositories.

In the case of repository to task connections, a file from a repository is used as one of the input files of some task. On the contrary, when a connection goes from a task to a repository, it means that an output file from the said task is stored in the aforementioned repository. The final case, task to task connections indicate how some output file in the first task is used to provide some input file for the second one.

In short, *a WorkflowMaker workflow is a directed graph*: nodes may be repositories or tasks; the connections describe the flow of information.

A workflow is, however, just like a template. It is true that it describes in detail what are the tasks and repositories involved as well as the connections between these components, but no assumptions are made about the actual locations of repositories or the names of the input or output files. That is: a workflow is independent of the data set to process. In later steps (section 2.4) it will be described how to provide actual data (values for keyboard parameters, paths to repositories and names for the input and output files). That is, the same workflow may be used to process many different data sets.

A workflow relies on some toolkit. This means that the tasks that may be inserted in a workflow are only those that has been characterized by the toolkit. Using a different toolkit it is then possible to prepare workflows for very different fields of application.

With WorkflowEditor (section 3.4) it is possible to design workflows visually, inserting repositories and tasks and making the necessary connections to indicate how data flows. Figure 15 on page 29 depicts an example workflow designed with WorkflowEditor.

2.4 Provide actual data and run the workflow

WorkflowMaker workflows are nothing more than templates that explain how a specific set of data should be processed.

The true usefulness of the workflow emerges when it is executed. In other words, when it is put into production, processing as many data sets as necessary.

To execute a workflow, it is necessary to provide the information that it does not have. This includes the location of the different existing repositories, the values of the keyboard parameters for the

different tasks, as well as the names of the input and output files. It is important to emphasize that *it is not necessary to indicate the names of the intermediate files*, that is, those that are the output of one task and are used as the input of another, without being stored in a repository. Such names are generated automatically, thus reducing the volume of information that users have to provide to execute the workflow.

At this point, all the necessary information is available to execute the workflow, that is, the procedure to follow on one hand, and the data set to process on the other. In this way, it is possible to generate the option files (see section 2.1) that each task needs to work correctly, as well as a command (shell or batch) file that, step by step, executes the involved tasks in the required order, feeding those tasks with the mentioned option files.

The entire process of data collection and generation of option and command files is carried out through the WorkflowLauncher application (see section 3.1). With this application, it is also possible to save the collected data in a specific kind of file (launcher file) in order to reuse it if necessary, changing the necessary values and thus comfortably regenerate the options and command files to process different data sets.

3 The tools

The next sections will describe in detail how the three applications making the WorkflowMaker suite work. First, however, a simple example toolkit including several tools will be described. In fact, it is included as a sample when installing WorkflowMaker (only in Windows platforms). The use of a specific, tangible example will make it easier to explain how the different WorkflowMaker tools work. Moreover, the example selected, very simple image processing, includes applications that are very easy to understand, so no extra complexity is introduced into the explanation. This allows the reader to concentrate on understanding how to use WorkflowMaker.

3.1 The example

In this section a very simple toolkit consisting of nine console applications (tasks, applications) will be briefly explained. The field of application of the toolkit is that of image processing. All the tasks included in it apply some type of effect to an input image (such as blurring it) and save the result in an output image. Additionally there are two extra tools used to convert between two image formats.

Table 1 on page 13 briefly summarizes the tools included in the toolkit. In the case of keyboard parameters, the **boldfaced** text stands for the identifier of such parameters; for both input and output files, the **boldfaced** text corresponds to the labels that will be used in the option files, while the text in red stands for the identifiers of the file type these belong to.

Identifier	Description	Parameters	Input files	Output files
BLUR	Blurs an image.	**SIGMA** – Standard deviation for the blur algorithm. **RADIUS** – Radius (in pixels) for the blur algorithm.	**INPUT_FILENAME** (IMAGE_JPG) – The input image to process.	**OUTPUT_FILENAME** (IMAGE_JPG) – The blurred image.
CHARCOAL	Apply the charcoal effect to an image.	**SIGMA** – Standard deviation for the charcoal algorithm. **RADIUS** – Radius (in pixels) for the charcoal algorithm.	**INPUT_FILENAME** (IMAGE_JPG) – The input image to process.	**OUTPUT_FILENAME** (IMAGE_JPG) – The output image, charcoal effect applied.
GRAYSCALE	Converts a color image to gray scale.	None.	**INPUT_FILENAME** (IMAGE_JPG) – The input color image.	**OUTPUT_FILENAME** (IMAGE_JPG) – The output gray scale image.
JPG2PNG	Convert an image in JPG format to PNG format.	None.	**INPUT_FILENAME** (IMAGE_JPG) – The input image to convert.	**OUTPUT_FILENAME** (IMAGE_PNG) – The output image, converted to PNG.
MIRROR	Mirrors an image, either horizontally or vertically.	**FLIP_MODE** – To select whether to flip the image horizontally or vertically.	**INPUT_FILENAME** (IMAGE_JPG) – The input image to mirror.	**OUTPUT_FILENAME** (IMAGE_JPG) – The output, mirrored image.
MOSAIC	Mosaics two images	None.	**LEFT_IMAGE_FILENAME** (IMAGE_JPG) – The input image that will be located in the left side of the output mosaic. **RIGHT_IMAGE_FILENAME** (IMAGE_JPG) – The input image that will be located in the right side of the output mosaic.	**MOSAIC_FILENAME** (IMAGE_JPG) – The output image, a mosaic of the two input ones.
OILPAINT	Apply the oil paint effect to an image.	**RADIUS** – Radius (in pixels) for the oil painting algorithm.	**INPUT_FILENAME** (IMAGE_JPG) – The input image to process.	**OUTPUT_FILENAME** (IMAGE_JPG) – The output image with the oil paint effect applied.
PNG2JPG	Convert an image in PNG format to JPG format.	None.	**INPUT_FILENAME** (IMAGE_PNG) – The input PNG image to convert.	**OUTPUT_FILENAME** (IMAGE_JPG) – The output image in JPG format.
ROTATE	Rotate an image	**DEGREES** – Amount in degrees to rotate.	**INPUT_FILENAME** (IMAGE_JPG) – The input image to rotate.	**OUTPUT_FILENAME** (IMAGE_JPG) – The output, rotated image.

Table 1: Characterization of the tasks in the example image processing toolkit.

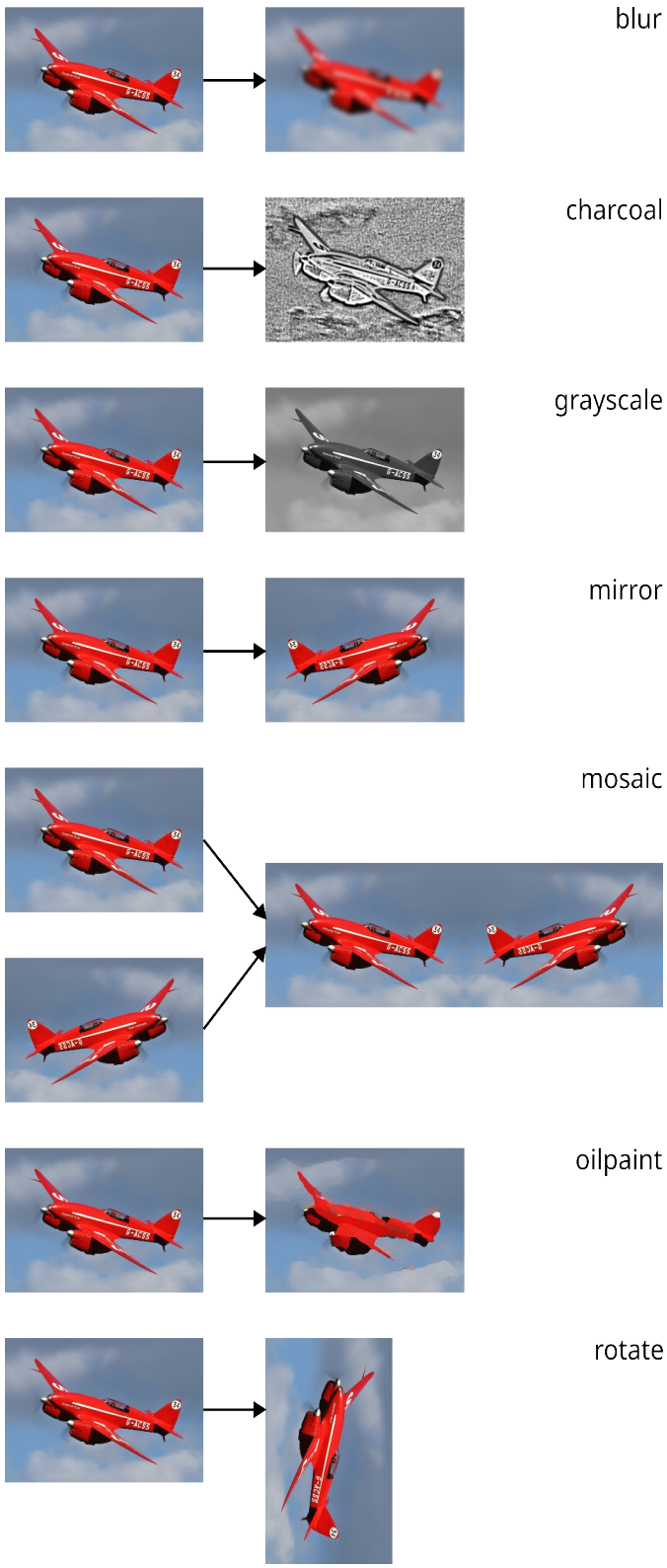

Figure 1: What do the tools included in the example toolkit do?

Figure 1 above graphically illustrates what the different tools included in the example toolkit do. The two converters (JPG2PNG and PNG2JPG) have been excluded from this figure as the type of work they perform is obvious.

With the information in Table 1 is possible to start defining the toolkit.

According to section 2.2 the first step is to define the list of file types. These are described in Table 2.

Identifier	Description	Default extension
IMG_JPG	An image in .jpg format	.jpg
IMG_PNG	An image in .png format	.png

Table 2: The list of file types in the example toolkit.

The list of unique parameters, the second thing to prepare, is shown in Table 3.

Identifier	Description	Data type
DEGREES	Amount in degrees to rotate	Floating point
FLIP_MODE	Flip the image horizontally (0) or vertically (1)?	Integer
RADIUS	The radius (in pixels) that some image processing functions require to carry out their work.	Floating point
SIGMA	The standard deviation that some image processing functions require to carry out their work.	Floating point

Table 3: The list of unique parameters in the example toolkit.

The list of tasks is already described in Table 1. Note the use of the identifiers in tables 2 and 3 when referring to file types and keyboard parameters.

Section 3.3 will explain how to define the example toolkit just presented. In section 3.4 a workflow combining some of the tools it includes will be visually designed. Finally, section 3.5 will show how to run such workflow, providing the necessary data to generate the command and options files to do it.

3.2 Running the tools

To start any of the tools in a **Windows** computer, click on the Windows start menu. Look for the “W” section in the emerging menu and the locate the “WorkflowMaker” entry. Unfold it and the select the tool of your choice. See Figure 2

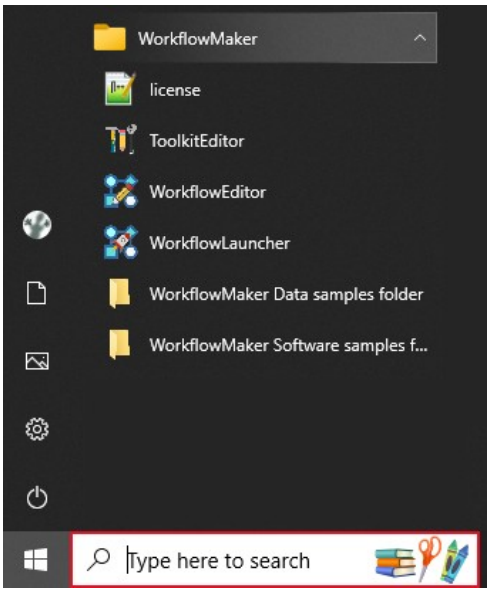

Figure 2: Running the WorkflowMaker tools on a Windows computer.

Alternatively, it is possible to start typing the name of these tools in the search box (`toolkiteditor`, `workfloweditor`, `workflowlauncher`) until a shortcut to the selected tool shows up in the start menu. Clicking on the said shortcut will start the desired application.

To run these tools on a **Linux** computer, proceed as follows:

- Open a command line window.
- Type the complete name of the desired tool (`toolkiteditor`, `workfloweditor`, `workflowlauncher`).
- Type `<return>`.

The selected tool will start after that.

3.3 ToolkitEditor

The ToolkitEditor is the tool that lets users formalize the description of the several tools making a toolkit. Such definition comprises four steps, to provide:

1. some general information (identifier and description) to identify the toolkit,
2. the list of file types,
3. the list of unique keyboard parameters and
4. the list of tasks.

Start the ToolkitEditor as described in section 3.2. Figure 3 depicts the Graphical User Interface (GUI) of this tool.

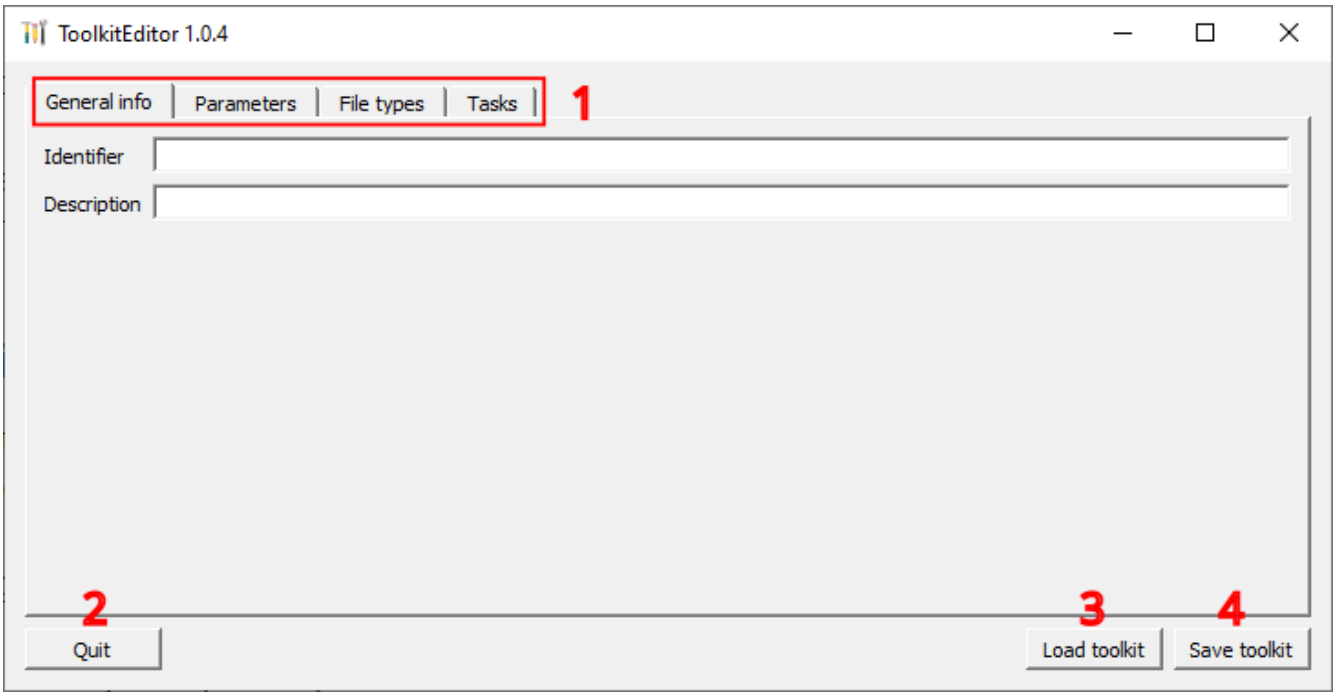

Figure 3: ToolkitEditor: the Graphical User Interface (GUI).

