Martin Luther University Halle-Wittenberg

Institute of Computer Science

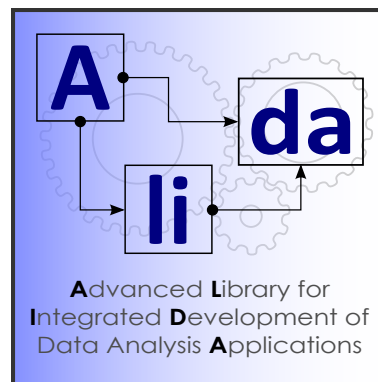Pattern Recognition and Bioinformatics

User and Programmer Guide

– Alida –

Advanced Library for Integrated Development
of Data Analysis Applications

Version 3.0, May 2020

written by

The Alida Development Team

Birgit Möller   Stefan Posch

Licensing information.

This manual is part of `Alida`, an
Advanced Library for Integrated Development of Data Analysis Applications.

Copyright © 2010–2020

This program is free software: you can redistribute it and/or modify it under the terms of the
GNU General Public License version 3[1] as published by the Free Software Foundation[2], either
version 3 of the License, or (at your option) any later version.
You should have received a copy of the GNU General Public License along with this manual.
If not, see http://www.gnu.org/licenses/.

Fore more information on `Alida` visit https://alida.informatik.uni-halle.de.

`Alida` **is a project at the Martin Luther University Halle-Wittenberg.**

Institution:

Institute of Computer Science
Faculty of Natural Sciences III
Martin Luther University Halle-Wittenberg
Von-Seckendorff-Platz 1, 06120 Halle, Germany

Contact: alida@informatik.uni-halle.de
Webpage: https://alida.informatik.uni-halle.de

---

[1]http://www.gnu.org/licenses/gpl-3.0.html
[2]http://www.fsf.org/

Contents

1   Introduction

`Alida`, which is the acronym for *Advanced Library for Integrated Development of Data Analysis Applications*, is the name of our integrated concept to ease the development and application of data analysis algorithms. For use in practice the concept is implemented in terms of a software library. This `Alida` library on the one hand allows for an automatic generation of generic user interfaces for implemented algorithms, and on the other hand subsumes the fully automatic documentation of data analysis processes performed using functionality from the library. The underlying core of the `Alida` concept is given by an interpretation of data analysis processes as a sequence of data manipulations solely performed by functional units, called *operators* in `Alida`. Given a generic framework for the implementation of operators in `Alida`, these can be handled, i.e. configured and executed, in a standardized manner. This results in a wide range of useful features for programmers as well as users.

The operator concept, with the operators as the central places of data manipulation and a unified invocation procedure, allows to monitor all data manipulations taking place during a data analysis process. Additionally, all objects ever manipulated are registered within the system and can be linked to their manipulating operators. An automatic documention of analysis procedures is supposed to subsume all input and output objects involved in the execution of operators, all manipulations performed with all their relevant parameters, the flow of data, and also software versions as used. All this information is summarized in the *processing graph*, which is implicitly defined by any data analysis process. As manipulative actions can either work sequentially or in parallel on data items, the processing graph is given by an acyclic directed graph. Its nodes are associated with the different operations applied to the data, and the edges in between represent the operators' ordering and the overall flow of data and control. This representation is shown in Fig. 1 for an example graph. As `Alida` allows to collect all relevant data for extracting processing graphs due to its operator concept and the standardized invocation procedure for operators, it allows to make the processing graph explicit without additional efforts to the programmer or user. In particular, for each output object of any analysis process the processing graph can subsequently be made explicit in terms of an XML representation. This allows for convenient visualization, reconstruction and verification of results at a later point in time, and also for long-term archiving, e.g., in databases.

While documenting analysis processes is helpful for verifying or reconstructing results at later points in time, i.e., for long-term consistency and preservation of data analysis outcomes, another important aspect of algorithm development is the accessibility of algorithms and tools for programmers and users. Usually neither developers of algorithms nor users are willing or able to spend much time on the implementation of user interfaces. Likewise a proper documentation for users covering the functions included in a library or toolbox and their usage is often neglected. While for API documentation of a library tools like `javadoc` are well-established to support developers with this task, less technical user documentation lacks such a support.
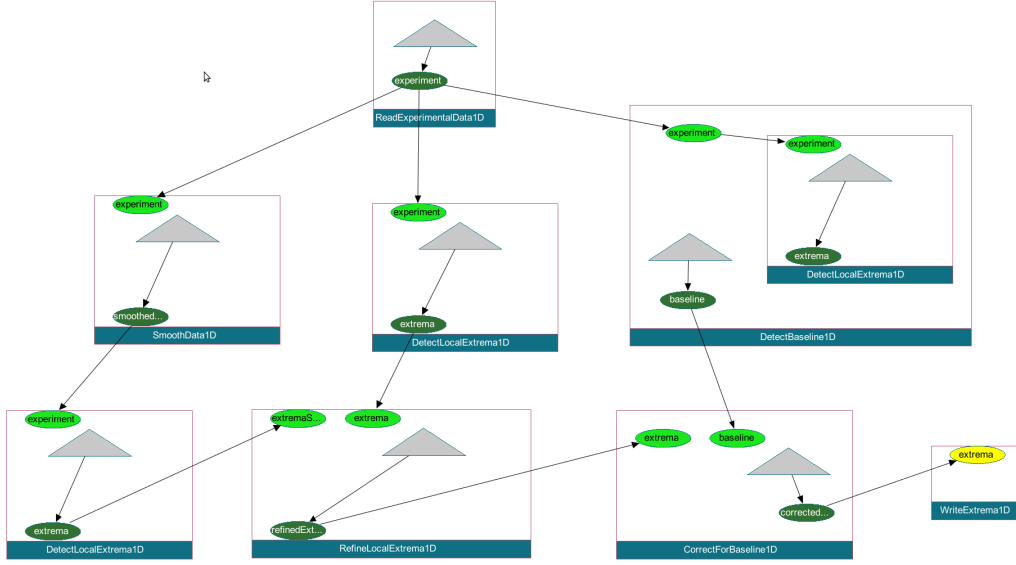
Figure 1: Example processing graph representing the history of operations for producing the data object shown as yellow ellipse. Each operator invocation is respresented by a blue or, if the operator is temporarily collapsed, to a violet rectangle. Light and dark green ellipses are input and output ports of operators, gray triangles represent newly generated data objects.

`Alida`'s operator concept and the standardized configuration and execution procedures also offer solutions to these problems. On the one hand `Alida` supports the automatic generation of user interfaces for operators, on the other hand its interfaces render it easy to programmatically integrate a basic user documentation for each operator. More specifically, `Alida`'s concept yields a suitable fundament for automatically generating graphical and command line user interfaces which allow to configure and execute all operators implemented in the `Alida` framework, provide a basic documentation of each operator's functionality to the user in terms of an online help, and offer options to inspect the results of processing. While a graphical interface mainly targets at immediate visual inspection of operator configurations and results, a command line interface is useful, e.g., for parameter tuning or batch processing facilitated by scripts.

The framework for documentation and user interface generation is independent of programming languages. `Alida` currently features a mature implementation in Java. This implementation is shipped with a command line tool for running operators from command line, and with a graphical user interface based on Java Swing, which supports comfortable parameter editing and result data inspection in a graphical framework, well-suited also for non-expert users. In addition, the graphical editor `Grappa` is included which currently supports the composition of flat workflows of operators and their partial or complete execution (see Sec. 2.4). For convenience the configuration of operators, their execution and the inspection of results are identical to the corresponding procedures in the graphical user interface. To inspect the automatically generated XML documentation, the graph visualization tool `Chipory` is available (see Appendix A).

In general, the `Alida` concept enforces minimal restrictions for users and programmers,

interferring as little as possible with usual software development cycles, and resulting in an automatic process documentation with a minimum of overhead. `Alida`'s concept is applicable to any data analysis process.

Download of source code and binary bundels as well as installation instructions are available on `Alida`'s web-page, `https://alida.informatik.uni-halle.de`.

For developing own software based on the `Alida` library we operate a Maven repository server from where `Alida` artefacts can directly be included in your own code: `https://moon.informatik.uni-halle.de/archiva/#browse/de.unihalle.informatik.Alida`

2 The User's View

2.1 Quick starter

After installation, it is easy to explore the user interfaces automatically generated by `Alida` using the shipped demo operators.

The generic graphical user interface may be invoked using the application `ALDOpRunnerGUI` in the package `de.unihalle.informatik.Alida.tools`. It may be started from your favorite IDE or from the command line by

java de.unihalle.informatik.Alida.tools.ALDOpRunnerGUI

as soon as the `CLASSPATH` is set correctly, i.e., it needs to include Alida's jar archive and all its dependencies. Running the above command will bring up a window displaying a package tree of all available operators, which are mainly given by the set of demo operators shipped with `Alida` (see Fig. 2). You may unfold the demo package and select the operator `ALDCalcMeanArray`, which is to compute the mean of a set of numbers in a 1D array.

Once you choose to configure the operator by double-clicking on its entry, a configuration frame pops up (Fig. 3). The operator configuration pane lists the parameters of this operator, which are separated into required, optional and supplemental parameters (see Sec. 2.2 for more details on the parameters of operators). If you hover over the parameter `'Compute mean free data'`, the tooltip displays the data type of the parameter and a descriptive text. Set this parameter to `true` by checking the box associated with the parameter. The input data may be supplied selecting `'Configure Native Array...'`, which in the newly created window (Fig. 3, window bottom left) allows to add or delete elements to or from the array, and to enter values. Once you opened the configuration frame and, by this, instantiated an array, the run button in the operator configuration frame turns to yellow to indicate that all required parameters of `ALDCalcMeanArray` are properly set and the operator is ready for execution. After completion, a result frame will show up displaying the resulting mean value. As the parameter `'Compute mean free data'` is declared as an `INOUT` parameter of the operator, its value is displayed in this result frame as well. Furthermore, the window also provides a button to once again inspect the operator configuration used to calculate the displayed results.

The operator `ALDCalcMeanArray` can also be invoked from command line, without running the graphical user interface, via the application `ALDOpRunner`. It expects the values of all parameters being handed over as command line argument strings. The operator can be executed as follows:

java de.unihalle.informatik.Alida.tools.ALDOpRunner ALDCalcMeanArray \
    data=' [1.0,1.5,2.2,0.4] ' doMeanFree=**true** mean=− meanFreeData=−
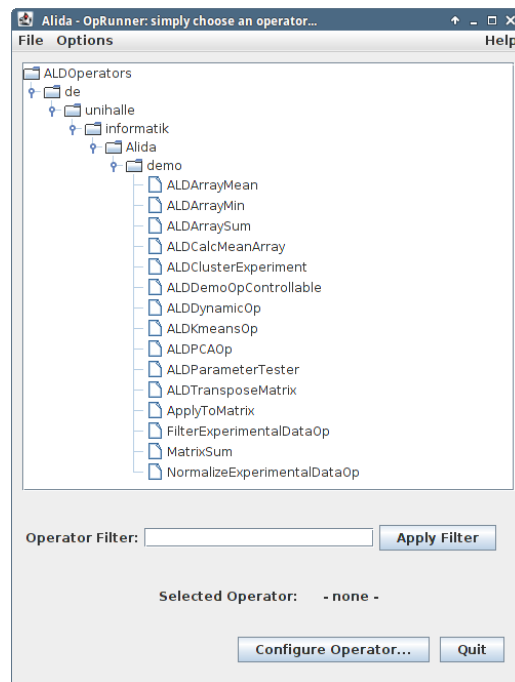
Figure 2: Screenshot of the main window of `Alida`'s operator runner with the `demo` package unfolded.
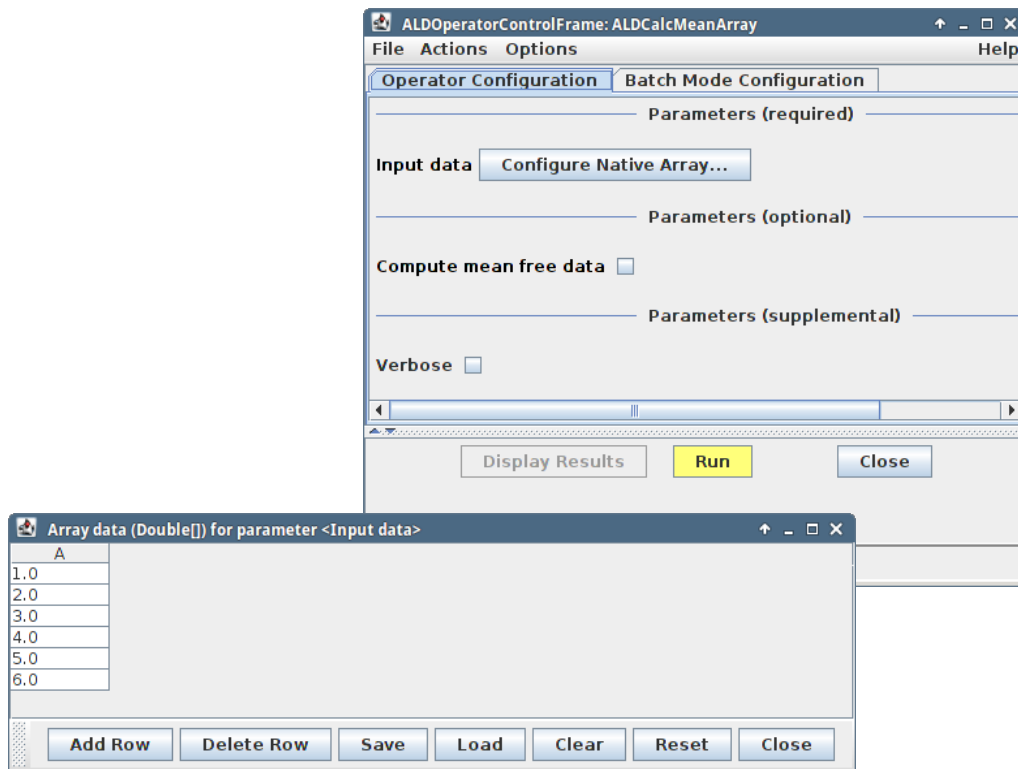

Figure 3: Screenshot of the automatically generated control window for the operator `ALDCalcMeanArray`.