In this figure:

1. Here the four tabs to select between the different data items to edit (namely, the general information, keyboard parameters, file types and tasks) are shown. Clicking on any of these tabs will make the GUI change to show the requested data section.
2. Clicking on this button will kill the application. Make sure that your work has been properly saved before leaving ToolkitEditor.
3. Loads an existing toolkit file for further editing.
4. Saves the current toolkit.

3.3.1 The general information

To provide the general information,

- Select the General info tab.
- Type a toolkit identifier. Identifiers should be **unique** and, in some way, indicate the purpose of the toolkit. For example, a toolkit identifier like "ID0023" does not convey any information about it, whereas using something like "GEOPROCESSING_RADAR_TK" allows for a more intuitive identification. Identifiers may consist of letters, numbers and special characters. White space characters are not recommended.
- Type a general description stating the purpose of the toolkit – thus complementing the identifier.

Figure 4 shows the general information tab for the example toolkit.

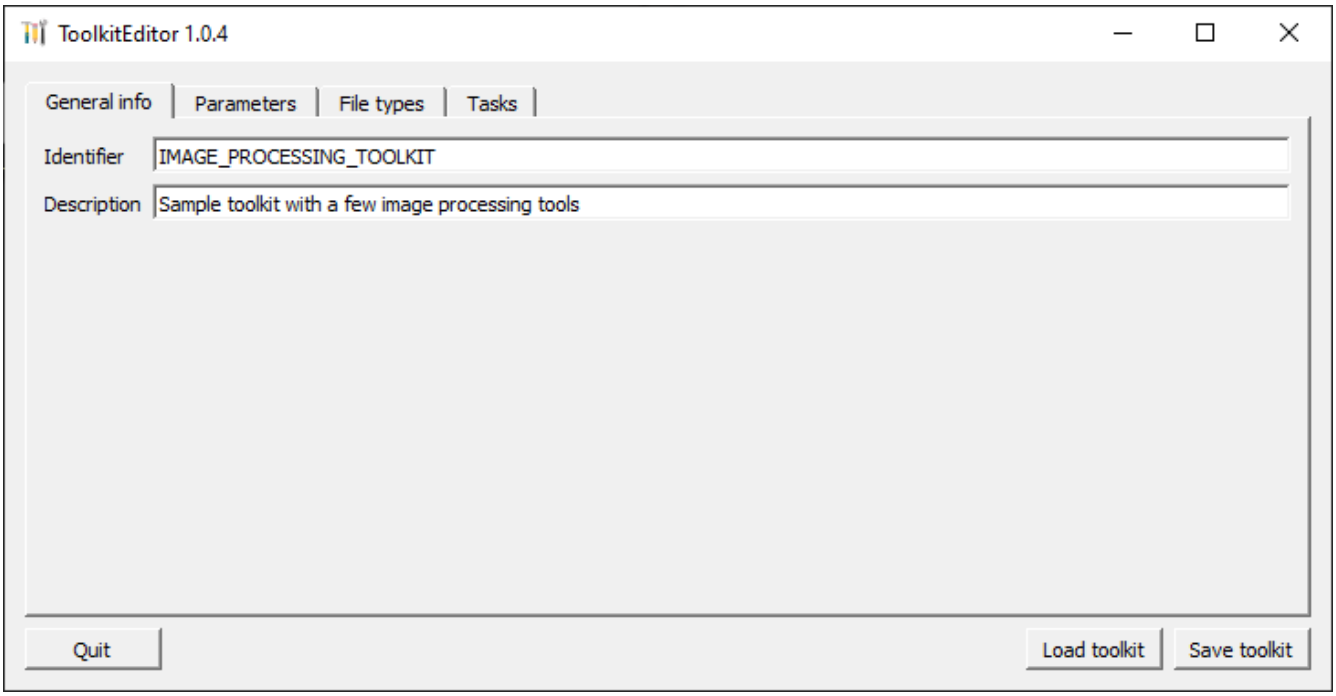

Figure 4: ToolkitEditor: the "General info" tab.

3.3.2 The list of keyboard parameters

By selecting the "Parameters" tab (see (1) in Figure 3), the interface for adding, modifying, and deleting keyboard parameters is displayed. Such interface is shown in Figure 5.

In this figure:

1. Button to remove an existing parameter.
2. Button to add a new parameter.
3. Button to edit an existing parameter.
4. List of parameters.

Note that the remove (1) and edit (3) buttons are only enabled when a parameter has been selected in the list of parameters (4) as shown in Figure 5.

To add a new parameter, just click on the add (2) button. The window in Figure 6 will appear. There, it is possible to type the identifier (which will become the options file label) for the parameter, as well as state its type (such as floating point, integer or string of characters) and provide the description explaining its purpose. Note that the identifier must be **unique**. By clicking on the “Save” button, the parameter will be added to the list of parameters.

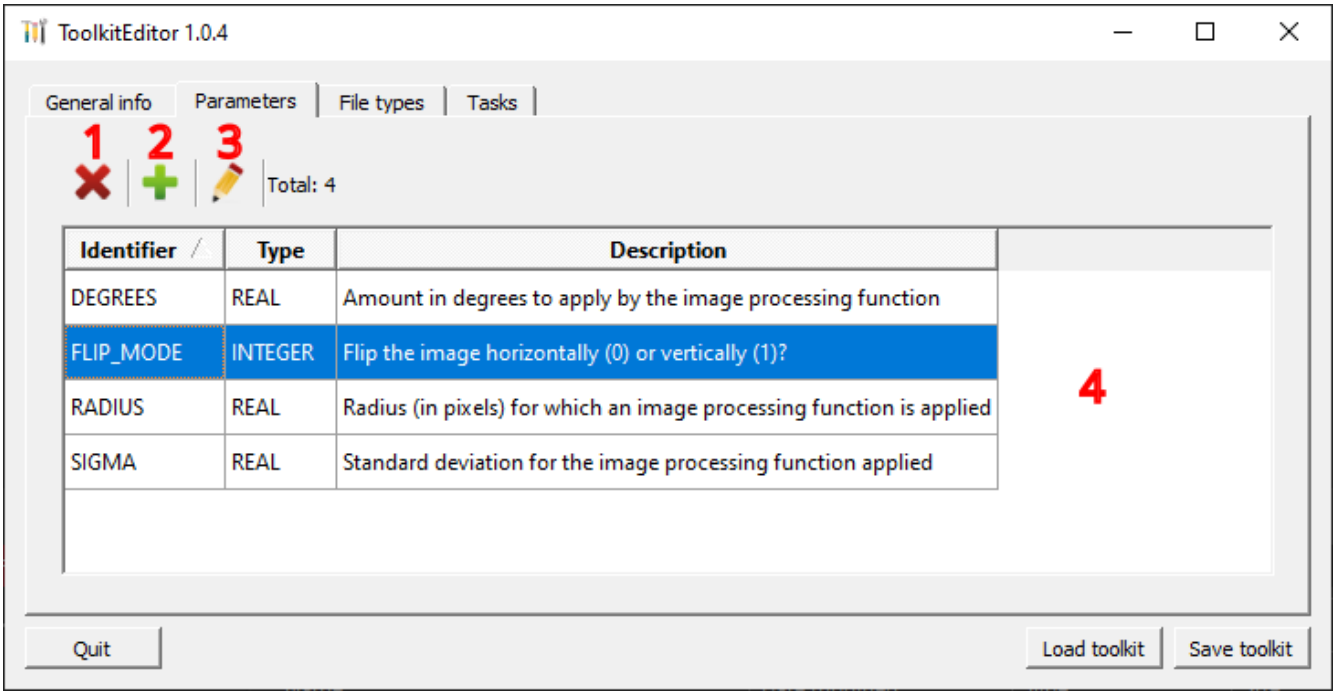

Figure 5: ToolkitEditor: the "Parameters" tab.

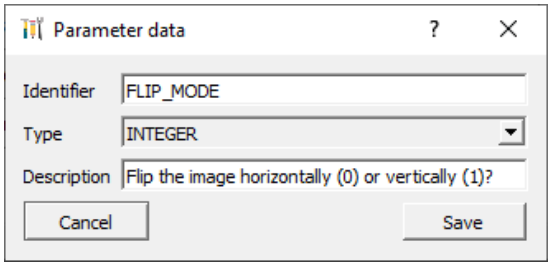

Figure 6: ToolkitEditor: adding a new parameter.

When editing an existing parameter by clicking on button (3), the window shown will be again that pictured in Figure 6. In this case, however, it is not possible to change the identifier of the parameter; only its type and description may be modified.

To delete a parameter just select it first in the parameter list (4) and click on button (1) (Figure 5). It should be noted that *it will not be possible to delete a parameter if it has already been included in any of the tasks in the toolkit,* as these tasks would then point to a non-existent parameter. In this case, if a parameter must be deleted, it will be necessary to edit each and every task that uses it and remove any references to the said parameter. Then and only then it will it be possible to delete it from the list.

Figure 7 shows the warning issued by ToolkitEditor when such a situation occurs. Note how the warning clearly states what are the tasks using the offending parameter, so, if it is really necessary to delete it, users may review the said tasks to remove it before proceeding to delete the parameter itself.

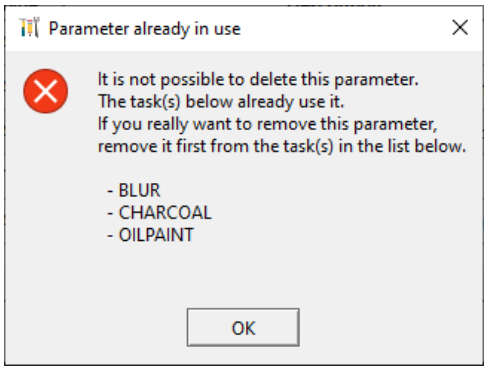

Figure 7: ToolkitEditor: warning about deleting a parameter already in use.

3.3.3 The list of file types

To add, edit or delete items in the list of file types, users must click on the “File types” tab (see (1) in Figure 3). Figure 8 shows this tab. Note that the interface is identical to that of the “Parameters” tab, so it is the way it works. However, and for the sake of completeness, a full description of the operative is included below.

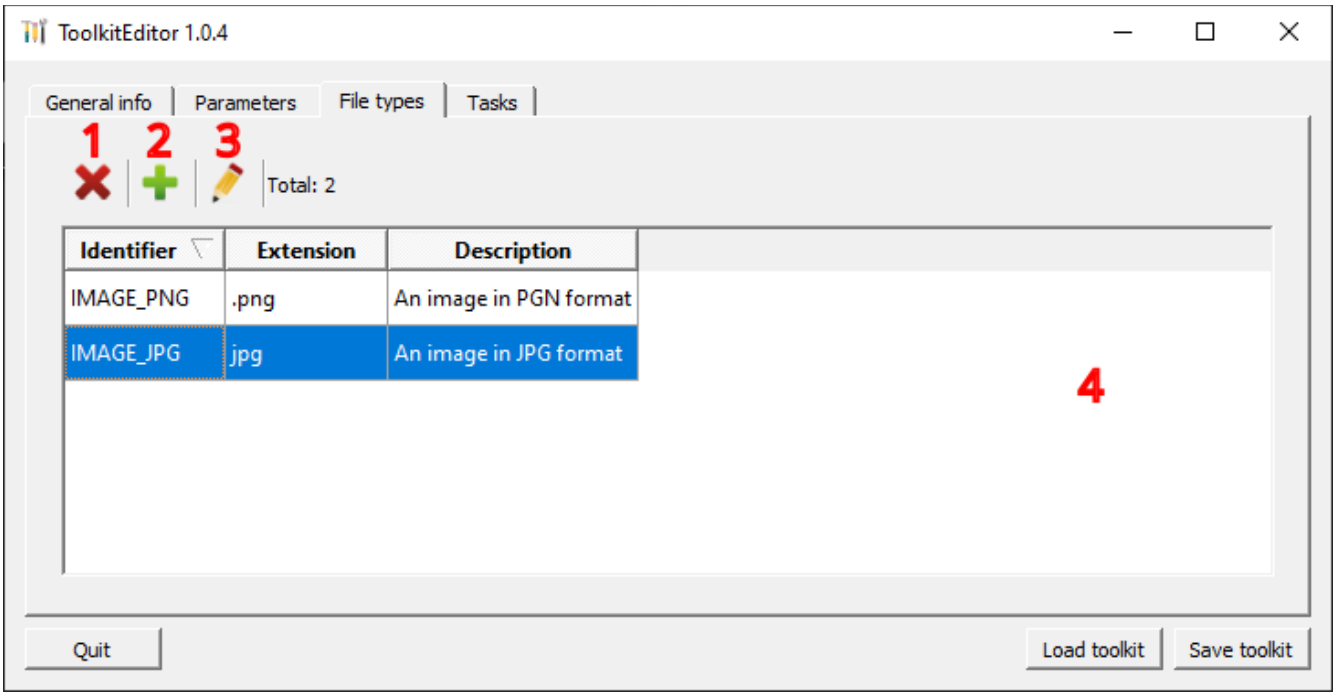

Figure 8: ToolkitEditor: the "File types" tab.

In this figure:

1. Button to delete an existing file type.
2. Button to add a new file type.

3. Button to edit an existing file type.
4. List of file types.

Note that the delete (1) and edit (3) buttons are only enabled when a file type has been selected in the list of file types (4) as shown in Figure 8.

To add a new file type, just click on the add (2) button. The window in Figure 9 will appear. There, it is possible to type the **unique** identifier (options file label) for the file type, as well as its description and default extension. By clicking on the “Save” button, the file type will be added to the list of file types.

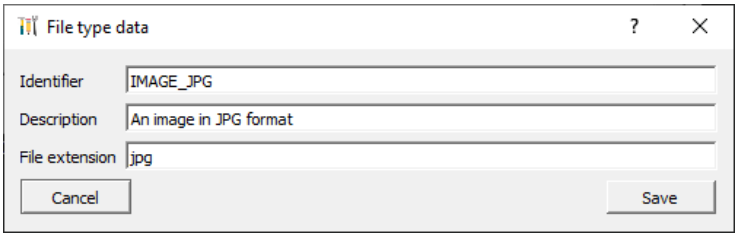

Figure 9: ToolkitEditor: adding a new file type.

When editing an existing file type by clicking on button (3) (Figure 8), the window shown will be again the one pictured in Figure 9. In this case, however, it is not possible to change the identifier of the file type; only its description and file extension may be changed.