The call will execute the operator with a 1D array of four double values as input data, and requests to compute the mean free data array in addition to the mean of the data. Setting the output parameters 'mean' and 'meanFreeData' to '-' requests to print the results to the standard output, yielding the following output in the console:

meanFreeData = [-0.2750000000000013 , 0.22499999999999987 ,
0.925 , -0.8750000000000001]
doMeanFree = **true**
mean = 1.2750000000000001

2.2   `Alida` operators

The heart of `Alida`'s concept are operators that implement all data analysis capabilities. Operators are the only places where data are processed and manipulated. Examples for data to be manipulated are, e.g., experimental measurements, sets of DNA sequences, or, for image analysis, images and sets of regions comprising a segmentation result. An operator receives zero or more input objects comprising all input data the operator is expected to work on. Operators with zero inputs are operators which for example create a data object for given parameters or read data from file. Further input to an operator are parameters, which configure or modify the processing on the input data. Examples are the selection of alternative processing procedures, e.g., if experimental measurements should be summarized by their mean or their median, a mask-size of a filter to be applied to an image, or the maximal number of iterations for a gradient descent algorithm. The distinction of an input acting as input data or as a parameter is not clear in all cases. As an abstract example consider an operator which is to compute the scalar product of two vectors. In this case, both vectors are most likely considered as input data. However, if the operator is to normalize a data vector by a scalar normalizing constant, this scaling factor may either be considered as an input or a parameter. Therefore, `Alida` does not distinguish between input data and input parameters. However, parameters of an operator may be optional, required or supplemental.

An operator produces zero or more output objects as the result of processing. An operator with zero output objects will, e.g., write data to disk.

All input and output data are denoted as *parameters* in `Alida`. The role of a parameter is identified by its direction, which may be input (`IN`) or output (`OUT`). In cases, where an input object should just be passed through the operator or is destructively modified, this parameter has the direction input and output (`INOUT`). An example is a vector which is modified in place.

In addition to parameters providing the input data and configuration of the operator, and output parameters representing the results of processing, an operator may use supplemental parameters. By definition, the setting of supplemental parameters must not influence the data

processing nor the results returned as output data. Consequently, supplemental parameters are not documented in the processing history. Examples for such parameters include flags to control output or debugging information, and intermediate results produced by an operator.

The relevant features defined for parameters in `Alida` are the following:

- the direction of the parameter, which may be `IN,` `OUT` or `INOUT`,
- a boolean indicating whether the parameter is supplemental,
- a boolean indicating whether the parameter is required or optional (which is only interpreted for non-supplemental `IN and INOUT` parameters)
- a label, e.g., used in the graphical user interfaces,
- a textual explanation of the parameter, for example appearing in tooltips,
- a data I/O order by which parameters can be ranked for generic GUI or command line interface generation, and
- an expert mode which, e.g., allows to hide parameters for advanced configuration from non-expert users.
- the name of a callback function which is called if the parameter's value is changed using the `setParameter()` method of the operator; the function can, e.g., be used to automatically update other parameter values or to even modify the set of parameters of the operator (refer to Sec. 3.1.2 for details)

The application of operators may be nested as one operator may call one or more other operators. At the top of this hierarchy we typically have appropriate user interfaces. Their parameter settings are facilitated via files, GUIs, command line, or via the console.

2.3   Graphical user interface

Easy and prompt access to new and improved data analysis algorithms is essential in many fields of application where the development of data analysis algorithms and progress on the application side are deeply linked to each other. `Alida` meets these requirements by providing a mechanism to automatically generate handy graphical user interfaces (GUI) for all operators implemented within its framework.

2.3.1   Graphical operator runner

The operator GUIs can easily be invoked from `Alida`'s graphical operator runner, named `ALDOpRunnerGUI`, to be found in the package `de.unihalle.informatik.Alida.tools`. Upon invocation the main window of the operator runner is shown, from where operators can be selected for execution. A screenshot of the main window is displayed in Fig. 2.

The main part of the window is formed by the tree view of all available operators, hierarchically arranged due to their package structure. From this view the operators to be executed can be chosen. An operator can be invoked by either double-clicking on its item in the tree, or by selecting the entry with a single mouse-click and then pressing the `'Configure Operator...'` button at the bottom of the window. Note that the tree view allows to configure the set of initially unfolded packages, i.e., visible operators. To this end the user needs to provide a file with the set of his or her favorite operators, and set a related environment variable to the name of that file (see Sec. 2.7 for details). In addition, the operator runner provides a search and filter function, accessible via the entry field and filter button in the bottom part of the window. For finding operators with specific names, type in the name of the operator or a substring of its name, then press the button or hit the return key. Subsequently the tree will only contain the operators matching the filter string. Note that the filter function is not case-sensitive.

To further ease operator selection and improve usability of the operator runner, `Alida` supports two different categories of operators. The first category of operators is mainly dedicated to non-expert users and often targets at concrete applications, the second set additionally subsumes more sophisticated operators often being very specialized and primarily intended to be used by experts. The tree view of the operator runner window allows to switch between the first set and a view showing all available operators by choosing the categories via the item `'Options'` followed by `'Operators to Show'` in the window's menubar. By selecting `'Default'` the tree view of operators is restricted to operators of the first category, while selecting `'All'` renders all available operators to be displayed. Note that the menubar also grants access to `Alida`'s online help and the current version of `Alida` being used.

2.3.2   Operator control window

Once an operator has been selected, the corresponding operator control window pops up (Fig. 4). It allows for configuration and execution of the operator. The window is subdivided into four parts, i.e., it consists of a menubar, a control section with a set of buttons at the bottom, a status bar, and the configuration section with different tabs typically occupying the largest fraction of the window.

Configuration section.   For most operators there are two tabs available in the configuration section. The first one, denoted `'Operator Configuration'`, subsumes graphical elements for handy configuration of the operator's parameters. As operator parameters may have different data types, and each data type requires individual I/O handling, with each data type a specific graphical element is associated. For example, for inserting values for native parameters of type `int` or `double` a simple text field is displayed, while for arrays and collections buttons are shown which allow to open editable tables. The demo operator `MatrixSum` defines the two input parameters `'Input matrix'` of type `Double[][]` and `'Summarize mode'`. The latter one is linked to an enumeration class, and a corresponding combobox for selecting one of the available
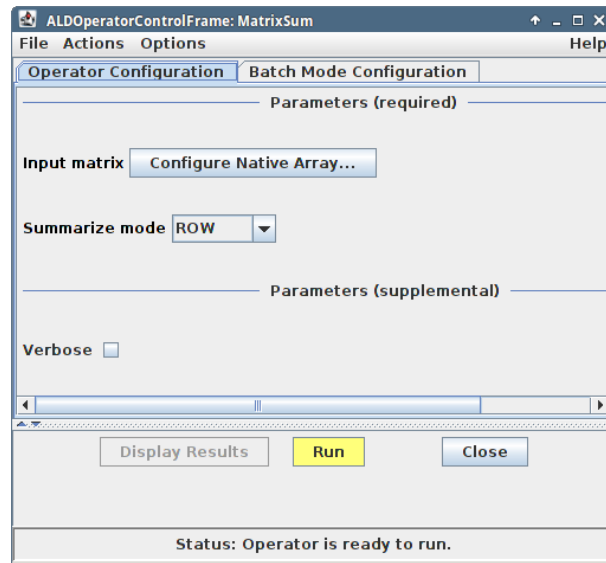
Figure 4: Automatically generated control window for the `Alida` demo operator `MatrixSum`.

enumeration elements is shown. As can be seen from the screenshot in Fig. 4, the configuration section is further subdivided into required and supplemental parameters. Also optional parameters may appear there, however, the demo operator in this example does not define any optional parameters, thus, the corresponding section is missing in this case.

The second tab, denoted `'Batch Mode Configuration'`, provides access to `Alida`'s built-in support for batch processing. The basic idea of the batch mode is to automatically execute an operator multiple times with different input values for a certain parameter. Consequently, on the tab a single input parameter of the operator can be selected and configured for batch processing. In Fig. 5 the tab for batch mode configuration of the operator `ALDArrayMean` is depicted. The operator expects as input an array of type `Double[]`. After activating the batch mode via the corresponding checkbox, it is possible to configure this parameter. In this case the user has to provide an array of type `Double[][]` as input for the operator[3] (see Fig. 5).

During batch processing the operator is run multiple times, each time processing a single row of the input array (also refer to subsequent paragraph). The result of such a batch procedure is given by a summary of the values of selected operator output parameters. The parameters of interest have to be selected on the batch tab as well, and upon termination of the batch run the values of the different runs are appropriately summarized (Fig. 6). Note that not all operators allow batch processing. For these operators the batch mode tab is not shown. The batch mode support in `Alida` is in an early state, and currently there is batch support only for very few input and output parameter data types.

---

[3]Note that if the batch mode is activated for a certain parameter, it is not possible to configure the parameter via the 'Operator Configuration' tab.
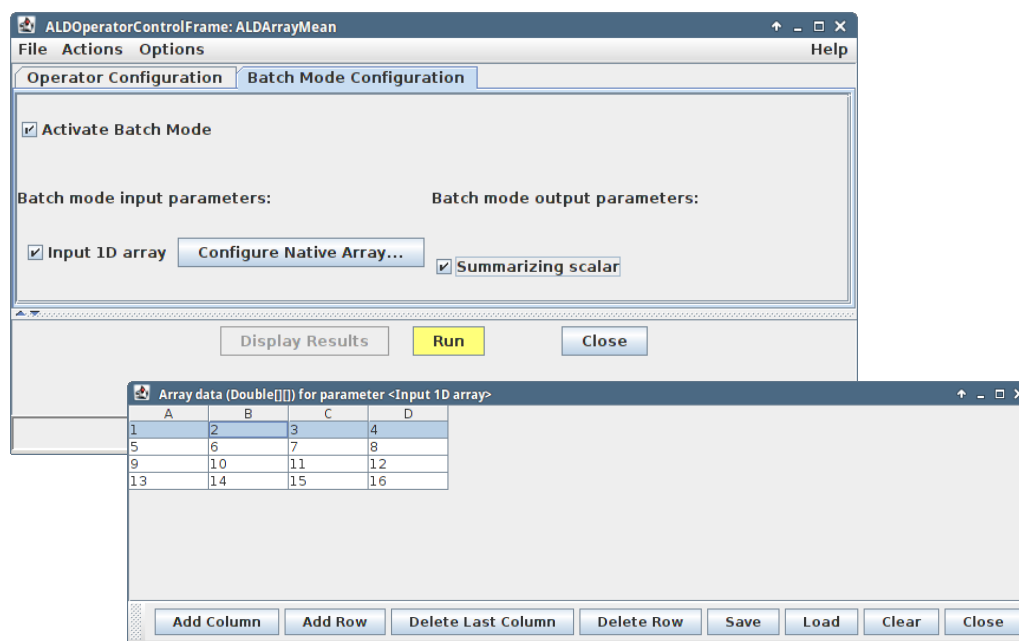
Figure 5: Screenshot of the tab for batch configuration of the operator `ALDArrayMean`. As the operator expects as input a 1D array of type `Double[]` the user has to provide an array of type `Double[][]` as batch mode input via an appropriate configuration window (visible at the bottom of this figure).
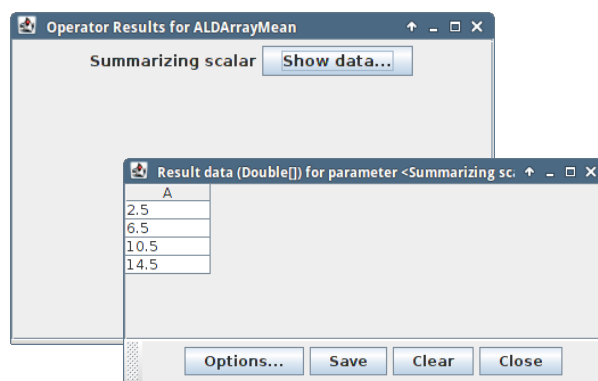
Figure 6: Window summarizing the batch processing results for the operator `ALDArrayMean`. Each entry of the result array refers to the mean of the elements in one row of the batch mode input array (Fig. 5).

Control section.   If an operator has been properly configured, either for normal execution or for batch processing, it can be invoked from the buttons in the control section of the window. By default, this section contains a button labeled `'Close'` to close the operator control window, a button `'Display Results'` (Fig. 5), however, which is deactivated until results are actually available, and of course a button labeled `'Run'` to run the operator. Once the run button is colored yellow, the operator is ready for execution. If it has red color, the configuration is not yet completed. A green color indicates that the operator was already executed with the given set of parameters and cannot be run again unless the configuration is changed. When the run button is clicked, `Alida` executes the operator. While the operator is running the button takes a blue
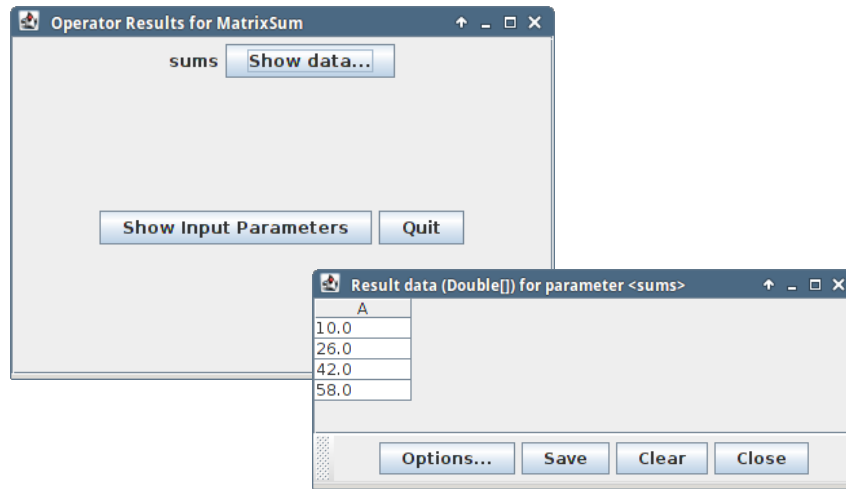
Figure 7: Result window for the demo operator `MatrixSum` as displayed upon termination. On the left, the actual result window is shown, while on the right the result for the operator's output parameter '`sums`' is displayed, which here is the set of row-wise sums of the two-dimensional input array. This window pops up by clicking the '`Show data...`' button in the result frame.

color. Upon termination its color switches to green, and a result frame is displayed summarizing the results and also providing direct access to the input parameters (Fig. 7).

Besides generic execution of operators in terms of invoking the operator and displaying its results, `Alida` also supports interactive operator execution. Interactive operators allow for user interaction in terms of at least pausing, resuming or interrupting the data processing. Additionally, interactive operators may also support step-wise data processing where the operator is automatically paused after a specified number of steps. The notion of a 'step' in this case is left to the programmer of the individual operator and may vary between operators. In most cases, however, step-wise execution will be supported by interactive operators performing several iterations during execution. Each iteration will then be associated with one execution step. For interactive operators additional control elements are displayed in the control section of the operator control window (Fig 8).

For all interactive operators two additional buttons labeled '`Pause`' and '`Stop`' are displayed. The execution of an operator has always to be invoked pressing the '`Run`' button. If the '`Pause`' button is pressed while the execution is ongoing, the operator will be paused. Note, however, that this does not imply that the operator immediately interrupts its execution, but rather proceeds until reaching the next *breakpoint* predefined by the programmer. It may even happen that the operator terminates its execution instead of just pausing it. The latter will be the case if the last breakpoint had been past before the pause command was triggered by the user. Common breakpoints of operators are for example the end of an iteration in iterative procedures or the termination of subroutines. In general, however, the definition of a breakpoint is left to the programmer of an operator and varies between operators. Pressing the '`Stop`' button during execution triggers a similar behaviour. The operator will proceed with its execution until
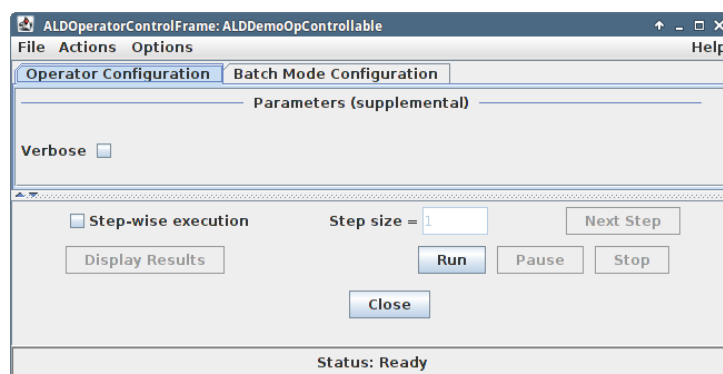
Figure 8: Control window for an operator supporting interactive execution. Note the set of additional control elements at the bottom of the window.

the next predefined *termination point* is reached, then does some clean-up and finally presents intermediate results to the user. Termination points may be identical to breakpoints, however, this is not mandatory.

If an interactive operator also supports step-wise execution the control windows displays additional graphical elements to activate the step-wise mode, specify a step size and to trigger execution of subsequent steps. After activating 'Step-wise execution' and specifying the step size such an operator can be invoked by pressing the 'Next Step' button. The operator then starts its execution and will automatically pause after the given number of steps. Subsequently the user needs to resume execution by clicking the button 'Next Step' again. During step-wise execution the operator can also directly be terminated by pressing the 'Stop' button.

Menubar.   The menubar of the operator control window allows for additional actions. From the menu item 'File' it is possible to save the current operator configuration to a file on disk in XML format, and also to load a configuration from such a file. Note that the batch mode configuration is currently not included in this file. The 'Actions' menu offers the possibility to reset the operator parameters to their default values, and the item 'Options' provides access to configuration options for the appearance of the operator control window, e.g., to switch between different modes of parameter display and to configure the status bar.

Besides defining required, optional and supplemental parameters, Alida supports different modes for an operator parameter. In detail, a parameter can be declared as STANDARD or ADVANCED. While standard parameters are assumed to be the most important parameters of an operator, advanced parameters allow for more sophisticated configuration, however, can most of the time be ignored by non-expert users. Consequently, the parameter view can configured to show all parameters of an operator or to just display the default set of most important parameters which is the default setting.

The item 'Help' in the menubar again grants access to Alida's online help.

Status bar. The status bar at the bottom of each operator control frame displays the current status of the operator. It tells the user if the operator is unconfigured, readily configured to be executed, or if it has been executed and result data is available. In addition, if the underlying operator is triggering progress events during execution (see Sec. 3.1.3 for more details), these can also be displayed in the status bar. The item 'Show Progress Messages' in the 'Options' menu of the menubar allows to enable or disable the display of these messages.

2.4   Graphical workflow editor: Grappa

Most of the time complex data analysis tasks cannot be solved by only applying a single operator to the data. Rather, selections of various operators need to be combined into more sophisticated *workflows* to extract desired result data. Alida inherently supports the development of such workflows. On the programmatic level it provides extensions of the operator concept towards workflow objects, and on the user side it includes *Grappa, the **Grap**hical **P**rogramming Editor for **A**lida*. Grappa allows for designing and manipulating workflows via graph edit operations, hence, offers an intuitive interface and large flexibility for developing workflows.

A workflow in Alida is defined as a graph data structure. Each node of the graph represents an Alida operator, while edges between different nodes encode the flow of data and control. Each node owns a selection of input and output ports which are associated with the operator's parameters. Consequently, edges are directed, i.e., an edge always connects an output port of one operator node with an input port of another. Grappa visualizes such workflow graphs and supports manual editing, manipulation, and also workflow execution and analysis of results.

Grappa can be started using the following command:

```
java de.unihalle.informatik.Alida.tools.ALDGrappaRunner
```

Figure 9 shows a screenshot of Grappa's main window. It is basically divided into two sections. On the left, the node selection menu is visible, while on the right the workbench area is located. In addition, the window features a menubar for configuring Grappa, loading and saving workflows, and accessing the online help. At the bottom of the window a panel displaying status and progress messages is available.

2.4.1   Operator node selection menu

In the selection menu on the left of Grappa's main window all Alida operators found in the classpath upon initialization are listed as potential nodes for Grappa workflows. In analogy to the graphical user interface (see Sec. 2.3) they are arranged in a hierarchical ordering according to their package structure. The different package subtrees can be folded and unfolded by double-clicking on a folder's name in the selection tree, or by single-clicking on the circle displayed left to
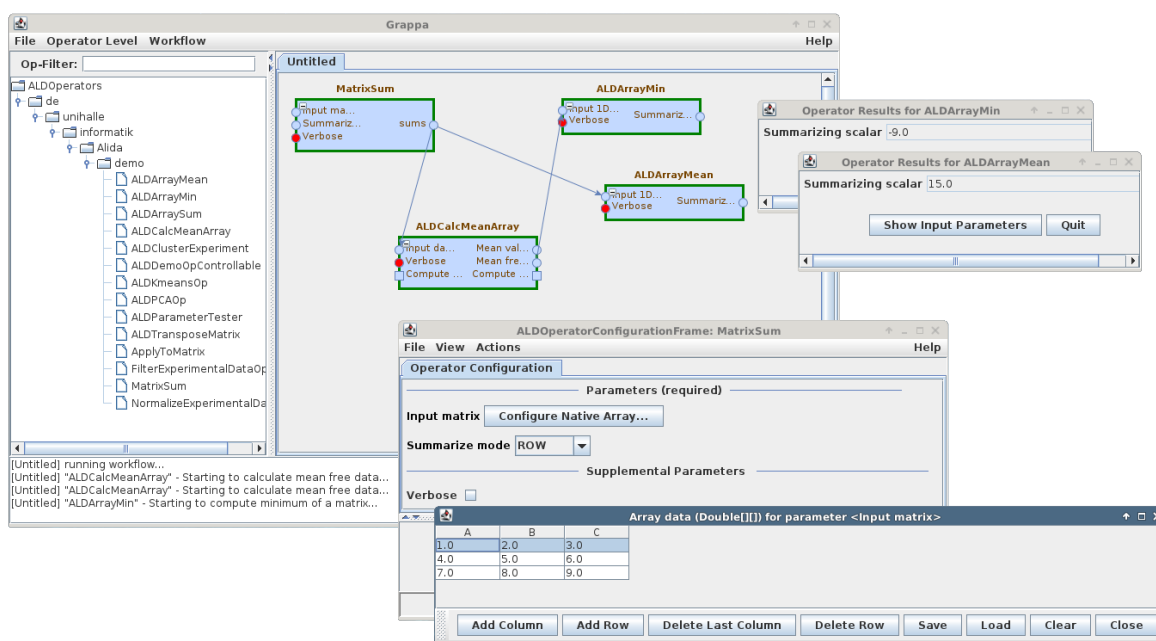
Figure 9: Screenshot of the graphical editor *Grappa*. In addition to the editor's main window (top left) two configuration windows for data input (bottom) and two operator result frames (top right) are displayed.

the folder icon. Above the tree view an operator filter is available which allows to select operators according to their names. For filtering, enter a substring into the text field and press the return key. As for the graphical user interface, `Alida` allows to customize the set of unfolded operators upon start-up to the user's needs (refer to Sec. 2.7 for details). Operator nodes can be added to a workflow by double-clicking on the operator name. A new operator node is then instantiated in the top left corner of the corresponding workflow tab, i.e., the *active* workflow (see below). Alternatively, an operator can be selected by clicking once on its name and afterwards clicking once on the position in the workflow tab where the new operator node should be positioned.

2.4.2   Workbench area

Workflows can be designed and executed in the workbench area on the right of the main window. It allows for instantiating multiple workflows in parallel where each workflow is linked to an individual tab of the workbench panel. A new workflow tab can be added via the item 'New' in the context menu of the workbench. The context menu is displayed upon right-click on an empty location of the workbench area. Upon selecting the item 'New' a new tab is added to the workbench panel. By default, the name of the new workflow is 'Untitled', but it can easily be renamed via the corresponding item 'Rename' in the context menu. Via this menu it is also possible to close a workflow tab if no longer required. Note that its contents are lost if not saved before! The currently selected tab in the workbench contains the *active* workflow which can be edited and where new operator nodes can be added as outlined in the previous subsection.

Operator nodes. For each operator selected via the selection menu, a node in terms of a rectangle is added to the currently active workflow. Above the rectangle the name of the operator is displayed, while on its left and right side the operator's input and output ports are shown as circles and squares. Circles are associated with operator parameters of directions `IN` or `OUT`, while squares refer to parameters with direction `INOUT` (Sec. 3.1.2). The latter ports are duplicated on both sides of the node. The colors of the circles indicate their type. Blue circles refer to required parameters, yellow circles are associated with optional parameters, and red circles are linked to supplemental parameters. To the left and right of the ports, respectively, the name of the corresponding parameters are written. Once operator nodes have been added to a workflow, they can easily be dragged and repositioned as well as resized via intuitive mouse actions.