To delete a file type just click on button (1) (Figure 8). It should be noted that it will not be possible to delete a file type if it has already been used by any of the tasks in the toolkit, as these tasks would then point to a non-existent file type. In this case, if you really want to delete the file type, it will be necessary to edit each and every task that uses it and remove the files belonging to the offending file type. Then and only then it will it be possible to delete it from the list.

Figure 10 shows the warning issued by ToolkitEditor when such a situation occurs. Note how the warning clearly states what are the tasks including files whose file type is the offending one, so, if it is really necessary to delete it, users may review the said tasks to remove the necessary files before proceeding to delete the file type itself.

3.3.4 The list of tasks

By clicking on the tab “Tasks” (see (1) in Figure 3) users will be able to manage the list of tasks. Figure 11 shows this tab.

Once more, the procedure to delete, edit or add a task (buttons 1, 2 and 3 in Figure 11) works in the same way than it does for keyboard parameters (section 3.3.2) and file types (section 3.3.3). However, the list of tasks (area 4 in the figure) shows just a summary for each task instead of their full descriptions because of the volume of information that should be included.

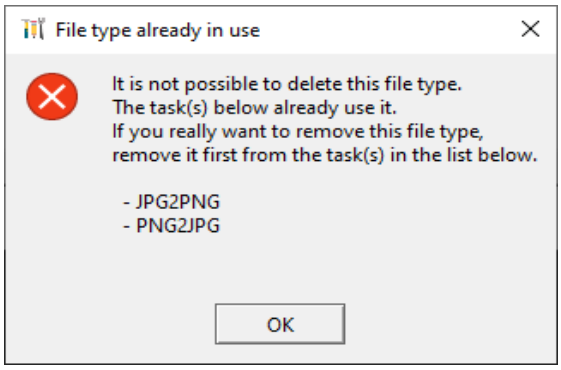

Figure 10: ToolkitEditor: warning about deleting a file type already in use.

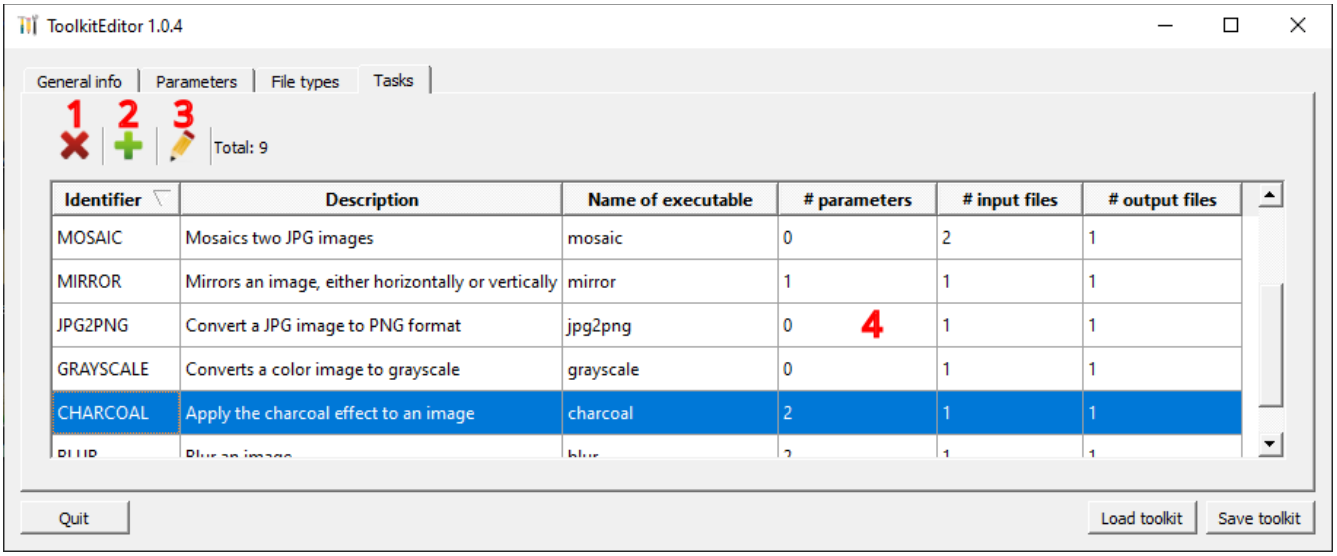

Figure 11: ToolkitEditor: the "Tasks" tab.

Again, to edit or delete a task it is necessary to select it in the list of tasks (as depicted in Figure 11) and click on the corresponding button (either 1 or 3). It is always possible to delete a task, since there are no other data entities that rely on these (as opposite to keyboard parameters or file types, which may be used by one or more tasks).

When adding a new task (button 2) or editing an existing one (button 3), a window like the one shown in Figure 12 will show up. This window is divided into four areas (numbered from 1 to 4 in that figure).

In area number 1, the unique identifier of the task, its description and the name of the executable file implementing it may be typed. *Note that it is not recommended to append the ".exe" extension to the name of the executable;* to run an executable file on Windows computers it is not necessary to specify its extension; on Linux computers, executable files do not carry an extension. Therefore, not appending the said extension makes possible to reuse a toolkit file in both platforms (assuming that the tasks involved have been migrated to both operating systems).

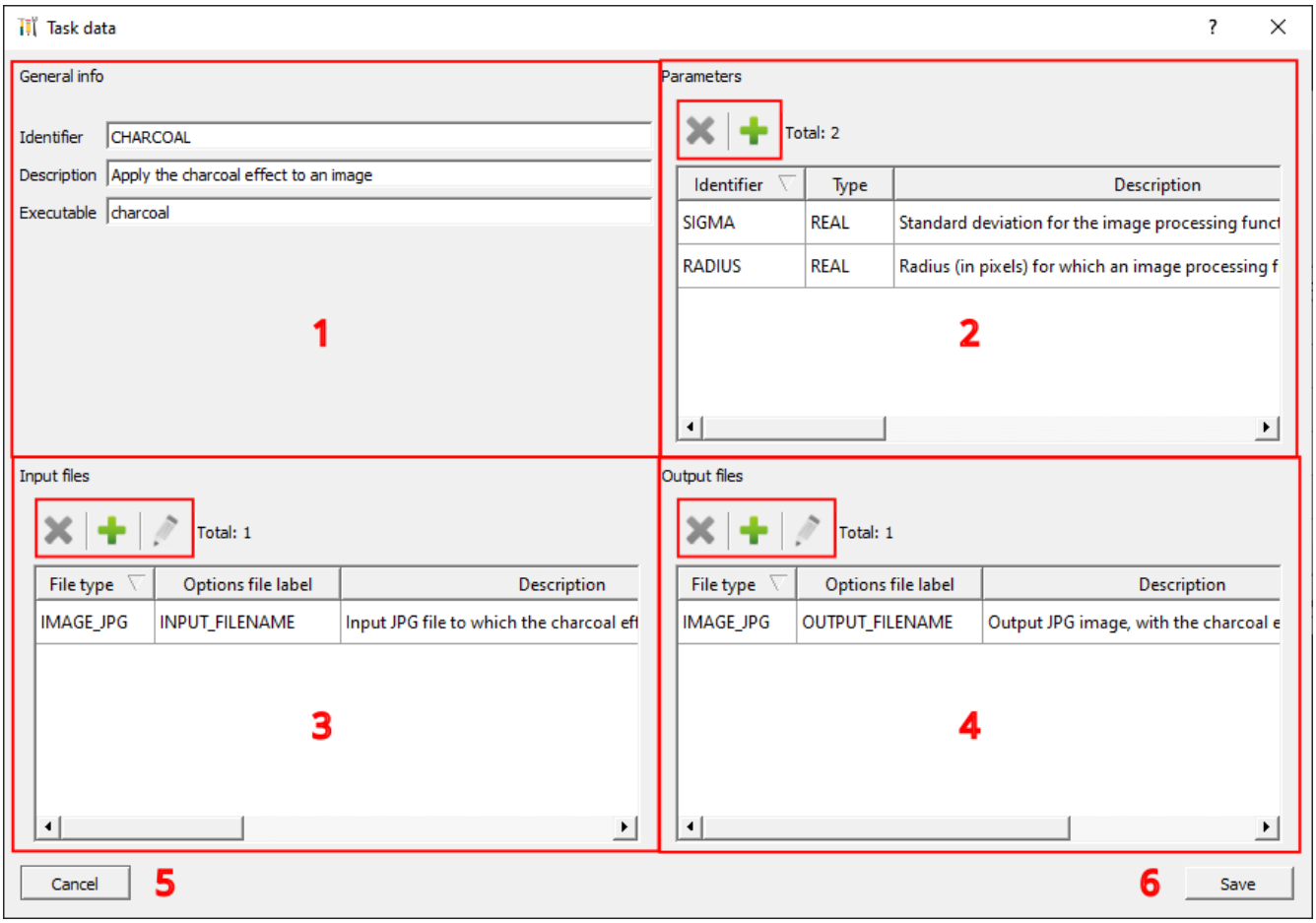

Figure 12: ToolkitEditor: adding or editing a task.

Area number 2 is devoted to the management of the list of keyboard parameters. Note the toolbar on the top left corner of this area: by clicking on the buttons there it is possible to add or remove parameters to or from the list. It is worth to highlight that *it is not possible to edit such parameters,* since in this case only *references* to those existing in the global list of keyboard parameters (section 2.2.2) are included here.

When adding a new parameter a window like the one shown in Figure 13 will appear. The window shows the list of available parameters, as defined in the global list. The user needs only to select one of them among those available and click on the confirmation button (number 2 in the figure) to add it to the list of parameters. On the contrary, it is possible to cancel the operation by just clicking on the reject button (number 1).

Areas 3 and 4 are provided to manage the list of input and output files respectively. In this case both toolbars allow not only for adding or removing files, but also for editing these. The reason is that, in this case, file types are just a component of a more complex data structure: files, here, state their file type but include also an options file label as well as a specific description, that is, the role played by the file in the task.

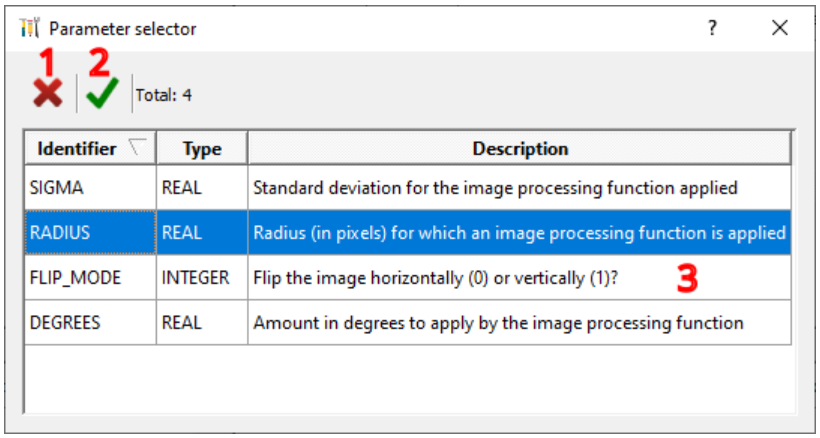

Figure 13: ToolkitEditor: managing the parameters in a task.

Again, it is possible to remove, add or edit a file (either input or output) clicking on the respective buttons included in the toolbars present in the top-left corners of areas 3 and 4.

When adding or editing a file, a window like the shown in Figure 14 appears.

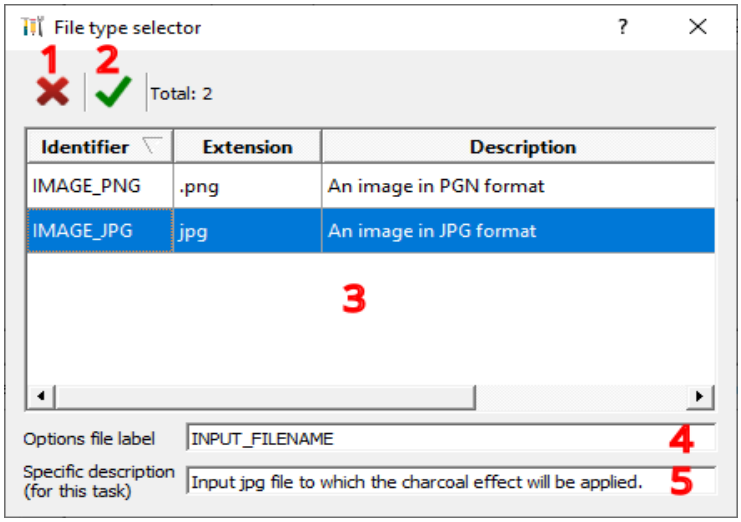

Figure 14: ToolkitEditor: managing the input / output files in a task.

To add or edit a file using the window shown above, users must:

- Select one of the available file types shown in the area labeled 3. These are the ones defined in the global list of file types (section 2.2.1).

- Type (field with label 4) an options file label that will be used to give a name to this file when executing the workflow. Note that options file labels must be **unique** within the context of a task. That is, different tasks may use repeated option file labels, but inside a task all labels

(including that of the parameters) must be unique.

- Type (field with label 5) *a description stating the role of this file in this particular task.* Note that a generic description is already available (that of the file type to which this file belongs), but the idea here is to describe what is this file used for by this specific task. In the figure, the generic description is “An image in .JPG format”, which is, indeed, generic enough. The description given for this specific task is “Input .jpg file for which the charcoal effect will be applied”. This states clearly what is the role of this file.

- Either accept (button 2) or reject (button 1) the changes.

Once that the general description, the lists of parameters, input and output files have been completed, it is possible to save the definition of the task by clicking the button “Save” (number 6 in Figure 12) or reject all changes by clicking the button “Cancel” (number 5 in the same figure).

3.3.5 Saving the toolkit

Click on button “Save toolkit” (labeled with a number 4 in Figure 3) the typical process to select a folder and a file name will start. ToolkitEditor files are saved as eXtensible Markup Language (XML) files, so do not forget to add the “.xml” extension to the name of the output file.

Note that at least one keyboard parameter, one file type and one task must have been defined before saving a toolkit file is possible. Otherwise, ToolkitEditor will complain about missing data.

3.3.6 Loading an existing toolkit

Click on button “Load toolkit” (labeled with a number 3 in Figure 3) it is possible to load an existing toolkit. The procedure to do it is the typical one where the folder and the file itself must be selected by means of a file selection dialog box.

3.3.7 Caveats: please *do* read this section very carefully

Please, read this section very carefully since it explains several limitations affecting ToolkitEditor that must be taken into account to avoid having problems later.

Editing the toolkit file in several sessions

As stated in section 3.3.5, it is necessary define at least one keyboard parameter, one file type and one task to be able to save a toolkit. This is a limitation of the current version of ToolkitEditor.

This may be rather inconvenient, specially because defining a whole toolkit may take several work sessions to complete it. To avoid these inconveniences, the author recommend to proceed as described below:

- Type the general information (identifier and description) for your toolkit.

- Define at least one keyboard parameter.
- Define at least one file type.
- Define just one **fake** task using the keyboard parameter and one input file and one output file belonging to the unique file type defined.

At this point it is possible to save the toolkit file.

Then proceed systematically:

- Define the whole set of keyboard parameters.
- Define the whole set of file types.

Again, it is possible to save the toolkit file as many times as necessary, since the conditions required to do it (having at least one parameter, file type and task) are met.

Then, and only then, it is recommended to define, one by one, the different tasks in the toolkit. Having defined the whole set of keyboard parameters and file types will guarantee that all tasks may be fully defined too.

Once more, it is possible to save the toolkit file as many times as needed (for instance, after defining each new task).

When all the tasks have been defined, delete the **fake** task created at the very beginning of this procedure and save the toolkit for the last time.

This procedure is recommended when defining complex toolkits consisting of a rather high number of parameters, file types and tasks. Of course, a toolkit may be defined in a single editing session if it is small enough as to make it possible.

It is highly recommended (see next subsection) that no workflows are designed relying on some toolkit until it is guaranteed that the said toolkit has been fully defined.

Modifying a toolkit for which workflows already exist

As soon as a workflow relying in some toolkit is created (see section 3.4), the toolkit should become immutable; it should not be changed anymore.

This is so because modifying a toolkit for which workflows have been created will invalidate the said workflows as well as the launchers (see section 3.5) relying on these workflows, and even making crash the WorkflowEditor and WorkflowLauncher tools.

Obviously, toolkits may grow with time, since new tools may appear after the toolkit file has been defined. This is a problem that the current version of WorkflowMaker has not yet solved. Thus, if a

toolkit file must be extended due to the inclusion of new tools, a safe way to proceed nowadays would be:

- Copy the original toolkit file to a **new** one.
- Edit the new toolkit and remove all the tasks.
- Add the new keyboard parameters and file types that might have appeared due to the inclusion of new tasks.
- Re-define the old tasks and add the new ones.
- Save the toolkit file.

Then, all the workflows relying on the old toolkit should be redrawn and saved, All the launchers relying on these workflows should be rebuilt.

Following this costly procedure guarantees that there will be no multiple versions of some toolkit.

There is a simpler solution that, although reducing the effort at first, may increase the complexity of the management of toolkits, workflows and editors:

- Keep the old toolkit, workflows and launchers, and keep using these as usually, that is, running the workflows already defined with no changes.
- Create the new toolkit (including as stated above, the new tools) according to the procedure described in the previous subsection. Do not forget to give the toolkit a new, unique identifier and description to tell it apart from the original one.
- Use the *new* toolkit to design only *new* workflows and the respective *new* launchers.

In this way, the result would be having two versions (or more, if the need to extend the toolkit appears again in the future) of the same toolkit, namely version 1 and 2. Some of the workflows and launchers will correspond to version 1 while the newer ones to version 2. This is the reason why saving some effort at first, avoiding the redefinition of existing workflows and launchers leads to a more complicated management process, due to existence of several versions of some toolkit.

3.4 WorkflowEditor

This section will describe how to use the WorkflowEditor tool to design workflows visually, To do it, an example workflow based on the example toolkit defined in section 3.1 will be described below. This is just an example workflow; many others may be created relying on the same toolkit.

3.4.1 The example workflow

The example workflow must perform the following tasks:

1. It will take a single input color image in PNG format, stored in some folder, and convert it to JPG format, since all the tools in this toolkit but the converters work with JPG images.

2. It will create three new versions of these images, applying the effects known as blur, charcoal and oil paint.
3. These three images, together with the original one, will be joined in a mosaic.
4. Then, the mosaic will be mirrored around the vertical axis.
5. The mirrored image will be rotated 90 degrees clockwise.
6. The rotated image will be converted to gray scale.
7. Finally the gray scale image will be converted to PNG format and saved to some output folder.

All tasks but the one with number 3 may be carried out directly by some of the tools included in the image processing toolkit. For instance, the tool named MIRROR will take care of step 4; to convert from PNG to JPG or conversely the tools named PNG2JPG and JPG2PNG respectively will be used.

On the contrary, the MOSAIC tool is able to join together only two images at a time. Therefore, step 3 must be performed as follows due to the limitations of this tool:

A. First, mosaic the blurred image with the one to which the charcoal effect has been applied.
B. Then, mosaic the result of step A and the image that has undergone the oil painting effect. The resulting image now includes three parts: blur + charcoal + oil painting.
C. Afterwards, mosaic the result of step B and the original image. The result is the image sought, including three of them showing different image effects plus the original one at the right side.

Obviously, the procedure to mosaic four images is conditioned by the limitations of the example tool MOSAIC. Real toolkits will have their own set of tools with their own limitations.

Figure 15 depicts the procedure described above. Note the text in red color labeling the steps in the said procedure (step 3 includes the three separate mosaic operations, A, B and C, for the sake of clarity). The red, dotted rectangles are not part of the WorkflowEditor interface, but have been added to make clear what are the said steps when these involve more than a single box (either for repositories or tasks).

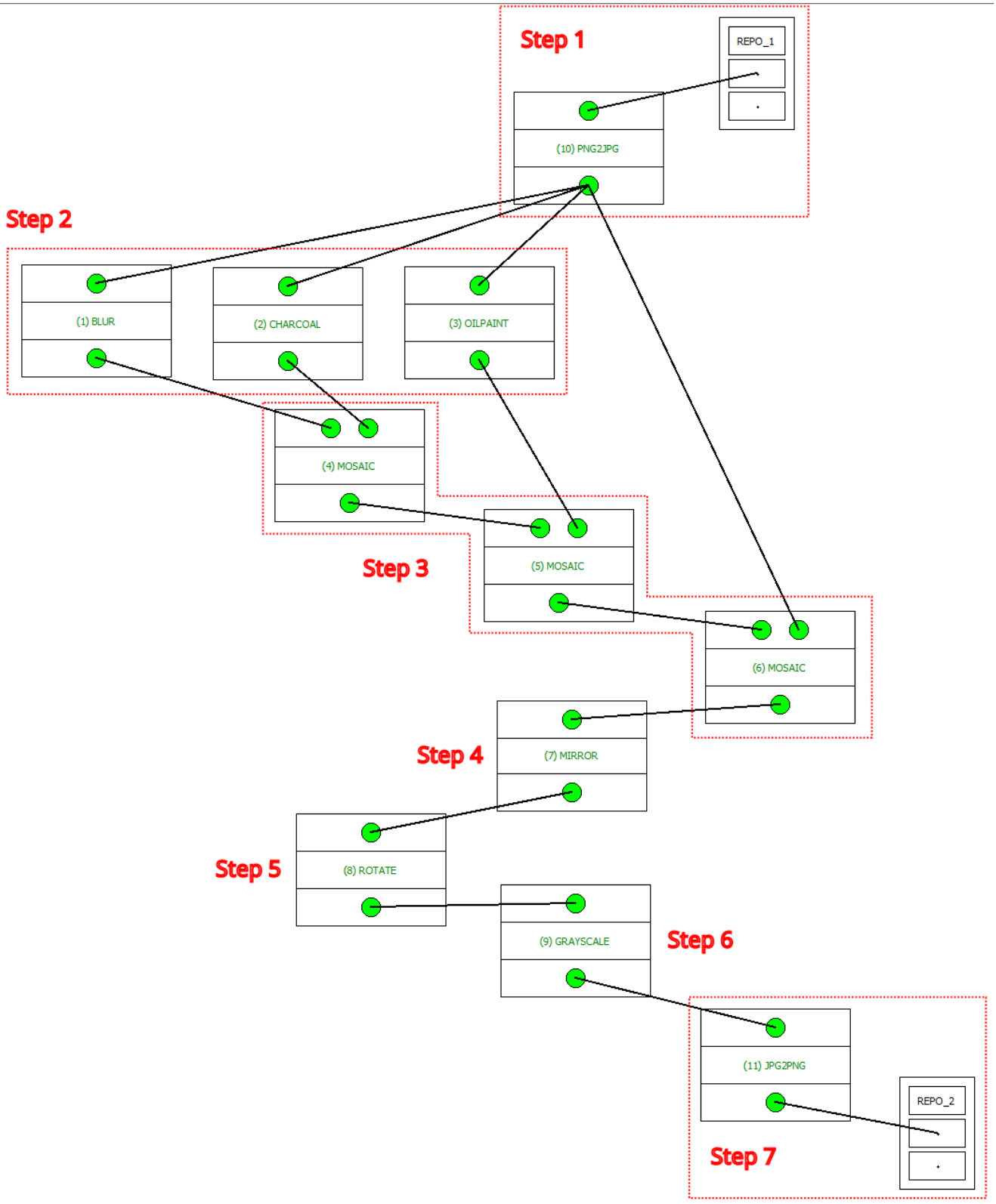

Figure 15: WorkflowEditor: the visual representation of the example workflow.

3.4.2 The interface of WorkflowEditor

Figure 16 depicts the interface of WorkflowEditor while editing the example workflow.

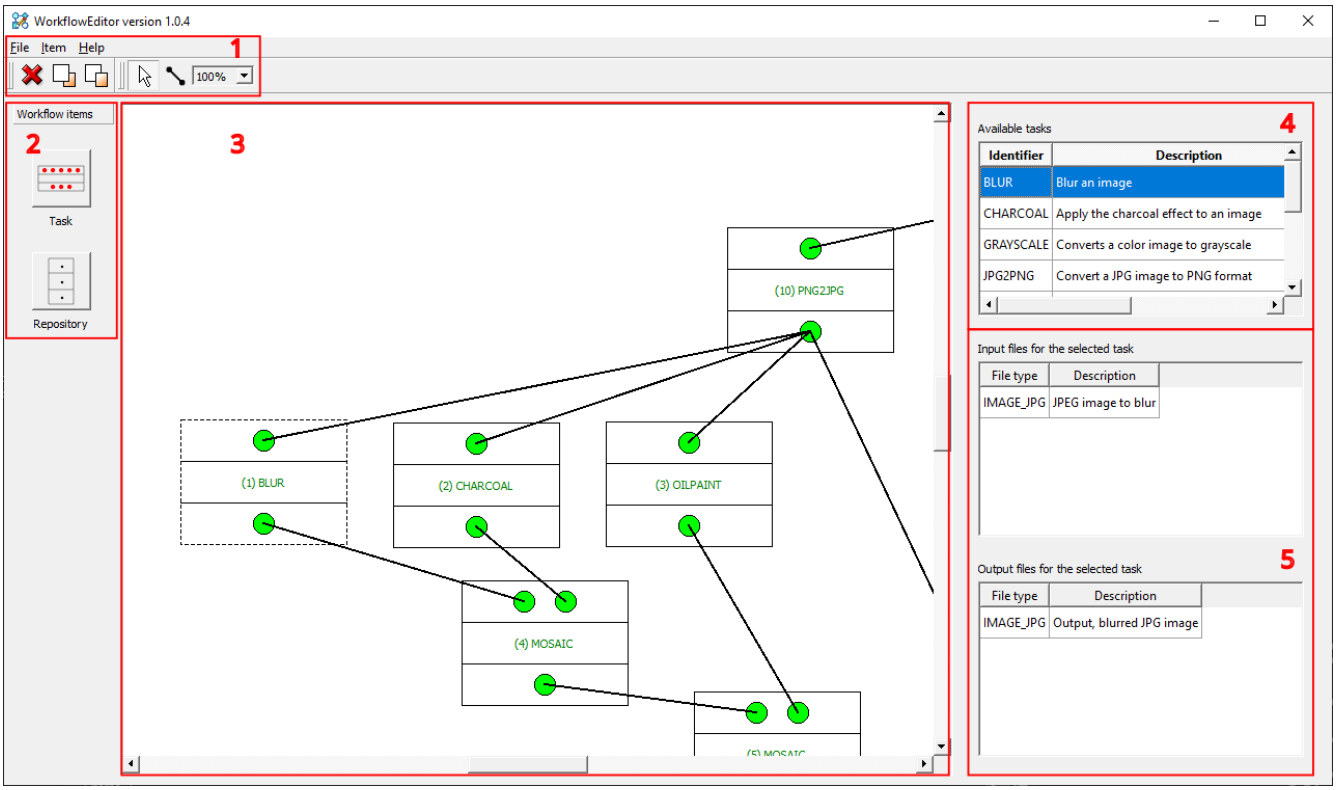

Figure 16: WorkflowEditor: the Graphical User Interface (GUI).

In this figure:

1. The menu and toolbar.
2. The elements (tasks and repositories) that may be inserted in a workflow.
3. The drawing area.
4. The task selector.
5. The list of input and output files related to the task being selected in (4) above.

The menu and the toolbar (GUI component #1)

Figure 17 shows the whole set of menus as well as the toolbar.

The first menu (Figure 17(a)) includes the several options related to file management, such as opening, creating or saving a workflow, plus quitting the application.

The second menu (Figure 17(b)), which directly corresponds to the left part of the toolbar (Figure 17(d1)) let users either delete a task or repository already inserted in the workflow.

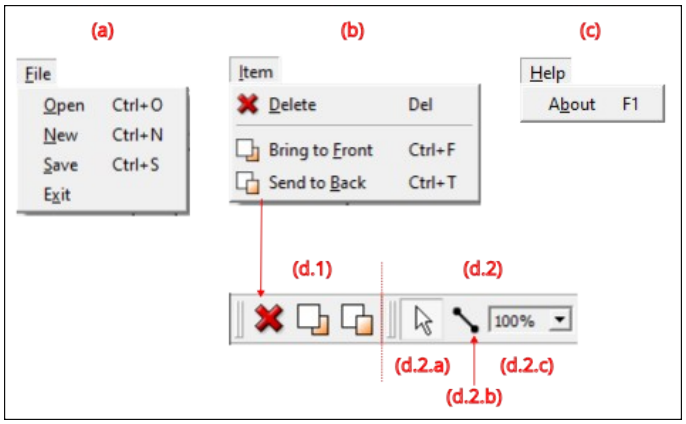

Figure 17: WorkflowEditor: the menus and the toolbar.

Additionally, it is possible to bring such items to front or back when these are obscured by other items in the workflow.

The third menu (Figure 17(c)) only includes an option to show a window providing some details about the application.

Finally, using the controls in the right part of the toolbox (Figure 17(d.2)) it is possible to choose the “Select / Move / Insert” pointer mode (Figure 17(d.2.a)), the “Add Connection” pointer mode (Figure 17(d.2.b)) or zoom in / out the workflow (Figure 17(d.2.c)).

The item selector (GUI component #2)

Figure 18 below depicts another component of the WorkflowEditor GUI: the item selector.

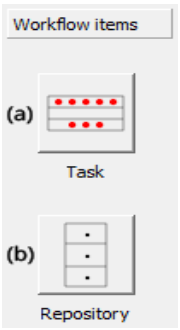

Figure 18: WorkflowEditor: the item selector.

It is used to select what to insert, that is, a task (Figure 18(a)) or a repository (Figure 18(b)), when an insertion operation is taking place.

The drawing area (GUI component #3)

In this part of the interface is where repositories and tasks and inserted and then linked by means of connections stating how information flows.