For each operator node a context menu can be popped up by clicking the node with the right mouse button. From this menu it is possible to delete the node (item `'Remove'`), or to configure the view via the item `'Options'`. It, e.g., allows to select the set of operator parameter ports to be shown, i.e. either all parameters or just the subset of non-expert parameters. From the context menu of a node it is also possible to configure the node (item `'Configure'`).

Node configuration and states. On selecting the item for configuration, a window is displayed which allows to enter parameter values (for an example, see the two windows at the bottom of Fig. 9). The window is automatically generated, i.e., actually the same mechanisms as for executing operators via the graphical operator runner are applied (cf. Sec. 2.3). Accordingly, the configuration window is identical to the corresponding operator control window and shares the same layout, except for the control buttons and the batch mode tab which are missing.

Operator parameters for a certain node can directly be specified via the configuration window, they can be loaded from a proper parameter file in XML format[4], or they can be configured by dragging edges between ports of different nodes with the mouse to propagate output data from one node as input data to another. To add an edge, move the mouse over an output port of a node until the port is surrounded by a green square, then press the left mouse button. Subsequently, while keeping the button pressed, move the mouse to the desired input port of another node. Once a green rectangle shows up around the target input port, release the button. Note that on dragging edges Grappa performs type and validity checks. Only ports being associated with compatible parameter data types can be linked to each other. Two parameter data types are compatible if they are either equal, the target data type is a super class of the source data type, or if `Alida` has access to a converter allowing to transform the source data type into the target type (refer to Sec. 3.4 for details). Also edges are forbidden that would induce cycles into the workflow graph.

Nodes in a workflow can have different states indicated by the color of their border. Red

---

[4]The parameters of an operator can be saved to such a file via the corresponding options in operator control windows as displayed by the graphical operator runner. Also the node configuration windows in Grappa provide an option to save the current parameter settings of the node.

framed nodes are not ready for execution, i.e., their configuration is not complete. If a node is readily configured and can directly be executed, its border has a yellow color, while nodes that are configured, however, require additional input data from preceeding operator nodes have an orange color. Prior to executing these orange nodes it is, thus, necessary to execute the preceeding nodes first. Note that Grappa takes care of such dependencies, i.e., automatically executes nodes first from which result data is required for proper workflow or node execution. The state of a node is updated by Grappa in real-time, i.e., each change in its configuration directly invokes internal checkings and may result in a change of the node's color.

Workflow execution.   Grappa offers various modes for executing a complete workflow or parts of it. From the context menu of the workbench the item 'Run' is available which executes the complete workflow, i.e., all nodes currently present on the tab. From the context menu of a single node and its 'Run...' item also the whole workflow can be executed (item 'Workflow'). Alternatively, via the item 'Nodes from here' it is possible to only execute the nodes of the workflow subgraph for which the current node is the root (of course considering required dependencies). Finally, the item 'Node' allows for running the workflow until the node in question. As mentioned before, Grappa automatically takes care of resolving dependencies, i.e., upon executing a node all nodes having a yellow or orange border and being predecessors of the node in question are also executed. Note that the execution of a workflow will fail if one of the nodes is still colored red, or if a node does not produce proper output data required by others.

After successful execution of the workflow or a subset of nodes, the colors of the corresponding nodes change to green indicating that result data are available. For all terminal nodes having no successor the result frames are automatically opened (see Fig. 9 top right). For all other nodes the result data can graphically be examined via the nodes' context menus from which the result windows can manually be opened. Once a node has been executed and is colored in green, it is not possible to re-execute the node until its configuration, or at least the configuration of one of its preceeding nodes, was changed.

2.4.3   Menubar and shortcuts

The Grappa main window features a menubar offering quick access to the basic functions of Grappa and some additional convenience functionality simplifying the work with the editor.

Via the menu item 'File' workflows can be saved to and read from disk. By saving a workflow currently two files are written to disk, one containing the information about the nodes and their configuration, and one storing graphical information regarding the current workflow layout. Both are required to load a workflow again. The first one has the extension '.awf', the latter one the extension '.awf.gui'.

Via the menu item 'Workflow' new workflows can be added and existing ones be renamed, closed, executed or interrupted. As outlined in Sec. 2.3, `Alida` supports two categories of opera-

tors, i.e. operators mainly dedicated to direct application by non-expert users and operators for special tasks and expert usage. Via the item 'Options' the menubar allows to switch the view in the selection menu between both categories. Also progress messages triggered by the operator node during execution and optionally shown in the status panel can be enabled or disabled via this menu. Finally, the menu item 'Help' grants access to Alida's online help system where information about its functionality and descriptions of the operators can be found.

The most important functions for workflow and node handling in Grappa are also accessible via keyboard shortcuts. Currently the following shortcuts are implemented:

- Ctrl-N   —   open a new, empty workflow in a new tab

- Ctrl-U   —   rename the active workflow

- Ctrl-W   —   close the active workflow

- Ctrl-S   —   save the active workflow to disk

- Ctrl-L   —   load a workflow from disk into a new tab

- Ctrl-A   —   run the complete workflow

- Ctrl-P   —   open the configuration frames of all selected nodes in the active workflow

- Ctrl-X   —   delete all selected nodes in the active workflow

2.5   Commandline user interface

The command line user interface of Alida allows to invoke all Alida operators properly annotated to grant generic execution. In the following examples the operators MatrixSum and ApplyToMatrix, included as demo operators in Alida, are used to explain the usage and features of this user interface.

Basics.   You may invoke an operator by calling the command line operator runner:

```
java de.unihalle.informatik.Alida.tools.ALDOpRunner [−u] [−v] [−n] [−r] [−s] \
    <classname> {parametername=valuestring}∗
```

It expects as arguments the name of the operator class to be executed and its parameters. The following options are available:

- -u / --usage   —   prints the help

- -v / --verbose   —   prints additional information during operator execution

- `-n` / `--donotrun`   —   does not execute the operator but rather prints its parameters

- `-r` / `--useRegEx`   —   interprets the `classname` as a regular expression

- `-s` / `--showProgress`   —   enables the display of progress messages on console during operator execution

- `--noDefaultHistory`   —   if this flag is given, no history is read or written for input and output parameters contrarily to `Alida`'s standard behaviour.

The operator `MatrixSum` for row-wise summation of the elements within a matrix can be called as follows:

```
java de. unihalle . informatik. Alida. tools . ALDOpRunner MatrixSum \
      matrix=' [[1,2,3],[4,5,6]] ' sums=−
```

which returns as result on standard output

```
sums = [6.0,15.0]
```

Parameter values are specified as name=value pairs. `Alida`'s syntax for 2D array should be self-explanatory from this example. As the mode of summation is not supplied as a parameter its default is used.

The name of operators need not to be fully qualified if they remain unambiguous. There are further options for abbreviation as well as regular expressions, see the help message of `ALDOpRunner`.

Note, that the command

```
java de. unihalle . informatik. Alida. tools . ALDOpRunner MatrixSum matrix='[[1,2,3],[4,5,6]]'
```

will return no output at all as the command line user interface returns only those output parameters which have been requested by the user. This is facilitated providing a dummy value for an output parameter, which is `-` in the example.

The enumeration defined in `MatrixSum` for `summarizeMode` is set in the next example. If a wrong value for an enumeration is given the `ALDOpRunner` prints a list of admissible values. The example also demonstrates redirection of output to a file, `sums.out` in this case, which is the standard in `Alida` if the value of an output parameter is preceded with a '`@`':

```
java de. unihalle . informatik. Alida. tools . ALDOpRunner MatrixSum matrix='[[1,2,3],[4,5,6]]'
   summarizeMode=COLUMN sums=@sums.out
```

Input can be read from file as well:

```
java de.unihalle.informatik.Alida.tools.ALDOpRunner MatrixSum matrix=@data sums=−
```

where the file data contains the string defining the matrix, e.g., `[[1,2,3],[4,5,6]]`

Derived classes The demo operator `ApplyToMatrix` takes as one parameter another operator. In this case this operator needs to extend the abstract class `ALDSummarizeArrayOp`. When invoking the `ApplyToMatrix` operator from command line we thus have to handle derived classes as value for parameters. In the graphical user interface `Alida` features a combo box where we may choose from. In the command line interface `Alida` allows to prefix the value of a parameter with a derived class to be passed to the operator. This is necessary as `Alida` has, of course, no way to itself decide if and which derived class is to be used. `Alida`'s syntax is to enclose the class name in a dollar sign and a colon. The value of the parameter is empty in this example as the operator `ALDArrayMean` can be instantiated with out arguments. As evident in the following example, abbreviations of the fully qualified class name are accepted as long as they are unambiguous.

```
java de.unihalle.informatik.Alida.tools.ALDOpRunner ApplyToMatrix \
   matrix=' [[1,2,3],[4,5,6]] ' \
   summarizeMode=ROW \
   summarizeOp='$ALDArrayMean:' \
   summaries=−
```

results in

```
summaries = [2.0,5.0]
```

`ALDOpRunner` may be persuaded to show all operators derived from `ALDSummarizeArrayOp` and known within the user interface if we enter an invalid class name:

```
java de.unihalle.informatik.Alida.tools.ALDOpRunner ApplyToMatrix \
       matrix=' [[1,2,3],[4,5,6]] ' \
   summarizeMode=ROW summarizeOp='$dd:' \
   summaries=−
```

yields

```
ALDStandardizedDataIOCmdline::readData found 0 derived classes matching <dd>
      derived classes available :
  de.unihalle.informatik.Alida.demo.ALDArrayMean
  de.unihalle.informatik.Alida.demo.ALDArrayMin
  de.unihalle.informatik.Alida.demo.ALDArraySum
```

Supplemental parameters are handled like other parameters

```
java de.unihalle.informatik.Alida.tools.ALDOpRunner ApplyToMatrix \
  matrix=' [[1,2,3],[4,5,6]] ' \
  summarizeMode=COLUMN \
  summarizeOp='$ALDArrayMin:{}' \
  summaries=− \
  returnElapsedTime=true \
  elapsedTime=−
```

gives

```
  summaries = [1.0,2.0,3.0]
  elapsedTime = 4
```

Parameterized classes   Alida supports so called *parameterized classes*. A parameterized class is essentially just an ordinary class where however some member fields have been declared to be required for an object of this class to be properly instantiated by Alida. These member fields resemble quite some analogy to parameters of operators and share some properties (see Sec. 3.1.3 for details). The syntax for parameterized classes is a comma separated list of name=value pairs enclosed in curly brackets where names refer to annotated member variables of the parameterized class. This is shown for the class ExperimentalData1D which holds an 1D array of experimental data and descriptive text as annotated member fields.

```
java de.unihalle.informatik.Alida.tools.ALDOpRunner SmoothData1D \
  experiment='{ baselineCorrected=false , \
    description="my_experiment", \
      data    =[1.0,2.0,2.2,3.3,2.0,1.0,1.0,1.0,1.0,2.0,3.3,2.0]}'    \
  smoothingMethod=MEAN width=3 \
  smoothedExperiment=−
```

yields

```
smoothedExperiment = { baselineCorrected=false ,
    data    =[1.5,1.73,2.5,2.5,2.1,1.33,1.0,1.0,1.33,2.10,2.43,2.65]    ,
    timeResolution=NaN , description="my_experiment" (smoothed) }
```

If a class derived from ExperimentalData1D was to be supplied to the operator, the curly brackets can be prefixed by a derive class definition starting with a dollar sign and ending with a colon as shown for the summarizing operators above.

Advanced examples

The following example shows, that the standard sytnax used for file I/O may be nested

```
java de.unihalle.informatik.Alida.tools.ALDOpRunner SmoothData1D \
        experiment='{data=@myexp.data,description="Demo_experiment"}' \
        smoothedExperiment=@Exp1-smooth.txt
```

Here, the parameter `description` of the parametrized class `ExperimentalData` is directly parsed from the string given on the command line, while the parameter `data` is parsed from the content of the file `myexp.data`. The resulting `smoothedExperiment=@Exp1` is written to the file `Exp1-smooth.txt`.

Likewise, if an output parameter is a parametrized class, as in this example, a subset of its parameters may be written to file, a part to standard output:

```
java de.unihalle.informatik.Alida.tools.ALDOpRunner SmoothData1D \
        experiment='{data=@myexp.data,description="Demo_experiment"}' \
        smoothedExperiment='{data=@myexp-smooth.data,description=-}'
```

will return

```
smoothedExperiment = { data written using @myexp-smooth.data ,
        description="Demo_experiment" (smoothed) }
```

which indicates that the data of the smoothed experiment have been written to the file `@myexp-smooth.data` as requested.

2.6   History

One of the main features of `Alida` is the capability of automatically documenting data processing pipelines. The operator concept allows for automatically logging all data manipulations, which can subsequently be used to convert the processing history into a directed graph data structure, denoted *processing graph* in the following.