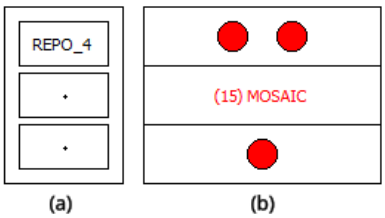

Figure 19: WorkflowEditor: representation of a repository and a task.

Figure 19 above depicts how repositories (a) and tasks (b) are shown when inserted in the drawing area. The name of repositories is generated automatically so they can be told apart. Tasks are a bit more complicated. These are divided in three stripes; in the central one the identifier of the task preceded by a numeral – also generated automatically – is included. The said numeral let users distinguish between two or more occurrences or *instances* of tasks running the same application (as MOSAIC in the figure above). The upper and lower stripes include a variable number or circles; these represent each of the input (above) or output (below) files that the task reads or creates, respectively. The color of these circles represents their status: red, not yet connected, green stands for connected while blue means that it is a valid candidate for a connection.

Connections themselves are represented by lines between two endpoints (see the example workflow in Figure 15). There, all files are connected so all circles are painted green.

The task selector (GUI component #4)

Figure 20 portraits the task selector component of the WorkflowEditor GUI. It is used to select the task that will be inserted when an insertion process is going on. Additionally, it updates the contents of the input / output files component

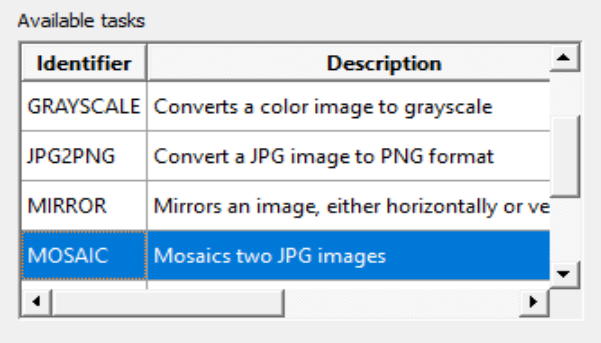

Figure 20: WorkflowEditor: the task selector.

The input / output files component (GUI component #5)

This is a read-only component of the WorkflowEditor GUI. It shows the list of input and output files for the task selected in the task selector. In this way, users are aware of what are these files and their role in the task (since descriptions of such roles are included). Figure 21 shows this component.

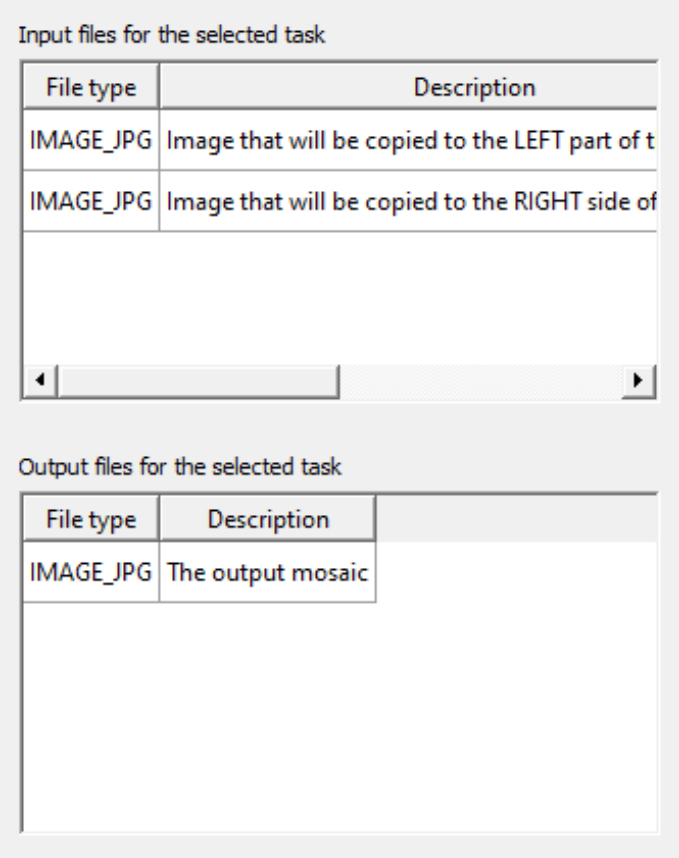

Figure 21: WorkflowEditor: the input / output files component.

The order (from top to bottom) in which the files appear corresponds to the order (from left to right) in which the circles are displayed when inserting tasks into the drawing area. This applies to both input and output files. For instance, the two circles in the upper part of task MOSAIC in Figure 19 correspond, respectively, to the input left and right images (see Figure 21) used to create the mosaic.

This way, the user can accurately determine the file type (and the role it plays) associated with each of these circles when connecting repositories and tasks together.

3.4.3 Creating, opening or saving workflows

To create a new workflow select the “New” entry in the “File” menu. Immediately, a window like the one shown in Figure 22 will appear.

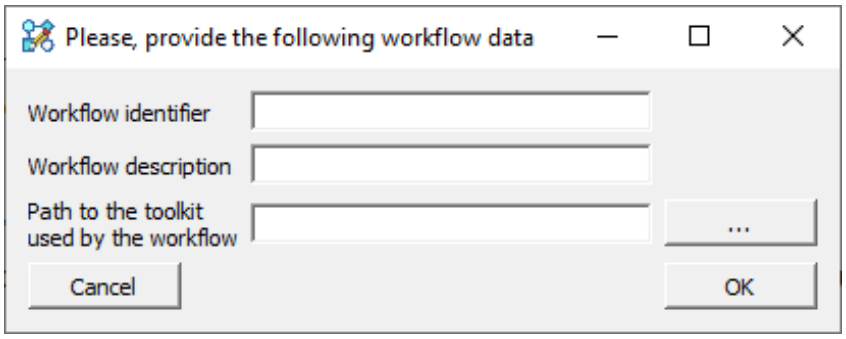

Figure 22: WorkflowEditor: metadata for a new workflow.

Here, a **unique** identifier as well as a textual description for the workflow must be typed. Additionally, it is necessary to locate the toolkit on which this workflow will rely. The identifier of such toolkit will be saved in the workflow file, ensuring that it will always be possible to identify (and use) which toolkit the workflow belongs to.

Click on the “OK” or “Cancel” buttons to either proceed to create the new workflow or to cancel this action.

Opening an existing workflow may be achieved by selecting the “Open” entry in the “File” menu. Immediately, the typical dialog to locate a file will show up. Once the workflow file is selected, a window asking for the toolkit file on which the workflow relies will show up. See Figure 23.

Note that in this window the identifier of the toolkit (‘IMAGE_PROCESSING_TOOLKIT’ in this example) is shown to let users know what is the toolkit the workflow relies on, so the appropriate file is selected in the next step; as soon as the “OK” button is clicked, another file dialog will show up to locate the toolkit file sought. Once it is identified, the workflow will be opened by WorkflowEditor.

This process may be canceled twice, that is, every time that a file dialog appears.

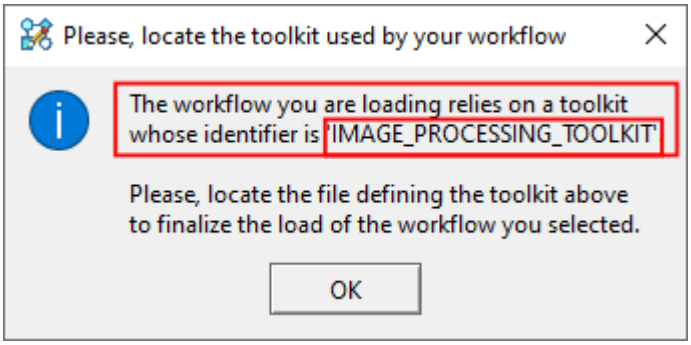

Figure 23: WorkflowEditor: asking for the toolkit on which a workflow relies.

To save a workflow just click on the “Save” entry in the “File” menu. Look for the appropriate folder and file name using the emerging file dialog to save the workflow.

Note that, when saving incomplete workflows, warnings stating this fact may appear. A typical warning is shown in Figure 24:

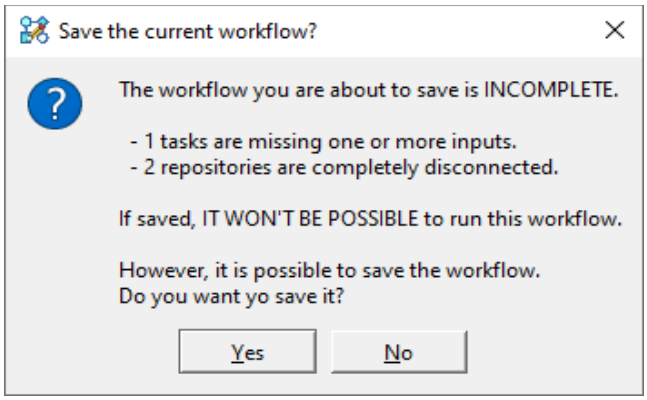

Figure 24: WorkflowEditor: warnings on incomplete workflow when trying to save it.

Nonetheless, even when warnings are issued, it is still possible to save the workflow. As stated in the figure, it will not be possible, however, to run it using WorkflowLauncher (section 3.5).

3.4.4 Inserting, moving and deleting repositories and tasks

To insert a new repository in the drawing area, follow the steps below:

1. Make sure that the pointer mode is set to “Select / Move / Insert” as shown in Figure 17(d.2.a).
2. Click on the repository icon in the item selector component (Figure 18(b)).
3. Click on the drawing area close to the place where the repository should stay (it may be moved later).

To move a repository to some other part of the workflow:

1. Make sure that the pointer mode is set to “Select / Move / Insert” as shown in Figure 17(d.2.a).
2. Place the cursor over the repository to move. Click and hold the left button of the mouse and move it to drag the repository. Release the mouse button when finished.

To delete a repository:

1. Make sure that the pointer mode is set to “Select / Move / Insert” as shown in Figure 17(d.2.a).
2. Click on the repository to delete.
3. Either (a) hit the “Del” key or select the entry “Delete” in the “Item menu” (Figure 17(b)) or click on the red diagonal cross in the toolbar (Figure 17(d.1)).

Deleting a repository means deleting all the connections starting or ending at it.

To insert a new task in the drawing area:

1. Make sure that the pointer mode is set to “Select / Move / Insert” as shown in Figure 17(d.2.a).
2. Select the task to insert clicking on it in the task selector component (Figure 20) so it is highlighted.
3. Click on the task icon in the item selector component (Figure 18(a)).
4. Click on the drawing area close to the place where the task should stay (it may be moved later).

To move a task to some other part of the workflow:

Proceed in the same way than when moving a repository. Click on the desired task instead.

To delete a task:

Again, proceed as when deleting a repository but selecting a task instead.

Deleting a task means deleting all the connections starting or ending at it.

3.4.5 Inserting and deleting connections

Connections materialize the flow of information (files) between repositories and tasks. These are represented by lines connecting two endpoints.

Before describing how to manage connections, it is very important to know some restrictions affecting the way these may be created. But first, a couple of definitions will help to simplify the wording of the said restrictions.

Input slot: any of the circles in the top of some task (Figure 19), which represents an input file.

Output slot: any of the circles in the bottom of some task (again, Figure 19), which represents an output file.

These are the restrictions:

1. There may be just a single connection ending at some task’s input slot. In other words, it is not possible to have multiple sources to provide a single input file.

2. On the contrary, an output slot can be used in multiple connections, thus feeding different tasks and/or repositories. In fact, there is no limit to the number of connections that start from a task’s output slot.

3. Repository to repository connections are not allowed. This would mean copying a file from one folder to another one.

4. Connections starting in a repository may end in any task's input slot, that is, all files residing in a repository can be used to provide input files to all tasks.

5. Connections starting at some task's output slot may always end at a repository. In other words, repositories may store files of any kind.

6. Connections starting at some task's output slot may only be connected to some other task's input slot if the file types of both slots are the same. For instance, an output spreadsheet may be used by some task that needs a spreadsheet as one of its inputs to work. On the contrary, it is not possible to provide a spreadsheet when some task needs an input file containing a list of coordinates.

7. All the input slots of all tasks must be connected. The reason is simple: if just a single input file is missing for some task, the said task will not be able to work.

8. At least one of the output slots of all tasks must be connected. Leaving all the outputs of some task disconnected means that none of its outputs is necessary. The immediate implication is that such task is not needed by the workflow and, therefore, it should not be included there. On the contrary, it is enough to connect just one of its outputs; this means that the remaining ones are not required by the workflow (and will be silently deleted by the scripts generated by WorkflowLauncher, see section 3.5).

All these restrictions are implemented by WorkflowEditor, so no "illegal" connections are allowed.

The input and output slots in a task are represented as circles (Figure 19) painted in different colors depending on their status:

- Red: the slot (either input or output) has not been connected yet.

- Green: the slot (again, input or output) has been connected.

- Blue: only for input slots; when a connection is started, either from a repository or another task's output slot, all the compatible and not yet connected input slots become temporarily blue to indicate that they are willing to become the final endpoint of such connection. In the previous sentence, "compatible" means "having the same file type", and "not yet connected" implies that such candidate input slots were painted in red before the connection was initiated.

The coloring schema described above is very useful with regard to restriction number 6 in the list above, since offers a clear visual feedback about what are the candidates that may be used to finish

an ongoing connection. Figure 25 depicts the behavior of (the coloring of) the slots when initiating a connection.

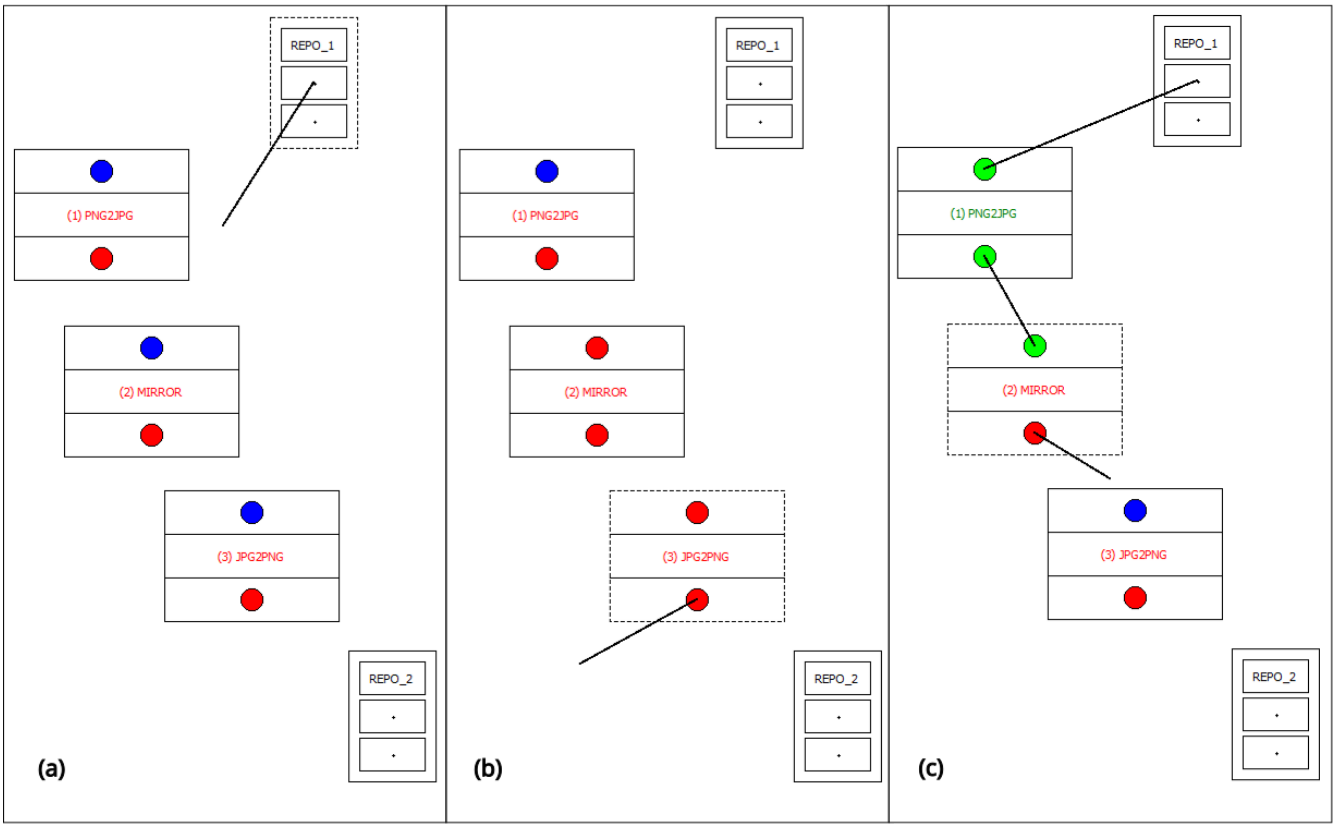

Figure 25: WorkflowEditor: coloring of slots when initiating a connection.