The `Alida` operator concept defines operators as the only places where data are processed and manipulated. Each invocation of an operator is associated with a certain configuration of the operator, defined by the values of its `IN` and `INOUT` parameters. A data analysis pipeline usually consists of a set of different operators that are applied to incoming data and produce result data. The order in which the operators work on the data depends on the specific pipeline as well as on the input data. The invocation of operators can be of pure sequential nature or subsume parallel processing steps. In addition, a nested application of operators is possible. Given these principles, each analysis pipeline and its data flow may be interpreted and visualized as a directed acyclic graph (cf. Fig. 10 for an example).

The processing graph is stored in XML format in a file accompanying the actual data
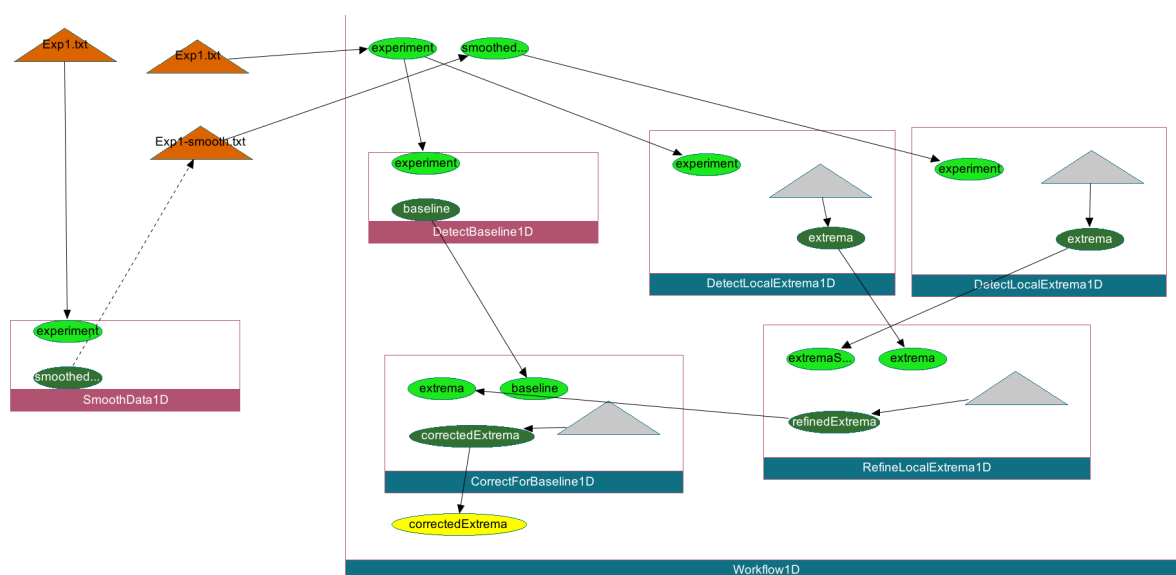
Figure 10: A processing graph: the directed acyclic graph represents the application of nested operators. Calls to operators are depicted as rectangles, input and output ports as ellipses filled in light or dark green, respectively. The yellow ellipse indicates the result data object to which this processing graph is linked to. The triangles relate to newly generated data objects which are colored in grey unless the data object was read from file. In this case the triangle is colored orange.

object file. The format basically relies on *GraphML*[5] with some `Alida` specific extensions. If the history is stored externally when a data object is written to disk depends in general on the data type. However, when invoking operators from the command line user interface, most `Alida` data types will write a history if output of a parameter is redirected to a file. `Alida` uses the extension '.ald' for an `Alida` *processing graph* file. The same is true when reading data. I.e., in general it depends on the data type if a history is read from file, if existing. The command line user interface will do this in most cases.

Note, the identity of data is *not* preserved in the processing history across file boundaries. If two (or more) input data for the current top level operator are loaded from the same file, both will nevertheless be displayed as different data nodes in the history. The reason is that object identity is not – and maybe even cannot – be checked from the processing history of former operations.

A processing graph basically consists of operator and data nodes which are connected by edges indicating the flow of data, as can be seen from Fig. 10. The figure shows a screenshot of `Chipory` which is a graph visualization tool derived from *Chisio* (see Appendix A for details). Within the processing graph each operator node, which is linked to the *invocation* of a specific

---

[5]GraphML website, http://graphml.graphdrawing.org/

operator, is depicted as a rectangle with the operator's classname in the bottom line. For each
input and output parameter object the operator node features input and output ports which
may be conceived as the entry or exit points of data into and out of the operator. These ports are
depicted as filled ellipses in light green (input ports) and dark green (output ports), respectively.
Each input port has exactly one incoming edge, while an output port may be connected to
multiple target ports, depending on where the data is passed to. Each port of an operator has
an individual name indicating the input or output object associated with the port.

In addition to operator nodes and their ports there are also data nodes in the graph, cor-
responding to the creation of new data objects, e.g., when data is read from file, cloned or
generated from scratch. These are depicted as triangles filled in light grey in most cases. If data
is read from file, the triangle is tagged by a string and colored orange. If in addition a processing
graph of a former analysis procedure was read, this history is also included into the processing
graph and connected by a dashed edge (see top left part of Fig. 10).

2.7   Configuring `Alida`

Sometimes it is desirable to configure some properties of `Alida` or the general behavior of specific
operators at runtime, e.g., to specify initial files or directories where operators should work on.
`Alida` basically supports two different ways for user specific configuration:

a) environment variables

b) properties of the Java virtual machine specified with the
   option '`-Dproperty=value`' upon invocation of the JVM

This order already reflects the priority of the options, i.e., environment variables overwrite JVM
properties. If for a certain requested property no configuration values are provided by any of
these ways, default settings are used. Some variables of general interest are used by `Alida`
and are summarized below. Further variables may be introduced, e.g., by additional operators
implemented in the framework.

In the following list, both the environment variable and the name of the property are given
in the form of `property / environment variable`:

`alida.oprunner.level / ALIDA_OPRUNNER_LEVEL`

Used by the graphical operator runner `ALDOpRunnerGUI` and Grappa, i.e., the application
`ALDGrappaRunner`, to configure which set of operators is to be displayed initially in the
selection menu. Possible options are either all available operators ('standard') or just the
ones categorized as being easier to use ('application'). The default is 'application'.

`alida.oprunner.favoriteops` / `ALIDA_OPRUNNER_FAVORITEOPS`   Holds a colon separated list
of filenames. Each file contains lines of fully qualified operator names which will be unfolded
in the operator selection window when starting the graphical user interface or `Grappa` (see
Sec. 2.3 and 2.4). The default is `${user.home}/.alida/favoriteops`.

`alida.oprunner.operatorpath` / `ALIDA_OPRUNNER_OPERATORPATH`   Here a colon separated
list of packages may be specified. Each package and all its sub-packages are searched
for operators in the classpath. These operators are incorporated in the tree of available
operators in the graphical user interface and in `Grappa`. This feature is useful to incorporate
operators which are not compiled, but just added within a jar-archive.

`alida.oprunner.workflowpath` / `ALIDA_OPRUNNER_WORKFLOWPATH`

Here a colon separated list of directories may be specified each of which is searched for work-
flows saved in a file. These workflows are incorporated in the tree of available operators in
the graphical user interface in `Grappa`. The default is `${user.home}/.alida/workflows`.

`alida.versionprovider_class` / `ALIDA_VERSIONPROVIDER_CLASS`

Implementation of `de.unihalle.informatik.Alida.ALDVersionProvider` to be used for
version information retrieval in process documentation (Sec. 3.5.4).

`alida.version` / —

Only available as JVM property, this variable is used by `ALDVersionProviderCmdLine`
to   discover   the   software   version   to   be   stored   in   the   history.   The   class
`ALDVersionProviderCmdLine`   implements   the   `ALDVersionProvider`   interface   and
gets the version to be used from the JVM.

3 The Programmer's View

3.1 Alida operators

3.1.1 Using operators

To use an operator an object of the operator class needs to be instantiated, and input data as well as parameters have to be set for this object. Subsequently, the operator can be invoked using the method `runOp()`. After return from that routine the results can be retrieved from the operator.

Important note: Do no invoke an operator directly by its `operate()` method as this will prevent the processing history from being constructed. Anyway, this only would be feasible from within the package of the operator as the abstract method `operate()` is declared `protected`.

An example of how to use an operator is given in Fig. 11. First, a new instance of the operator is created (line 1), and subsequently further input parameters are set (lines 2 and 3). If all required input parameters have been assigned for the operator object, it can be invoked calling its `runOp()` method (line 4). Upon invocation of `runOp()` the validity of input parameters is checked. Validity requires for an operator that all required input parameters have values different from 'null'. In addition, the implementation of an operator may impose further constraints which, e.g., may restrict the admissible interval of numerical parameters (see Sec 3.1.3). Subsequent to successful validation, the method `operate()` is invoked. Each operator is supposed to implement this method as it does the actual work. After return from `runOp()`, the resulting output data can be retrieved from the operator either directly accessing the member variables or by getter methods as provided by the specific operator. Note that the value of the operator parameters may have changed upon return from `runOp()` due to modifications in the `operate()` method. `runOp()` may throw an exception if validation of inputs and parameters or data processing itself fails.

```
1    ApplyToMatrix normalizeOp = new ApplyToMatrix( data);
2    normalizeOp.setSummarizeMode( ApplyToMatrix.SummarizeMode.ROW);
3    normalizeOp.setSummarizeOp( new ALDArrayMean());
4    normalizeOp.runOp();
```

Figure 11: An example of how to use an operator, in this case `NormalizeExperimentalDataOp`.

For the `runOp()` method two version are available. Besides the one mentioned above without arguments, the method `runOp(hidingMode)` is available where the value of 'hidingMode' influences the visibility of the operator invocation in the history. If 'hidingMode' is VISIBLE then the invocation of the operator is visible in the history. If the value is HIDDEN the invocation

of the operator and all is children is hidden. Finally, if `hidingMode` is set to `HIDE_CHILDREN` the operator itself is visible, but all its children are hidden from the history See Section 3.5 for more details.

An operator object may be reused to invoke processing several times as long as input parameters are changed between subsequent calls of `runOp()`.

3.1.2   Implementing operators: Basics

Each operator in `Alida` is implemented by extending the abstract class `ALDOperator`. The example given in Fig. 12 is the implementation of the demo operator `MatrixSum`, included in the package `de.unihalle.informatik.Alida.demo`. It shows that an operator usually has to be annotated with the `@ALDAOperator` annotation provided by `Alida`. Some functionality of `Alida`, most importantly the execution via automatically generated user interfaces, requires this annotation to register the class as an `Alida` operator. If an operator is annotated with `@ALDAOperator`, a public standard constructor has to be supplied (see below), otherwise a compilation error will result. Note that abstract classes can not be annotated with `@ALDAOperator`.

An operator may declare its preferences for generic execution, i.e., whether to be or not to be generically executed, by using the parameter `'genericExecutionMode'` of the annotation. It currently supports four possible values:

- `NONE (default)`, to prohibit generic execution completely
- `SWING`, to allow generic execution via GUI only,
- `CMDLINE`, to allow generic execution via command line only, and
- `ALL`, to allow for generic execution in general.

Furthermore, operators can be categorized into `'STANDARD'` or `'APPLICATION'` using the parameter `'level'`. The latter one is intended to subsume only operators that can easily be applied by non-expert users, while the first category subsumes all operators. The graphical operator runner included in `Alida` provides two different view modes for either only operators annotated as `'APPLICATION'`, or all operators registered according to the `@ALDAOperator` annotation.

Optionally, a short documentation of the operator's functionality and usage can be defined using the parameter `'shortDescription'` which takes a string as argument. The short description is for example used as tool tip text in the package tree shown in `Alida`'a main window (see Sec. 2.1). Note that this string should really be short and precise, a longer documentation to be included in `Alida`'s online help system can be defined via a predefined operator method (see paragraph 'Online Help' on page 33 for more details).

Finally, the annotation also allows to enable batch processing for an operator. If the parameter `'allowBatchMode'` is set to `'true'`, which is the default, the operator control frame

triggered by the GUI operator runner will show the batch mode tab, and it is expected that the operator behaves reasonable in that mode. If the parameter is set to `'false'`, batch mode execution will not be possible at all.

```
1   /**
2    * Demo operator to calculate colum or row sums of a 2D array.
3    *
4    * @author posch
5    */
6   @ALDAOperator(genericExecutionMode=ALDAOperator.ExecutionMode.ALL,
7           level=ALDAOperator.Level.APPLICATION)
8   public class MatrixSum extends ALDOperator {
```

Figure 12: Example deriving the operator `MatrixSum`.

There are five issues which have to be taken care of when implementing an operator, namely

- to define the parameters of the operator,
- to implement the functionality of operation per se,
- to provide constructors, particularly a public one without arguments
- to optionally constrain admissible parameter values,
- and to indicate whether this operator prefers a complete processing history or a processing history according to data dependencies.

The first three issues are described in the following, while the last two are deferred to Sec. 3.1.3.