For instance, in Figure 25(a) all unconnected input slots are candidates to accept a connection starting in a repository. In Figure 25(b), on the contrary, only the task named PNG2JPG is willing to accept the output of JPG2PNG, a PNG file, since PNG2JPG is the only task in the workflow that takes this type of files as input. Finally, Figure 25(c), the output of MIRROR, a JPG file, may be accepted only by JPG2PNG (as well as any of the available repositories) since this is the only task with an unconnected input slot compatible with such file type.

To insert a new connection starting at a repository:

1. Make sure that the pointer mode is set to “Add connection” as shown in Figure 17(d.2.b).
2. Click (left mouse button) on the repository; do not release the button. All the unconnected input slots will become blue.
3. Move the pointer to the desired unconnected input slot and release it.
4. The input slot will become connected and its color will turn to green.

To insert a new connection starting at a task's unconnected output slot:

1. Make sure that the pointer mode is set to "Add connection" as shown in Figure 17(d.2.b).
2. Click (left mouse button) on the task's unconnected output slot; do not release the button. All the unconnected, compatible input slots will become blue.
3. Move the pointer to either a compatible, unconnected task's input slot or to a repository and release the mouse button.
4. The output slot where the connection was initiated will become connected and its color will turn to green.
5. If the connection ended in some task's compatible, unconnected input slot, it will become connected too and its color will also turn to green. On the contrary, if the connection ended in a repository, no visual feedback is provided with regard to the repository.

To delete a connection, no matter what are its endpoints:

1. Make sure that the pointer mode is set to "Select / Move / Insert" as shown in Figure 17(d.2.a).
2. Click (left mouse button) on the connection.
3. Either (a) hit the "Del" key or select the entry "Delete" in the "Item menu" (Figure 17(b)) or click on the red diagonal cross in the toolbar (Figure 17(d.1)).

Deleting a connection will make any involved slot, either input or output, to return to the unconnected state and be colored in red.

3.4.6 Caveats: please *do* read this section very carefully

Please, read this section very carefully since it explains important issues related to the use of WorkflowEditor especially when related to WorkflowLauncher.

Do not modify a workflow for which there are already WorkflowLauncher files relying on it

The workflows created by WorkflowLauncher (section 3.5) are used by the next tool in the WorkflowMaker suite, WorkflowLauncher.

A workflow is just a template; with WorkflowLauncher actual values for the keyboard parameters as well as for the paths of the repositories and input and output files are provided. However, if the underlying workflow is modified, the "launcher files" (the files produced by WorkflowLauncher to store these values) may become unusable because its contents will not match the template defined by the original workflow. Furthermore, trying to open a launcher file relying on a modified workflow may make WorkflowLauncher crash.

Therefore, if a workflow needs to be modified and launcher files relying on it exist, it is recommended to create a new workflow with a new identifier and description to avoid the problems reported above.

It is possible to duplicate the original workflow and the use the copy to modify it. However, this is a bit tricky, since the workflow identifier should be changed to avoid conflicts with the original one; the general description stating its purpose should be also changed. However, the current version of WorkflowEditor provides no mechanism to change these identifier and description strings. It must be done manually by opening the workflow file using any text editor, and then replacing the contents of the fields delimited by the <id> and <description> tags; then, the file may be saved and edited with WorkflowEditor.

For instance, the original contents of the said fields could be:

```
<id>IMAGE_PROCESSING_WORKFLOW</id>
<description>A workflow combining several image processing tools</description>
```

Duplicating the workflow file and changing the contents of these fields could deliver the following result:

```
<id>IMAGE_PROCESSING_MODIFIED_WORKFLOW</id>
<description>A workflow combining some more tools than before</description>
```

Then saving the modified workflow file as a regular text file will be enough to have a new version of the workflow that may be edited safely with WorkflowEditor, respecting the old one and the launcher files that depended on it.

3.5 WorkflowLauncher

This section will describe how to use the WorkflowLauncher tool.

This tool performs two tasks:

1. Providing actual values for the keyboard parameters, repository paths and input and output file names required by some workflow. Such values may be saved, loaded and modified at will in the so-called **launcher files.**

2. Create scripts, either for the Windows command line or the bash console on Ubuntu-based Linux distributions, including the necessary commands to execute the workflow on which some launcher file relies, using the values that have been stored in there. Create the necessary options files as well.

Most of the following subsections will describe task 1 in the list above, since creating the scripts and options files, although the final goal of the whole WorkflowMaker software suite, is a rather simple task requiring very little explanation.

3.5.1 The interface

Figure 26 depicts the interface of WorkflowLauncher when no launcher file is open. In this figure:

1. Tab selector. Here it is possible to switch between the three tabs into which the application is divided, so the different kinds of values (keyboard parameters, paths to repositories, names of the input and output files) may be reviewed separately.
2. Data area. The information relative to the different said tabs will be shown here when editing a launcher file.
3. Button to quit the application.
4. Button to load an existing launcher file.
5. Button to create a new launcher file.
6. Button to save a launcher file.
7. Button to generate the script (shell) and option files needed to, finally, execute the workflow with the values defined by the user.

3.5.2 Creating a new launcher file

A launcher file depends on some workflow which, in turn, depends on some toolkit. Therefore, before creating a new launcher file it is necessary to know exactly which are the files that, respectively, contain said workflow and toolbox. The paths to these files will be requested when creating a new launcher file.

This is how a new launcher file is created:

First, click on the “New launcher” button (see Figure 26 on page 42). The window in Figure 27 (page 42) will show up. In this window, as usual, a **unique** identifier as well as a description stating the purpose of the launcher must be provided. Moreover, it is necessary to locate the workflow on which the launcher relies.

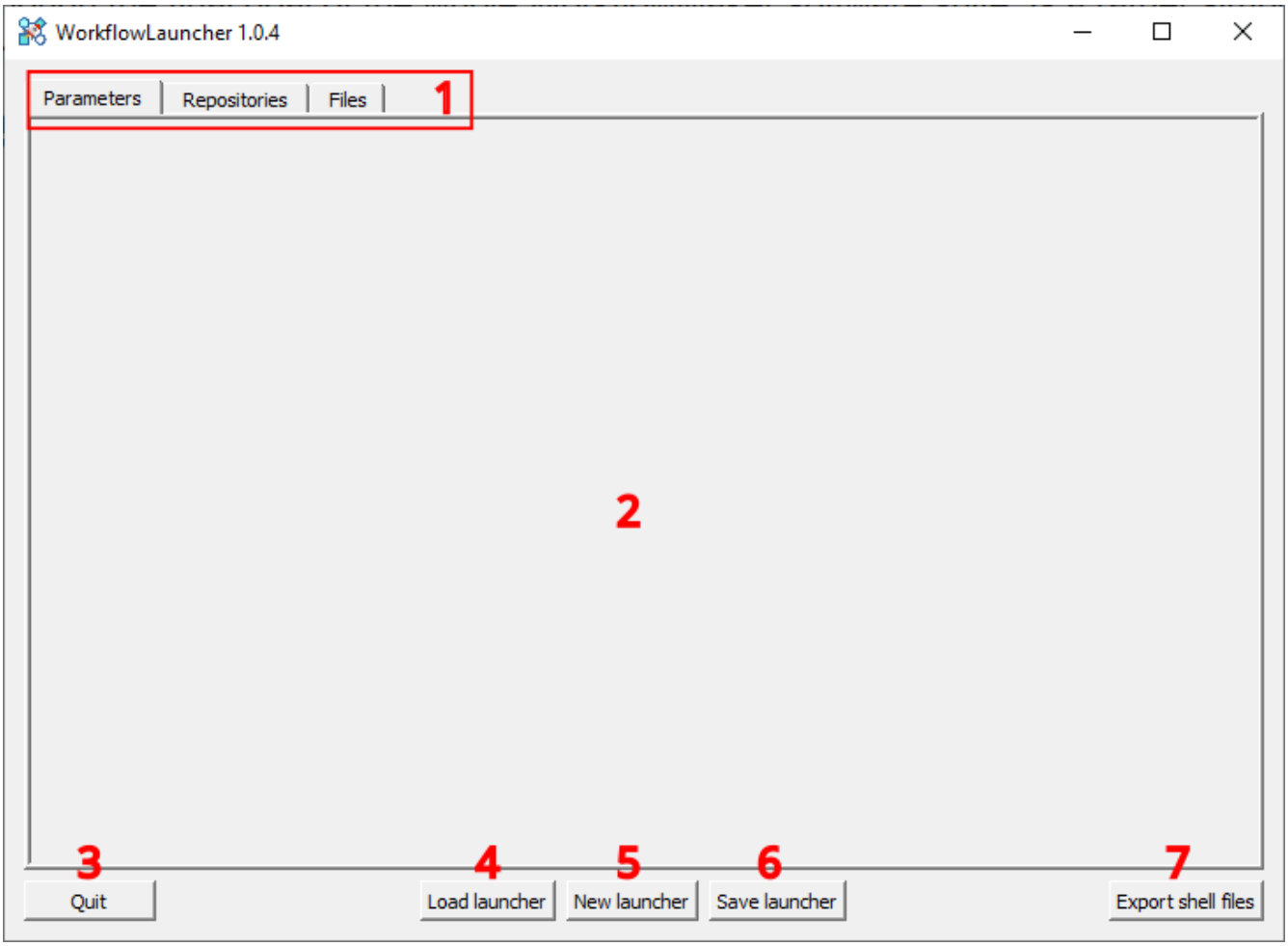

Figure 26: WorkflowLauncher: the Graphical User Interface (GUI).

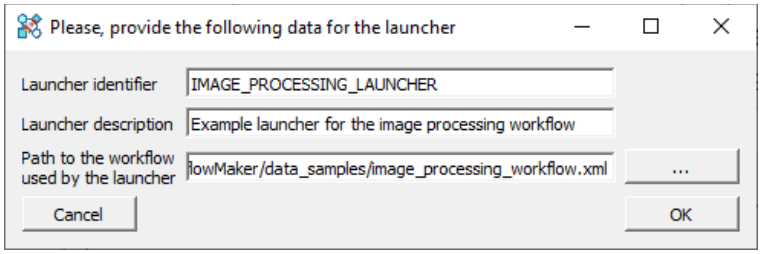

Figure 27: WorkflowLauncher: metadata for a new launcher file.

Once the “OK” button is clicked, the path to the toolkit on which the workflow relies will be requested too, as shown in Figure 28 (page 43). Note that the identifier of the said toolkit is clearly stated in this window.

Once the “OK” button is clicked, the process of creating a new (blank) launcher for the selected workflow and toolkit finishes, and the interface changes to offer the several items (parameters,

repositories and input / output files) that must be provided.

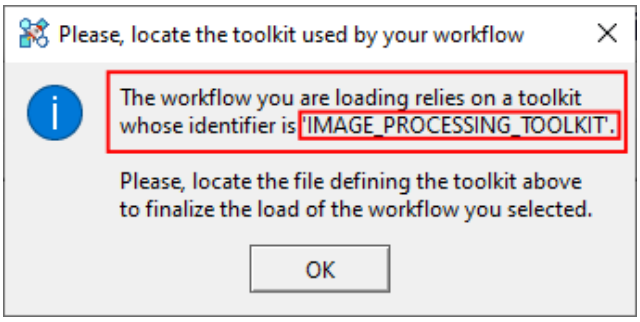

Figure 28: WorkflowEditor: finding the toolkit on which the launcher workflow relies.

In Figure 29 the “Parameters” tab of the WorkflowLauncher application is shown once that a new launcher file for the example workflow has been created using the procedure just described. Note the red squares: there is where users input the values of the parameters. A similar interface is shown when switching to the “Repositories” and “Files” tabs.

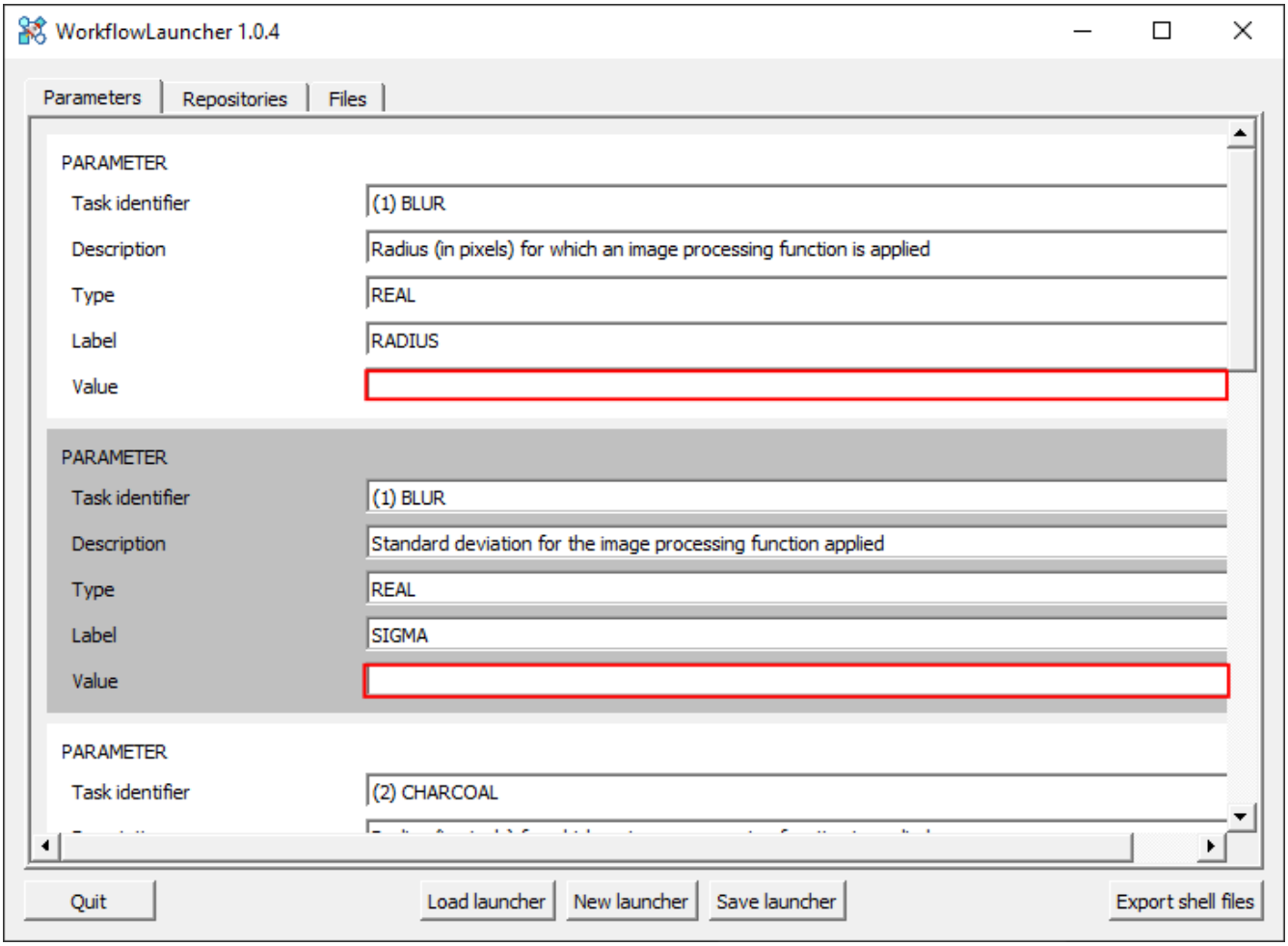

Figure 29: WorkflowLauncher: the "Parameters" tab just after creating a new launcher file.

3.5.3 Opening an existing launcher file

Opening an existing launcher file is quite similar to creating a new one.

To do it, click on the “Load launcher” button (number 4 in Figure 26). The typical file dialog to locate the file to open will be shown.

After selecting the desired launcher file, a window like the one similar to the one shown in Figure 30 stating that the workflow on which the launcher relies must be identified will pop up:

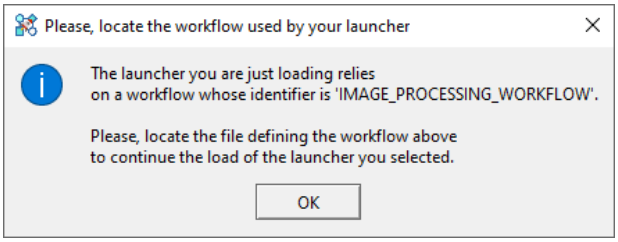

Figure 30: WorkflowLauncher: asking for the underlying workflow for an existing launcher.

Since a launcher relies on a workflow, it is necessary to locate it. As soon as the button “OK” is clicked, a new file dialog will be shown to select the workflow file. Then, again, a new window like the one shown in Figure 28 above (section 3.5.2) will appear, asking for the toolkit file on which the workflow selected is based. Clicking again on the “OK” button will make the third file dialog appear, being possible, finally, to select the toolkit sought.

Once this process is finished, WorkflowLauncher will show the launcher data, in a window similar to that of Figure 29 (again, section 3.5.2). In this case, and since an existing launcher file is being opened, the data cells (surrounded by red rectangles in that figure) show the values that were typed in there when this file was previously saved.

3.5.4 Values for the keyboard parameters

Providing the values for the keyboard parameters is simple. After selecting the “Parameters” tab, the whole list of parameters will show up.

All the information needed to properly identify each parameter is presented to the user, as shown in Figure 31 (page 45). In this figure:

1. *Task identifier.* a numeral plus the unique task identifier—(1) + BLUR in the example—enables the identification, within the workflow to which the launcher is associated, of the precise instance of the task for which the parameter needs to be typed. Consequently, if there were multiple instances of the same task within the workflow, there would be no ambiguity regarding which instance the parameter in question pertains to.

2. *Description.* The textual description given to the parameter when it was defined in the underlying toolkit.
3. *Type.* Data type (integer, real, string of characters) of the value to provide.
4. *Label.* The string that will be used to identify the parameter when creating the options file for this task instance.
5. *Value.* The value assigned to the parameter.

Note that fields 1 to 4 in the list above may not be modified by the user. Only the 5th field (the value of the parameter) may be provided.

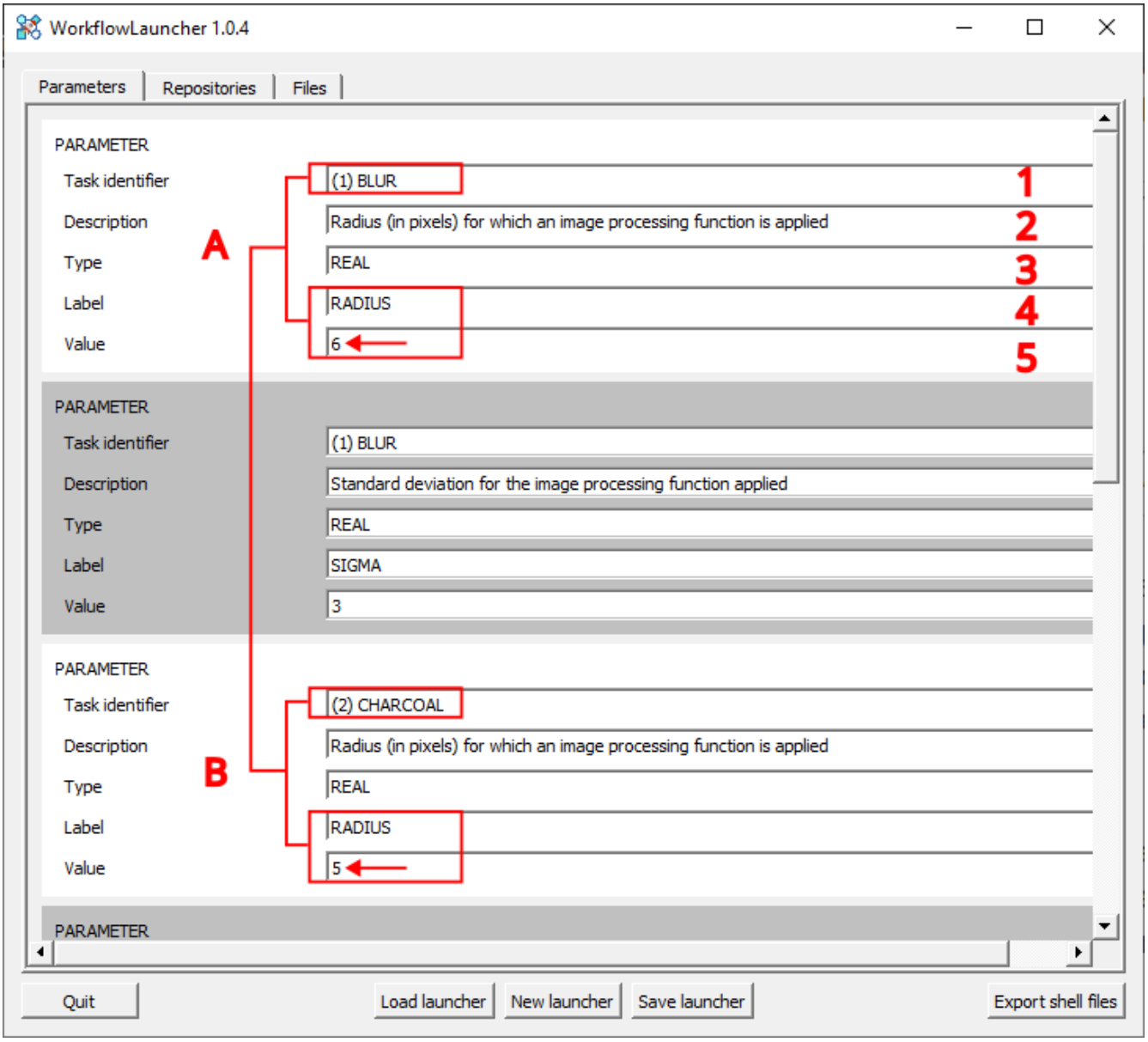

Figure 31: WorkflowLauncher: typing the keyboard parameters.

In this figure it may also be seen how *it is possible to assign different values to a parameter that is needed by more than one task*. The blocks A and B in the figure show how parameter RADIUS, required by tasks (1) BLUR and (2) CHARCOAL is set with two different values, 6 and 5 respectively.

3.5.5 Paths for the repositories

The procedure to assign an actual path to every repository involved in the workflow is very simple.

1. First, select the “Repositories” tab in the WorkflowLauncher application. The application shows the repository data. See Figure 32.
2. Then, and for each repository included in that tab, click on the button labeled “…” to open a folder selection dialog.

Note that the identifier of the repositories is included as it appears in the workflow in order to know which one is which.

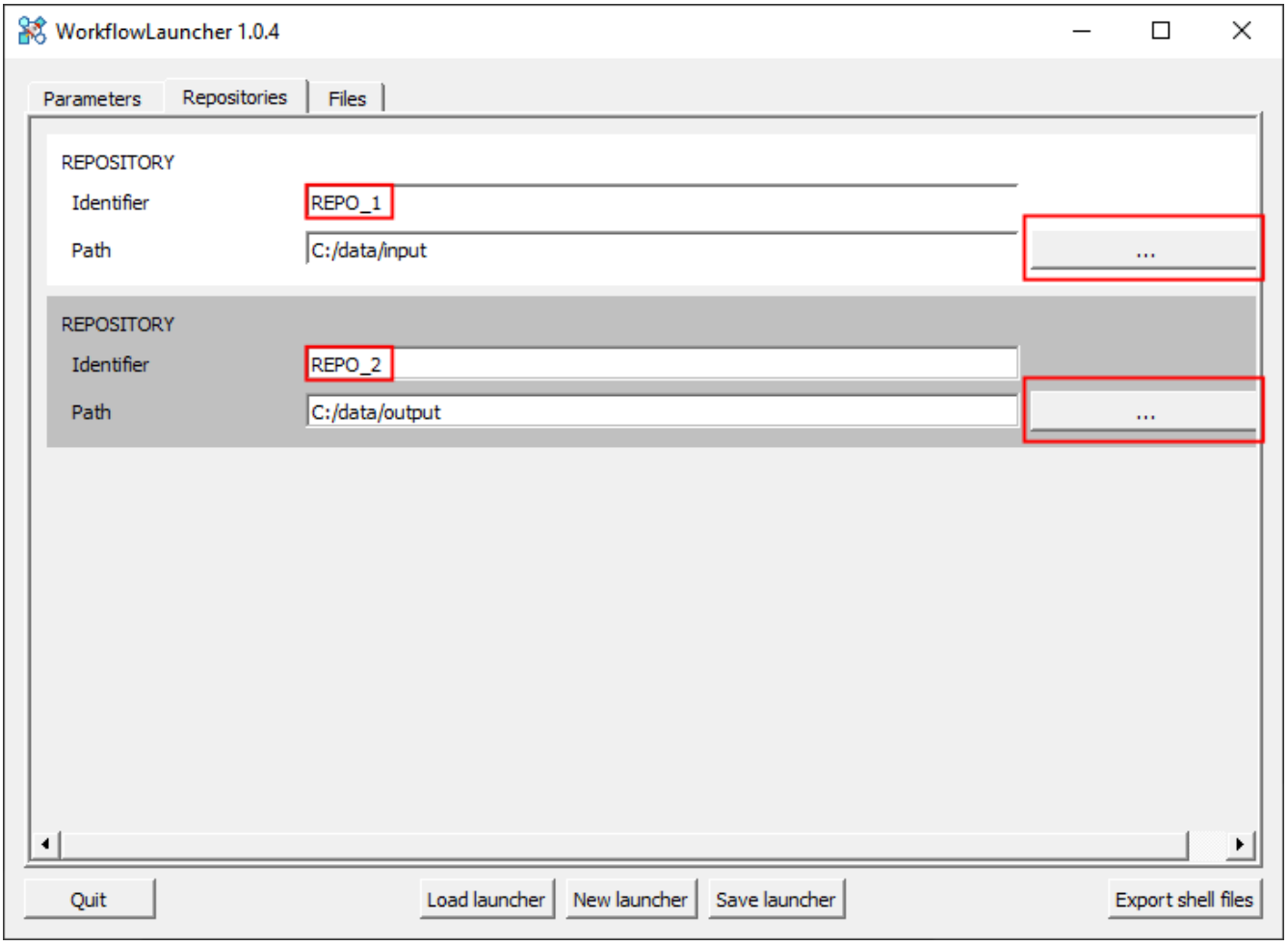

Figure 32: WorkflowLauncher: assigning actual path to repositories.

3.5.6 Give names to files

The last thing to do to complete the launcher is to provide names to the input and output files. It is very important to highlight that this involves only those files, either input or output, that reside (or will be saved) in a repository. *No names are needed for all the intermediate files created and read by tasks.*

This means that for the example workflow depicted in Figure 15, only two files need to be given a name: first, the input file for task “(10) PNG2JPG”, which is read from the repository named “REPO_1”; second, and last, the output of “(11) JPG2PNG”, which is saved to the repository whose name is “REPO_2”.

The remaining, intermediate files, are given names automatically when generating the script and option files (see section 3.5.8). These names are built as follows:

```
<task_identifier>_<task_numeral>_<output_slot_number>.<default_extension>
```

where:

- `<task_identifier>` refers to the unique identifier of the task producing the file.
- `<task_numeral>` is the number preceding the task identifier in the workflow.
- `<output_slot_number>` is the number of the output slot (starting at 0) producing the file.
- `<default_extension>` is the extension stated as default when the file type to which the file belongs was defined in the toolkit.

Thus, the name of the unique output file produced by task “(10) PNG2JPG” (see Figure 15) is, according to the naming rule above, `PNG2JPG_10_0.jpg`.

This naming mechanism guarantees that no intermediate files will have the same name.

It is very important to know how this naming mechanism works to avoid giving names to input or output files that might clash with that of the intermediate files. In short, users should avoid naming the input or output files (those residing in repositories) using the said naming schema.

In the image processing example, only two files need to be given a name. See Figure 33 (page 48). There:

1. Description. The role played by the file, as stated in the corresponding task when defining the toolkit. This is the *specific* description for the file.
2. Type. The combination of the file type identifier plus the description given to such file type. That is, this is the *generic* description of the file, as stated when defining the file type in the toolkit.
3. Usage. This field states what is the precise connection (either repository to task or vice-versa) to which the file corresponds. In this way, users may check the graphical depiction of the

workflow (such as the one shown in Figure 15, page 29) and be sure about what is the file for which a name is being given.

4. File name. the name given to the file. **Note that paths must not be included**, since these are derived from the repositories where files reside (input files) or will be stored (output files).