Parameters. The common way to define the parameters of an operator is by annotation of corresponding member variables. In addition, since `Alida` 2.5 it is also possible to dynamically add and remove parameters via methods provided by `ALDOperator`. For defining parameters via annotations currently a modified version of the annotation `@Parameter` as under development for ImageJ 2.0[6] is used. The relevant fields of this annotation are listed below and will be detailed in the following:

- `direction` — direction of the parameter, i.e., `IN`, `INOUT` or `OUT`
- `required` — flag to mark required parameters
- `label` — custom name of parameter
- `description` — short descriptive explanation
- `supplemental` — flag to mark supplemental parameters

---

[6]ImageJ 2.0 project, http://developer.imagej.net/about

```
1   /**
2    * Input matrix
3    */
4   @Parameter( label= "Input_matrix", required = true,
5       direction  = Parameter.Direction.IN, description = "Input_matrix.")
6   private Double[][]  matrix;
7
8   /**
9    * Mode of summarizing
10   */
11  @Parameter( label= "Summarize_mode", required = true,
12      direction  = Parameter.Direction.IN, description = "Sum_over_columns_or_rows?")
13  private SummarizeMode summarizeMode = SummarizeMode.ROW;
14
15  /**
16   * 1D Array of sums.
17   */
18  @Parameter( label= "sums",
19      direction  = Parameter.Direction.OUT, description = "Row_or_column_wise_sums.")
20  private transient Double[] sums = null;
```

Figure 13: Example defining the parameters of `MatrixSum`.

- `mode` — importance category of the parameter
- `dataIOOrder` — I/O rank among all parameters of the operator
- `callback` — name of a callback method to be automatically invoked upon changes of the parameter's value
- `modifiesParameterDefinitions` — changing the parameter's value may add or remove parameters of this operator

Fig. 13 shows an example how parameters are defined this way. If the `'direction'` of a parameter is set to `'IN'` or `'INOUT'`, the field `'required'` defines whether this parameter is required or optional. The field `'description'` of the parameter gives a textual explanation, and the `'label'` may be used for display purposes. Setting the field `'supplemental'` to `'true'` declares the corresponding parameter as supplemental. Via the Java inheritance mechanism an operator inherits all parameters defined in its super classes.

The annotation parameter `'dataIOOrder'` allows to rank parameters in interface generation. For example, in GUI generation it might be favorable to place the most important parameters on top of the window, while parameters of minor importance only appear at the bottom. Likewise,

in command line tools some parameters might be supposed to appear earlier in the help system than others. Such a ranking can be achieved by specifying an I/O order. Smaller values refer to a high importance of the corresponding parameter, larger values to minor importance.

`Alida` allows to categorize operator parameters according to the level of knowledge required for their use. Often some parameters of operators are only of interest for experts, and non-expert users do not even have to be aware of them. To this end each parameter may be annotated as `'STANDARD'` or `'ADVANCED'`. Accordingly, `Alida`'s graphical user interface allows to switch the view of parameters between showing only standard parameters and displaying all.

The annotation parameter `'callback'` allows to specifiy the name of a method of this operator which is to be automatically invoked by `Alida` if the annotated parameter's value is changed. Note the callback method is invoked automatically only when changing the parameter value using the `setParameter()` method. Callback functions offer amongst others the possibility to dynamically reconfigure operators depending on the current context. In particular it is possible to add and remove parameters dynamically. Consider for example an operator allowing for different types of inputs, e.g. an `int` or a `float` value. Each of these types requires specific treatment in data I/O, i.e. the graphical user interface needs to provide two different graphical elements. Thus, to handle the different parameters one option would be to add two parameters of the two different types to the operator and always display both (although only one is needed at a time). Using callbacks a more elegant way to solve the problem is available. We can add another parameter `'useRealData'` to the operator which allows to select the type of input (see Fig. 14). If we furthermore define a callback function for that parameter the user interface of the operator can dynamically be reconfigured. Depending on the chosen input mode the corresponding input parameter can be added and the second one be removed dynamically. Adding and removing parameters can be accomplished with the methods `'addParameter()'` and `'removeParameter()'` of `ALDOperator`.

```
1   @Parameter( label= "useRealData", required = true, dataIOOrder = 2,
2       paramModificationMode = ParameterModificationMode.MODIFIES_INTERFACE,
3       callback = "initDataType", direction = Parameter.Direction.IN,
4       description = "Should_we_use_real_or_integral_data?.")
5   private boolean useRealData = false;
```

Figure 14: Example declaring a parameter which dynamically reconfigures the operator `ALDDynamicOp` in the demo package.

Note that changing the set of parameters of an operator dynamically requires `Alida` to update its internal representation and also graphical user interfaces attached to the operator. To this end the programmer is requested to let `Alida` know that a callback function changes the parameter set of an operator. This is accomplished using the annotation parameter `modifiesParameterDefinitions` which is to be set to `MODIFIES_INTERFACE`. If changing

the parameter value does only modify the values of other parameters, but not the set of configured parameters, `modifiesParameterDefinitions` is to be set to `MODIFIES_VALUES_ONLY`. Note if `modifiesParameterDefinitions` is set incorrectly, undefined behaviour of the operator may result. The callback method handling the modification of current parameters known by the operator should for safety not assume a consistent configuration of the parameters. For example upon instantiation an inconstitent state my exist temporarily. Care has been taken to consistently configure the operator in any of the constructors of the operator.

```
 1   public void initDataType() throws ALDOperatorException {
 2     if ( verbose ) System.out.println( "ALDDynamicOp::initDataType");
 3     if ( useRealData) {
 4       if ( hasParameter(intParameterName)) {
 5         this.removeParameter(intParameterName);
 6       }
 7
 8       if ( ! hasParameter(floatParameterName)) {
 9         this.addParameter( floatParameterName);
10       }
11     } else {
12       if ( hasParameter(floatParameterName)) {
13         this.removeParameter(floatParameterName);
14       }
15
16       if ( ! hasParameter(intParameterName)) {
17         this.addParameter( intParameterName);
18       }
19     }
20
21     if ( verbose ) this. printInterface ();
22   }
```

Figure 15: Example of a callback method modifying known parameters of the operator `ALDDynamicOp` in the demo package.

Finally it should be noted that in general care has to be taken when using and implementing callback functions and reconfiguring the parameter. For example, mutual calls of different callback functions need to avoid infinite calls. Consider two parameters `width` and `height` which should adhere to a given aspect ratio. Thus each of the two parameters is supplied with a callback function which sets the other parameter according to the new value and the aspect ratio. If this is however done via the `setParamteter()` method this will provoke infinite recursion. To avoid this either the parameter can be changed using directly assigning a value, or – probably

the better choice – using `setParamteter()` only in case that the value indeed needs to be set to a new value. Otherwise, if old and new values are identical, nothing is to be done.

Operator functionality.   The method `operate()` implements the functionality of the operator. All data passed into and returned from the operator have to be passed through the parameters of the operator. They may be set and retrieved with the generic `getParameter(name)` and `setParameter(name,value)` methods of `ALDOperator`, by specific setter- and getter-methods as provided by each operator, or by directly accessing the member fields if allowed. To invoke the processing within an operator, i.e., to run its `operate()` routine, the final method `runOp()` supplied by `ALDOperator` needs to be called by the user of an operator.

Online Help.   `Alida` provides an online help system which allows to show documentation for an operator directly in a separate window in the GUI. While a short description can be defined via the parameter `'shortDecription'` of the `@ALDAOperator` annotation (see above), a longer documentation can be provided by implementing the super class method

> **public** String getDocumentation()

which by default returns an empty string. The result string can contain HTML formattings and will be used to automatically generate the help text for the operator in the online help system.

Constructors.   As noted above, for automatic code generation and documentation capabilities as well as generic execution of an operator, the operator class needs to implement a public standard constructor as shown in Fig. 16. This is, however, not necessary if the operator is only to be used explicitly by the programmer and is not annotated. Further convenience constructors may be implemented which additionally set parameters.

Processing history.   As decribed in Sec 3.1.1 the visibility of an operator invocation in the processing history may be influenced by the parameters of the `runOp()` method. By this means the visibility of each operator invocation may be set to `VISIBLE`, `HIDDEN`, or `HIDE_CHILDREN`. This also allows to determine the visibility of further operators directly invoked by their `runOp()` method but not of operators indirectly invoked. Thus, the implementor of an operator may influence the visibility of its own invocation using the method `setHidingMode(hidingMode)`. The main usage of this method is to set the operators visibility to `HIDE_CHILDREN` to hide recursive calls to further operators from the history. This also hides operator calls which are indirectly invoked arbitrary methods used when implementing the `operate()` method.

   Important note: It is strongly recommended that an operator does not rely on specific initializations, e.g., of private fields, that are performed in a constructor (besides the default

```
 1    /**
 2     * Default constructor.
 3     * @throws ALDOperatorException
 4     */
 5    public MatrixSum() throws ALDOperatorException {
 6    }
 7
 8    /**
 9     * Constructor.
10     *
11     * @param _matrix Input matrix.
12     * @throws ALDOperatorException
13     */
14    public MatrixSum(Double[] [] _matrix) throws ALDOperatorException {
15      this.matrix = _matrix;
16    }
```

Figure 16: Constructors of `MatrixSum`.

```
 1     /**
 2      * Summarizing opererator
 3      */
 4     @Parameter( label= "Summarizing_operator", required = true,
 5           direction  = Parameter.Direction.IN,
 6           description  = "Specifies_the_summarizing_operation_to_apply")
 7     private ALDSummarizeArrayOp summarizeOp;
```

Figure 17: An example of an operator as a parameter in the operator `ApplyToMatrix`

constructor) and depend on the values of input parameters. Rather, it is advised that upon invocation of the `operate()` method an operator performs all necessary initializations according to the parameter settings. This allows to take into account changes of parameter values subsequent to construction of the operator object by using the `setParameter(...)` method or dedicated setter methods. Otherwise generic execution of the operator is not feasible and the operator should not be released for generic execution.

```
1   @ALDAOperator(genericExecutionMode=ALDAOperator.ExecutionMode.ALL,
2          level=ALDAOperator.Level.STANDARD)
3   @ALDDerivedClass
4   public class ALDArraySum extends ALDSummarizeArrayOp {
```

Figure 18: The operator `ALDArraySum` as an operator in standard level.

3.1.3 Implementing operators: Advanced techniques

Operators as parameters. `Alida` supports as parameters of an operator also any (other) `Alida` operator. This is shown for the demo operator `ApplyToMatrix` in Fig. 17. In this case, the class `ALDSummarizeArrayOp` is an abstract operator which takes a 1D array as input and returns a summarizing scalar. Now, one may implement concrete examples of such a summarizing operation. As examples `Alida` ships with operators implementing the summation (`ALDArraySum`), mean (`ALDArrayMean`) and minimum (`ALDArrayMin`) operations. Each operator implements the `operate()` method and has to supply a standard constructor. As shown in Fig. 18, the operator `ALDArraySum` is declared as operator on the `'STANDARD'` in contrast to `'APPLICATION'` level, as it is not expected to be invoked as an application. However, setting the level to standard in the menu of the graphical user interface stills allows their direct execution. When extending the super class `ALDSummarizeArrayOp`, it is necessary to annotate the class with **@ALDDerivedClass** in order to allow `Alida`'s data I/O mechanisms to find the derived class in automatically generated user interfaces. This holds for other parameter types as well, see Sec. 3.1.4.

Constraints on admissible parameter values. The implementation of an operator may impose custom constraints on the input parameters beyond the general requirement, that all required input parameters need to have non-null values. This is achieved by overriding the method

```
public void validateCustom() throws ALDOperatorException
```

which, e.g., may restrict the admissible interval of numerical parameters. This method is called by `runOp()` prior to the invocation of the `operate()` method of an operator. It is supposed to throw an exception of type `ALDOperatorException` with type `VALIDATION_FAILED` and a striking error message if the validation was not successful.

Preference for history graph construction. Each operator may specify a preferred way to create the processing history by setting its member variable `'completeDAG'`. The default mode is a complete processing history, i.e., `completeDAG == true`. This works in all cases, but potentially includes additional operator invocations as performed in the `operate()` method

which do not directly influence the values of the object for which the history is constructed. To generate a leaner history graph the programmer of an operator may choose to set `'completeDAG'` to `false`, see Section 3.5 for details.

In case an implementor decides to completely disable the construction of the processing history this can be accomplished with the static method `setConstructionMode()` of the `ALDOperator` class.

Progress events. `Alida` provides a mechanism for operators to send status and progress messages to the user during execution. These messages are shown in the status bar of the operator control windows (Sec. 2.3.2), in Grappa's log panel (Sec. 2.4), and also on commandline if the parameter `'-s'` is used. Accordingly, in constrast to output to standard out, which is not guaranteed to reach the user in a graphical environment, these messages will always be visible to the user. For triggering such messages, an operator has to fire an event of type `ALDOperatorExecutionProgressEvent` using the method

---

protected void
  fireOperatorExecutionProgressEvent(ALDOperatorExecutionProgressEvent ev)

The event takes a message string as argument during construction which is then displayed to the user.

3.1.4   Implementing operators: Parameters

Derived classes.   As noted above, if an instance of a class is to be supplied in an automatically generated user interface as a value for a parameter of one of its super classes, `Alida` requires the annotation `@ALDDerivedClass`.

Parameterized classes.   `Alida` provides automatic I/O of primitive data types, enumerations, arrays, collections, and operators. In addition, so called *parameterized classes* are supported. Any Java class may be declared to be a parameterized class in `Alida` by annotating the class with `@ALDParametrizedClass` as shown for the class `ExperimentalData1D` in Fig.19. The class is annotated by `@ALDParametrizedClass`, and all members to be configured via `Alida`'s user interfaces are to be annotated with `@ALDClassParameter`. This annotation offers the following arguments:

- `'label'` — name of the parameter, field has same semantics as for parameters of operators

- `'dataIOOrder'` — I/O rank of the parameter

- `'mode'` — importance of the parameter, i.e., `'STANDARD'` or `'ADVANCED'`

- '`changeValueHook`' — post-processing routine after reload (see below)

The first three annotations share their semantics with their counterparts in the `@Parameter` annotation of operators (cf. Sec.3.1.2). Only the forth one is special to parameterized classes. On loading or saving operator configurations, i.e. the values of its parameters, also objects of parameterized classes used as parameters have to be handled. To this end these are also converted to XML format. However, in doing so only *annotated* member variables of the parametrized class are taken into account. Consequently, also only these member variables of the object will get proper values after reload. In some cases, however, this might lead to inconsistencies in an object's state after reload. Consider for example a parameterized class representing a set of points. Most likely this class will define a member variable of a list type which holds the points, and as this member holds the core data of the data type, it will be annotated. Now, for efficiency of implementation it might be of advantage to define another member variable keeping track of the number of points in the list, usually not being annotated. To ensure that this variable is also correctly set on reload of the object, the '`changeValueHook`' argument can be used. It expects a string defining a method of the class at hand which is to be called after all annotated member variables have been properly set during reload. The method may then perform some kind of post-processing of the object's configuration, i.e., set the point counter variable to the number of points found in the given list. Note that the hook method may not expect any arguments.

The above annotations are the only implementational overhead to allow `Alida` to automatically generate user interfaces where the parameterized class acts as a parameter. Operators acting as parameters of other operators may be viewed as special cases of parameterized classes.

Automatic documentation The value of each '`IN`' and '`INOUT`' parameter is recorded upon invocation of an operator via its `runOp()` method using the method `toString()` of the parameter class for later storage in the processing history. Thus, it is recommended to supply an appropriate `toString()` method for data types used as parameters to yield informative histories.

If a parameter of an operator is expected to be documented in the data flow of the processing history, it may be of any Java class being uniquely identifiable. This excludes only primitive data types, interned strings and cached numerical objects. If the parameter need not to be part of the data flow, all classes are acceptable.

Note that before returning from `runOp()`, additional documentation is done for output objects derived from the abstract class `ALDData`. This class essentially features a property list which may be used to augment data objects, e.g., by a filename or URL specifying the origin of the data.

```
1   @ALDParametrizedClass
2   public class ExperimentalData1D extends ALDData {
3
4     /** Description */
5       @ALDClassParameter(label="description", dataIOOrder = 1)
6       private String description = null;
7
8       /** The data */
9       @ALDClassParameter(label="data", dataIOOrder = 2)
10      private Double[] data = null;
11
12      /** Are the data baseline  corrected? */
13      @ALDClassParameter(label="Baseline_corrected",
14          dataIOOrder = 3)
15      private boolean baselineCorrected = false;
16
17      @ALDClassParameter(label="Time_resolution_in_milliseconds", dataIOOrder = 4)
18      private Float timeResolution = Float.NaN;
19
20      /**
21       * Standard constructor is  required
22       */
23      public ExperimentalData1D() {
24      }
25
26      /** Constructor for an experiment.
27       * Baseline correction  is assumed to be false  and nothung known about
28       * the time resolution .
29       *
30       * @param description   a textual  description  of the experiment
31       * @param data   measurements
32       */
33      public ExperimentalData1D( String description, Double[] data) {
34          this( description , data, false, Float.NaN);
35      }
```

Figure 19: The class ExperimentalData1D as an example of a parameterized class.

3.2   Data I/O provider

For the generic execution of operators via command line or graphical user interfaces in `Alida`, knowledge is required about how to perform input and output operations for parameter data objects. In particular, in graphical contexts `Alida` requires functionality to query values for parameter objects from the user via graphical input components, and to adequately visualize parameter values graphically. For invocation of operators from command line, parameter data objects need to be instantiated from textual input, and parameter objects need to be transformed into a user-friendly textural representation.

To enable flexible data I/O `Alida` implements an easily extendable provider mechanism. A *provider* is a class implementing the functionality to perform I/O for objects of specific data types, i.e., to instantiate objects from given user input and visualize their values graphically or in a textural fashion. These providers are managed by *data I/O managers* which keep track of available providers and provide methods for getting provider objects for specific data types at run-time, e.g., to facilitate the automatic generation of graphical user interfaces.

Currently, two user interface contexts are implemented in `Alida`, i.e., a tool for executing operators from command line (cf. Sec. 2.5) and a graphical operator runner based on Java Swing (cf. Sec. 2.3). For both contexts `Alida` already offers built-in providers for a large variety of different parameter data types. In detail, all primitive data types available in Java (e.g., int, double, boolean) as well as corresponding wrapper data types (e.g., Integer, Double, Boolean), enumerations, arrays of primitive and wrapper data types, and also all kinds of Java collections[7] are supported out-of-the-box. Also, operators may be used as input parameters of other operators, and by the concept of parameterized classes (Sec. 3.1.4) `Alida` allows for easy extensibility. In some cases, however, it might be necessary to implement a data I/O provider for a certain parameter data type from scratch. Below we outline the basics of how to accomplish this.

All providers in `Alida` are registered by I/O managers upon start-up. These managers basically keep a map of data types and related providers and allow the framework to get a provider object for a certain parameter data type at run-time. Additionally, they may give hints to the provider classes to adapt their functionality to the current mode of operation. The type of these hints is specific to the context, e.g., command line or GUI, and will be described below.

To enable the managers to automatically register all provider classes available on the classpath, it is necessary to annotate the provider classes with the `Alida` annotation `@ALDDataIOProvider`. This annotation features the field 'priority' which is used to select a provider in case that several providers handle the same data type. This allows, e.g., to override providers delivered by `Alida` with custom made providers. In addition, all providers have to

---

[7]The only exception are collections of collections which are not yet supported.

implement the interface

de. unihalle .informatik.Alida.dataio.provider.ALDDataIO

It basically defines a single method that all providers need to implement:

public Collection<Class<?>> providedClasses();

This method is used by the I/O managers upon start-up to gather information about which classes a specific provider supports, i.e., to fill its maps. Of course, apart from this method the actual functionality of a provider class depends on the user interface context in which it is to be used. In the following subsections we will now describe the specifics of implementing providers for the Java Swing (Sec. 3.2.1) and the command line context (Sec. 3.2.2).

3.2.1   Implementing a Swing data I/O provider

To enforce all providers dedicated to the Swing context to implement the corresponding graphical data I/O concept, Alida defines a specific interface for the Swing context:

de. unihalle .informatik.Alida.dataio.provider.ALDDataIOSwing

This interface subsumes all methods to be implemented by Swing providers.

Graphical data input.   The basic workflow which Alida defines for graphical input of a parameter data object is as follows. First a graphical component needs to be generated by the provider suitable for querying values from the user. Such a component is for example given by a checkbox to query the user for a boolean value, a textfield to query for a string or number, or a combobox to let the user select a single or multiple values from a larger collection. This component is then included in graphical user interfaces, e.g., the graphical control windows of Alida's graphical operator runner. Subsequently, once the user has entered appropriate values, functionality is required to read the specified values from the graphical component.

Accordingly, the interface basically defines the following two methods:

public abstract ALDSwingComponent createGUIElement(
    Field field , Class<?> cl, Object obj, ALDParameterDescriptor descr);

public abstract Object readData(
    Field field , Class<?> cl, ALDSwingComponent guiElement);

The first method is supposed to return a graphical component suitable for being included in graphical control windows. In general, `Alida` does not define any strict rules for these components except that they have to be of type `ALDSwingComponent` and implement the methods defined in that interface. Objects of this type basically wrap an object of type `JComponent`. Only the latter one is actually included in the configuration and control windows. Although there are principally no restrictions on the design of the components returned by providers, note that proper automatic GUI layout is only possible if the different components do not vary too much in their sizes. Hence, it is advisable to layout the components in such a way that they only fill one row in a panel. Elements which naturally obey to this rule are for example buttons, text fields, and combo- or checkboxes. If you require more complex components to query values from the user, one solution could be to return a button as main GUI component by which a new window can be opened with an arbitrary size where the actual parameter values can then be entered. This technique is employed, e.g., by `Alida`'s standard providers for parameterized classes and collections.

Besides wrapping a graphical component, objects of type `ALDSwingComponent` have to implement an event reporter interface. This interface lays the foundation for `Alida` to be able to visualize the current state of an operator's configuration in its control windows in real-time. The interface basically enforces the graphical component to trigger events on changes of the values specified in the graphical component. For most providers relying on Swing components as graphical components it is sufficient to implement listeners for the Swing components and simply forward Swing events as `Alida` events of type `ALDSwingValueChangeEvent` to the framework. Refer to the API documentation of `de.unihalle.informatik.Alida.dataio.provider.swing.components.ALDSwingComponent` and related event reporter and handler classes in the package `dataio.provider.swing.events` for more details. Also note that `Alida` already provides implementations of graphical elements for data I/O of common data which subclass `ALDSwingComponent`. They are to be found in the package `dataio.provider.swing.components` and can be used out-of-the-box.

The method `createGUIElement(field, cl, obj, descr)` takes four arguments as input. The argument `'cl'` specifies the class of the parameter object which should be read by the component (which is of special interest for providers supporting various data types), while the `'obj'` argument allows to pass a default value to the provider. The argument `'descr'` allows to hand over additional information to the provider about the operator parameter with which the object is associated. `Alida` uses this information, e.g., for generating more meaningful labels in the graphical control windows. Note that if no descriptor is available, `'null'` is also a valid value, hence, providers must account for this special case. The first argument `'field'` can be ignored by most providers. There are only a few providers where the object class is not sufficient to instantiate a parameter object, and where in addition the field associated with the member variable representing the parameter is required to have all relevant information available. One

example are collections in Java, where the data type of the elements contained in the collection is not obvious from the class of the collection itself.

The second method `readData(...)` of the `ALDDataIOSwing` interface takes as argument a formerly generated graphical component `'guiElement'` and is supposed to return an object of the specified class. The object should contain the value as currently specified by the graphical component. The `'field'` argument can most of the time be ignored by the provider, refer to the previous paragraph for details.

In addition to the formerly discussed methods the interface defines a third method

> **public abstract void** setValue(
>   Field field , Class<?> cl, ALDSwingComponent guiElement, Object value);

which is supposed to set new values in the graphical component. Its arguments are quite similar to the `createGUIElement(...)` method. The new value to be set is specified by the `'value'` argument.

Graphical data output.   For performing graphical data output the Swing provider interface defines the method

> **public abstract** JComponent writeData(Object obj, ALDParameterDescriptor d);

It basically takes an object `'obj'` as input and generates a suitable Java Swing component to properly visualize the object's value. Note that the class of the object is directly derived from the object itself. The additional descriptor argument is used to enhance the graphical component with more information about the object. The object might be `'null'`, i.e., the provider has to check its value prior to accessing the object directly. For components generated by this method the same rules hold as for the graphical input components. In particular, ensure that each component does not exceed the height of one row in a panel to enable proper GUI layout.

Interaction level.   The I/O manager for the Swing context gives providers a hint on the amount of user interaction level the providers should generate. The reason is that there are situations when only warnings should be signaled to the user or no pop up of windows and user interaction is intended at all. To request the desired level of interaction and to modify the setting, the Swing data I/O manager offers the following methods:

> **public** ProviderInteractionLevel getProviderInteractionLevel()
>
> **public void** setProviderInteractionLevel(ProviderInteractionLevel level)

3.2.2   Command line provider

In analogy to Swing providers each command line provider is required to implement

---

de.unihalle.informatik.Alida.dataio.provider.ALDDataIOCmdline

---

This interface defines two methods to be implemented, namely

---

public abstract Object readData(Field field, Class<?> cl, String valueString);

public abstract String writeData(Object obj, String locationString);

The first method is expected to instantiate an object of class `'cl'` from the string `'valueString'`. For the meaning of the argument `'field'` see the subsection on Swing providers. In general, the syntax of the value string depends on the data type and may freely be defined by the implementation of the provider. However, `Alida` offers the notion of a standardized command line provider taking care of derived classes as well as reading and writing to and from files (see below). Additionally, `Alida` provides a command line provider for parameterized classes featuring a specific syntax (see Sec. 3.1.4). If appropriate, it may be sensible to adopt the syntax for other providers as well.

The method `writeData(...)` is used to format the given object to textual form into a string. The `'locationString'` specifies whether this textual representation is to be returned as the result of the method or should be written to, e.g., a file. In the latter case the string returned may contain information of this process, e.g., return a note that the object was written to a specific file. As for the syntax of the `'valueString'` in `readData(...)`, the syntax of the `'locationString'` may in principle be freely defined by each provider. However, it is advisable to adhere to `Alida`'s standard definitions by the standardized command line provider as described in Sec. 2.5. In addition, the `'locationString'` may also be used by a provider as a format string to modify the textual representation of the object generated.