Note that in the “Usage” field slots are numbered starting from 1 and not from 0 (as in the naming policy for intermediate files). Furthermore, fields 1 to 3 in the list above and in Figure 33 are read only. Only field number 4 (the name of the file) may be input by users.

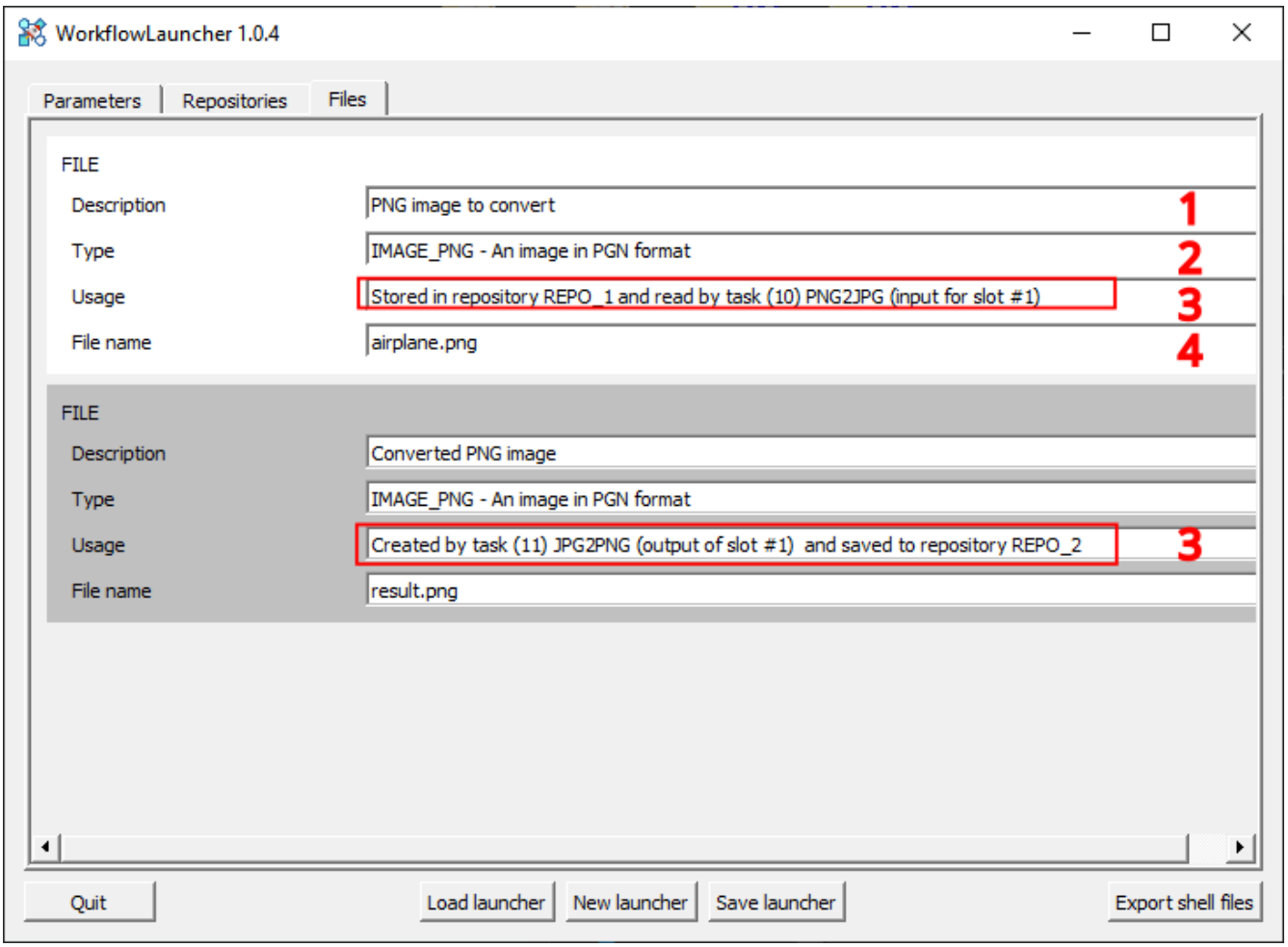

Figure 33: WorkflowLauncher: giving names to input and output files.

3.5.7 Saving the launcher file

Once all data has been input, it is possible to save the launcher file just by clicking on the button labeled “Save launcher”. A file dialog will show up to save the launcher file.

It is very convenient to save the launcher file before trying to generate the script (shell) and option files.

3.5.8 Generating the script (shell) files

When the launcher file is complete, either because the user has just saved it (section 3.5.7) or an existing launcher has been opened (section 3.5.3), it is possible to generate the scripts / shell files as well as the several option files required by the tasks included in the workflow.

This is the procedure to do it:

- Click on the “Export shell files” button.
- A window (Figure 34) will show up asking whether to save the current launcher file. If the answer is positive, then the classic file dialog will appear to save the said file.
- Then, another dialog will appear (Figure 35) to set the options for the generation of the scripts. In this figure:
  1. First, select the operating system for which the scripts will be generated by means of the dropdown menu provided for this purpose.
  2. Then, select the folder where the scripts and options files will be generated. It is recommended to select an empty directory to avoid merging the script and options files with any other files that might exist before.
  3. Finally, either confirm the operation or
  4. cancel it.

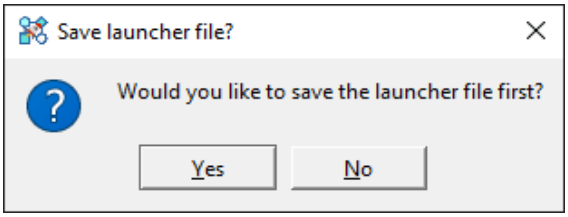

Figure 34: WorkflowLauncher: saving the launcher before creating the scripts.

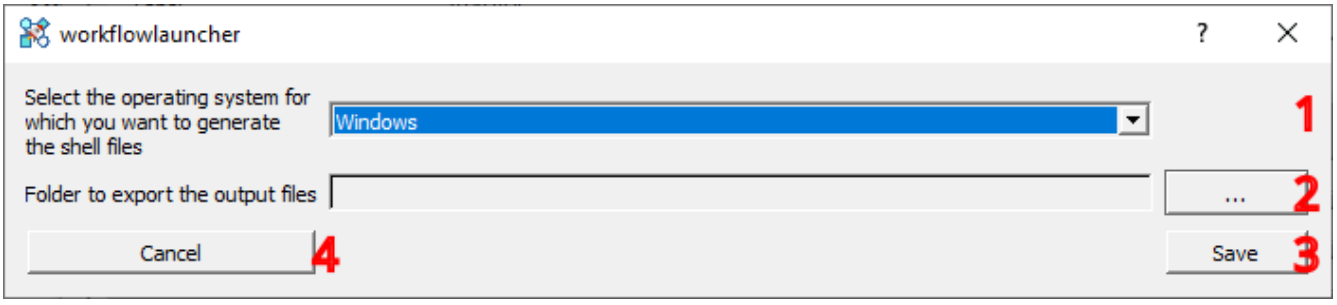

Figure 35: WorkflowLauncher: options to generate the script (shell) files.

If the user confirms the operation, the following set of files will be created in the folder just selected:

- `go.bat` (the script for windows, whose task is execute the workflow). In Linux, its name would be `go.sh`.

- As many option files as task instances exist in the workflow. In the case of the example workflow in Figure 15 (page 29) this means that 11 options file will be created. These are named according to the pattern `<task_identifier>_<task_numeral>.op`. Then, for the said example, the following option files will be created:

  - BLUR_1.op
  - CHARCOAL_2.op
  - GRAYSCALE_9.op
  - JPG2PNG_11.op
  - MIRROR_7.op
  - MOSAIC_4.op
  - MOSAIC_5.op
  - MOSAIC_6.op
  - OILPAINT_3.op
  - PNG2JPG_10.op
  - ROTATE_8.op

Never modify any of these files. They contain the appropriate information to process the data selected according to the workflow selected.

Figure 36 shows the contents of one of these files, CHARCOAL_2.op. Note how the labels used for both keyboard parameters and the input and output files are those stated in Table 1 on page 13.

```
INPUT_FILENAME  = C:/work_folder/PNG2JPG_10_0.jpg
OUTPUT_FILENAME = C:/work_folder/CHARCOAL_2_0.jpg
RADIUS          = 5
SIGMA           = 2
```

Figure 36: WorkflowLauncher: example of an automatically generated options file.

Note also that the names of the input and output files are built according to the naming rule described in section 3.5.6. The input file is the first output of task “(10) PNG2JPG”, therefore the name “PNG2JPG_10_0.jpg”. The unique output file of “(2) CHARCOAL” is named “CHARCOAL_2_0.jpg”, again obeying the aforementioned naming rule. Furthermore, *intermediate files are sought or created in the same directory (`C:/work_folder` in the example) where the launcher file was saved.*

Figure 37 (page 51) shows the contents of `go.bat`, the windows script file that must be run to execute the workflow processing the selected data set. There, the text in green shows how the several executables are run passing the corresponding options file as their unique parameter. Red is used to show how (and when) the intermediate files are deleted. Finally text in light blue denote error control instructions.

```
ECHO OFF

cd C:/work_folder/
@ECHO Running (10) PNG2JPG with options file PNG2JPG_10.op
png2jpg PNG2JPG_10.op
IF %ERRORLEVEL% NEQ 0 GOTO problems
@ECHO Running (1) BLUR with options file BLUR_1.op
blur BLUR_1.op
IF %ERRORLEVEL% NEQ 0 GOTO problems
@ECHO Running (2) CHARCOAL with options file CHARCOAL_2.op
charcoal CHARCOAL_2.op
IF %ERRORLEVEL% NEQ 0 GOTO problems
@ECHO Running (4) MOSAIC with options file MOSAIC_4.op
mosaic MOSAIC_4.op
IF %ERRORLEVEL% NEQ 0 GOTO problems
DEL C:\work_folder\BLUR_1_0.jpg
DEL C:\work_folder\CHARCOAL_2_0.jpg
@ECHO Running (3) OILPAINT with options file OILPAINT_3.op
oilpaint OILPAINT_3.op
IF %ERRORLEVEL% NEQ 0 GOTO problems
@ECHO Running (5) MOSAIC with options file MOSAIC_5.op
mosaic MOSAIC_5.op
IF %ERRORLEVEL% NEQ 0 GOTO problems
DEL C:\work_folder\MOSAIC_4_0.jpg
DEL C:\work_folder\OILPAINT_3_0.jpg
@ECHO Running (6) MOSAIC with options file MOSAIC_6.op
mosaic MOSAIC_6.op
IF %ERRORLEVEL% NEQ 0 GOTO problems
DEL C:\work_folder\MOSAIC_5_0.jpg
DEL C:\work_folder\PNG2JPG_10_0.jpg
@ECHO Running (7) MIRROR with options file MIRROR_7.op
mirror MIRROR_7.op
IF %ERRORLEVEL% NEQ 0 GOTO problems
DEL C:\work_folder\MOSAIC_6_0.jpg
@ECHO Running (8) ROTATE with options file ROTATE_8.op
rotate ROTATE_8.op
IF %ERRORLEVEL% NEQ 0 GOTO problems
DEL C:\work_folder\MIRROR_7_0.jpg
@ECHO Running (9) GRAYSCALE with options file GRAYSCALE_9.op
grayscale GRAYSCALE_9.op
IF %ERRORLEVEL% NEQ 0 GOTO problems
DEL C:\work_folder\ROTATE_8_0.jpg
@ECHO Running (11) JPG2PNG with options file JPG2PNG_11.op
jpg2png JPG2PNG_11.op
IF %ERRORLEVEL% NEQ 0 GOTO problems
DEL C:\work_folder\GRAYSCALE_9_0.jpg
EXIT /B
:problems
@ECHO An error occurred. Please check the logs for more information.
EXIT /B
```

Figure 37: WorkflowLauncher: the Windows shell file for the example in Figure 15.

The error control sentences ("`IF %ERRORLEVEL%`" in the case of Windows shell files) show how the status value returned by the console applications (see section 2.1) is used to detect problems when executing the workflow.

3.5.9 Running the workflow

To execute a workflow, follow the steps below:

1. Make sure that the input and output repositories exist. Additionally, the input files must reside in the corresponding input repositories. In the example, the paths to REPO_1 and REPO_1 are,

respectively, `C:/data/input` and `C:/data/output` (see Figure 32, page 46). The file named `airplane.png` (see Figure 33, page 48) must be placed in `C:/data/input`. The output file `result.png` (see again Figure 33) will be stored in `C:/data/output`.

2. Open a command line window / shell.
3. Change the default directory to that where the script and option files were saved.
4. Run either the `go.bat` or `go.sh` script.

On Linux computers it will be necessary to guarantee that the `go.sh` shell file has the execute privilege. To do it, run the command `chmod a+r go.sh` in the command line window once that the default directory has been changed to that where this file was created.

The `go.bat` (or `go.sh`) scripts **do not prepend any paths** to the names of the command line executables when running these. Therefore, it is necessary to guarantee that the said executables may be found by the operating system or the script will fail when run. The most easy option to do this in both operating systems is to add the folder where the executables reside to the PATH variable. The procedure to change such variable is different for Windows and Linux.

3.5.10 Reusing launcher files

Unlike ToolkitEditor and WorkflowEditor, it is possible to reuse a launcher file as many times as necessary. Therefore, it can be modified whenever desired, allowing the regeneration of new script and options files. This is the way to change the data set to use when running a workflow.

3.5.11 Running the example workflow. Results

Figure 38 on page 53 shows the input file `airplane.png`, which is the unique input file needed by the example workflow to run. Figure 39 (also in page 53) depicts the result of such processing. Please, review both section 3.4.1 and Figure 15 to check that such result corresponds to what it was expected. In short, the steps to execute where:

- Change the format of the input image from PNG to JPG
- Create a mosaic including, from left to right, a blurred version of the input image, another to which the charcoal effect has been applied, one more with the oil paint effect and, finally, the original one.
- Then the previous mosaic is mirrored (around a vertical axis), rotated 90 degrees clockwise and converted to gray scale.
- Finally, the image is converted once more, this time from JPG to PNG.

Note that the size of the result image is not the right one. It has been reduced to make it fit in the page. This is important to highlight, since no resizing operations were included in the example workflow.

Figure 38: WorkflowLauncher: the unique input image for the example workflow.

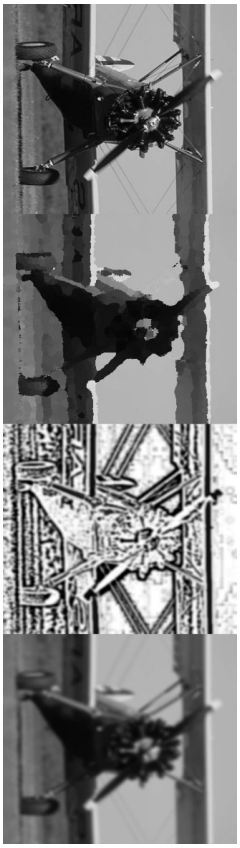

Figure 39: WorkflowLauncher: the result of running the example workflow.