`Alida` features a so called *standardized commandline provider* to generically handle data I/O from and to file, and for the input of derived classes, in the abstract class `ALDStandardizedDataIOCmdline`. It is easy to implement new providers extending this class and, thus, to automatically inherit the ability to handle file I/O and derived classes. It is only required to implement the two abstract methods

---

abstract public Object parse(Field field, Class<?> cl, String valueString);

abstract public String formatAsString(Object obj);

---

of `ALDStandardizedDataIOCmdline`.

The first method should instantiate an object of class 'cl' from the 'valueString' quite similar to the method `readData(...)` introduced above, however, making no prior interpretation regarding a file to use or derived class to return. This has already been handled by the class `ALDStandardizedDataIOCmdline` prior to calling `parse(...)`. Likewise, the method `formatAsString(...)` should always return a textual representation of the given object as return value. If it is necessary to use formatting information provided as part of the argument 'locationString' of `writeData(...)`, the method

---

public String formatAsString(Object obj, String formatString)

---

may be overridden.

As mentioned, I/O managers may give providers hints to adapt their functionality. In the case of the command line context this is the request to read or write a history file in case the I/O is to be performed from or to file. The method `isDoHistory()` may be used to inspect the state of the manager. The standardized command line provider adheres to this standard.

3.3   XML provider for external representation

XML providers are used to store and retrieve parameter configurations of operators and work-flows. As a consequence, (typically) transient data are not stored (and retrieved).

`ALDXMLObjectType` is the base type of all Alida XML data types. In only contains the classname (fully qualified classname). Extending types will add the data itself. Such XML representations do not only contain the data itself, but allow to infer the class of the represented object. Thus, it is feasible to instantiate the correct object and fill it with all data as stored in the XML object.

Currently there exist the following extending types:

`ALDXMLArrayType`	to hold 1D arrays and collections
`ALDXMLParametrizedType`	for parametrized classes holding a list of (key,value)-pairs, where the values are of type `ALDXMLObjectType`
`ALDXMLEnumType`	for an enum value, which is just represented by a string
`ALDXMLAnyType`	to hold any type, used, e.g., for primitive data types, numerical data types

For writing an XML provider on your own

1. create an XML beans object which can hold the data item. This may be a native XML beans data type like `XmlInt` or `XmlString`, or a custom XML beans data type typically created via an XML schema and the XML bean compiler (`xmlbean` ant task).

2. create an object of type `ALDXMLAnyType`,

- set its member 'className' to the class name of the data item
- set its value to the XML beams object created in step 1.

3.4   Automatic data type conversion

`Alida` features a generic mechanism for data type conversion. This conversion mechanism is currently used within the graphical workflow editor `Grappa` (see Sec. 2.4). `Grappa` allows to connect an output port of one operator to the input port of another operator if the data types of the underlying parameters are compatible. The data types are defined as compatible if either the target data type is assignable from the source data type according to the Java class hierarchy or an appropriate data type converter is supplied. For example `Alida` ships with the converter `ALDNumberConverter` between numeric data types, obviously with potential loss of precision. Likewise conversion from vectors to 1D arrays is supported by the class `ALDVectorNativeArrayConverter`.

Similar to data I/O providers, data type converters are registered on start-up by a singleton instance of the class `ALDDataConverterManager`. This manager redirects conversion requests to appropriate converters. This requires each data type converter to implement the interface

de. unihalle . informatik. Alida. dataconverter. ALDDataConverter

and to be annotated with the `Alida` annotation `@ALDDataConverterProvider`. In analogy to `Alida`'s data I/O provider a priority may be specified which is used by the converter manager to resolve competion between multiple converters for a given pair of data types to be converted.

A data type converter has to implement the methods

Collection<ALDDataConverterManager.ALDSourceTargetClassPair> providedClasses()
boolean supportConversion(Class<?> sourceClass, Type[] sourceTypes,
                                    Class<?> targetClass, Type[] targetTypes)

The first one is called upon start-up and it has to return pairs of source and target data types which the converter is able to handle. However for parameterized types this only indicates that the converter can in principle handle conversion for these classes but depending on the type parameters still may refuse to do so. The second method, `supportConversion` states precisely if conversion is supported taking also type parameters passed as arguments into account.

The actual conversion is performed invoking the method

Object convert(Object sourceObject, Type[] sourceTypes,
               Class<?> targetClass, Type[] targetTypes)

As stated above this method is typically not invoked directly but via the `ALDDataConverterManager`.

Note that both converters supplied by `Alida` are implemented as an `ALDOpertor`. Thus the

conversions are reflected within the processing history unless they are intentionally hidden (see Sec. 3.1 and 3.5).

3.5   The processing history

Data processing pipelines in `Alida` build on the idea of operators that manipulate data objects. According to the specification of `Alida` operators (see Sec. 2.2), data objects that are to be manipulated by a certain operator will have to be stored in member variables of the operator annotated as operator parameters with direction 'IN' or 'INOUT'. Result data objects of an operator will be stored in member variables annotated as parameters with direction 'OUT' from where the user of the operator can access the result object for further processing tasks.

Logging the complete processing history of individual objects enforces `Alida` to link data objects used as inputs or resulting as outputs from operator invocations directly to the manipulating operators. These links essentially form the base to lateron build the history graph representation for each object ever seen by any of the `Alida` operators during a processing chain.

3.5.1   Basics of the history concept

The key for logging all operator invocations and the corresponding operator configurations during each run of a processing pipeline is `Alida`'s port hash. Within this hash all objects participating in the processing pipeline are registered. For each object a link to the relevant port of the most recent operator invocation, which manipulated or generated the object, is stored as a reference to a port object in a weak hash map. This kind of hash map only holds *weak* references to objects, which allows the Java virtual machine to destroy the objects if they are not referenced somewhere else anymore. The port hashmap allows to link input and output ports of operators as well as data ports according to the data flow, and to lateron traceback the sequence of manipulations for each object manipulated during the course of the processing.

The complete history of a data processing chain is only implicitly represented by the links between the different kinds of ports. Each operator invocation is represented by an object of type `ALDOpNode` which consequently needs to store its current inputs and outputs. The `ALDOpNode` class defines input and output ports for input and output objects of an operator. Data ports represent newly created data objects that were not passed to an operator as input so far. Such objects appear, e.g., when a new data object is allocated to store the results of an operator. Altogether these ports provide the functionality to establish connections between new data and inputs and outputs of operators.

The history of a data object is built on request traversing the connections that are stored in `Alida`'s port hash. While many different objects can be linked to a single processing chain, i.e. can be manipulated by operators during one run of various operators, a single object has always its own individual manipulation history. This history is given by a certain path within

the manipulation graph of the complete processing chain. The starting point of the object's path is always the most recent operator invocation which involved manipulations of the object. Consequently, the link to the port associated with this operator invocation is the one stored in the port hash. Tracing back the history from this port then allows to recover all object manipulations and build up the final history graph. Neither the programmer nor the user of an operator have to take care of the data stored in the port hash or the correct logging of operator invocations. Object registration and the update of port links are done automatically each time an object is fed into an operator or taken out of an operator as result. In particular, the `runOp()` method of `ALDOperator` takes care of all this and handles the history data management internally.

The only situation when programmers get in touch with the processing history and the port hash is when the processing history of an object is to be created explicitly, e.g., to be saved to disk. While this is done automatically for some `Alida` data types which provide read and write methods, there is still the need for programmers to take care of this for own data types not providing appropriate read and write routines so far.

3.5.2   Accessing history data

At any point in time during data processing the processing history of any object manipulated is implicitly represented in the processing history graph. To access this data and transfer the processing history from the implicit to an explicit representation, it is possible to generate this history using the static method `createGraphmlDocument()` of the class `ALDProcessingDAG`. This creates the processing history associated with the object in a graph data structure as generated by *XmlBeans* [8]. It is based on the XML schema definition of *GraphML* [9] with `Alida` specific extensions. Although intended for writing and reading the history to or from file (see next paragraph) in the first place, this data structure may also be used to inspect the processing graph as constructed directly from Java.

To store the processing history in XML format, to be more precisely in *GraphML* format, the class `ALDOperator` provides a static method to save the processing history of an object to an XML file:

public static void writeHistory(Object obj, String filename)

These files can then be opened, e.g., with `Chipory` (see Appendix A), to discover details of the analysis process. In subsequent processing chains, these histories can be read from such a file using the following static method of `ALDOperator`:

public static void readHistory(Object obj, String filename)

This reads the processing history of `obj` from the specified file. If such a history is present, this old history is attached to the newly created data port initially linked to this object. Note that

---

[8]http://xmlbeans.apache.org/
[9]http://graphml.graphdrawing.org/

invoking the `readHistory()` method on an object will trigger the registration of the object in the port hash if this did not happen before.

3.5.3   Different modes of processing graph construction

There are two mechanisms to influence which operator invocations are to be included or excluded from a processing history. One is hiding of operator invocations by the user of an operator, the other to influence the explicit construction of the processing graph to a certain extent by the programmer of an operator. We discuss both issues in turn in the remainder of this section.

Hiding of an individual invocation of a single operator is accomplished using `runOp(HidingMode.HIDDEN)` as mentioned in Section 3.1.1. This effectively excludes this invocation from any processing graph for an object which indeed depends on this operator invocation. Using `runOp(HidingMode.HIDE_CHILDREN)` to invoke an operator will include this operator in the history, but recursively hide invocations inside this operator. This only allows to determine the visibility of further operators directly invoked by their `runOp()` method but not of operators indirectly invoked. In addition, the operator may manipulate the visibility of further operators invoked from its `runOp()` method using the method `setHidingMode(hidingMode)`. The main usage of this method is to set the operators visibility to `HIDE_CHILDREN` to hide recursive calls to further operators from the history. This also hides operator calls which are indirectly invoked arbitrary methods used when implementing the `operate()` method.

This hiding of operator invocation can be ignored when creating the processing graph using the static method `createGraphmlDocument()` of class `ALDProcessingDAG` as described in the class documentation, e.g., for debugging purposes.

The second mechanism to influence the processing graph is somewhat more involved. If the mode `ALDProcessingDAG.HistoryType.COMPLETE` is used when constructing the history via `ALDProcessingDAG.createGraphmlDocument()`, for each operator invocation (i.e. each `ALDOpNode`) included in the processing graph recursively all nested invocations of further operators are also included into the graph unless the invocation was hidden.

However, sometimes only the dependency as implied by the data flow should be reflected in the processing graph. This can be accomplished by using the mode `ALDProcessingDAG.HistoryType.DATADEPENDENCIES` on generation.

A third mode of generation is available, namely `OPNODETYPE`. In this case when constructing the processing graph each `ALDOpNode` decides whether all its directly nested operator invocations are to be considered, or only those which are connected via data dependencies. This decision is made by the programmer and the user of the operator by appropriately setting the protected member variable `completeDAG`. The same is true, if the history is not constructed for one of these two objects, but for another object which depends on one of them.

In the abstract class `ALDOperator` the member `completeDAG` is set to true, thus, a complete history is the default. To be on the safe side the programmer of an operator may choose this default mode with the only penalty to potentially generate history graphs with non important operator invocations. If she or he is certain that the data dependencies of the operator yield all (intended) operator invocations setting the variable to false may yield leaner processing histories.

3.5.4   Software version handling

Documenting the processing history for data items requires not only to log all operator invocations and their parameter settings, but also to remember the software versions of the operators. Consequently, the method `runOp()` of `ALDOperator` retrieves upon invocation the current software version of an operator. Indeed, where this version is queried from can be specified by the user. Popular options are for example version control system like SVN, CVS or Git, but there are lots of alternatives as well. `Alida` implements a dynamic framework allowing for flexible runtime configuration of the software version retrieval procedure which is outlined below.

The basis for runtime configuration in `Alida` is the abstract class `ALDVersionProvider` in package `de.unihalle.informatik.Alida.version` which all version provider classes have to extend. `ALDVersionProvider` mainly defines the method

---

public String getVersion()

---

returning a string object, e.g., containing the software version or another identifier or tag. The concrete implementation of `ALDVersionProvider` to be used for version information retrieval can be specified at runtime by JVM properties or environment variables (cf. Sections 2.7 and 3.6). In particular, use the property `alida.versionprovider_class` to specify the desired class, e.g.:

−Dalida.versionprovider_class=\
    de.unihalle.informatik.Alida.version.ALDVersionProviderCmdLine

---

Of course, the class passed via this option needs to extend `ALDVersionProvider`. If this is not the case a warning is printed to standard error and the version provider mechanism falls back to the dummy version provider `de.unihalle.informatik.Alida.version.ALDVersionProviderDummy` shipped with `Alida`. It always returns the version identifier `'unknown'`.

Internally, a factory named `ALDVersionProviderFactory` extracts the desired implementation from the given environment variable or JVM property and creates corresponding objects. Note that by default a dummy version provider is initialized, i.e. the version is al-

ways set to `'unknown'`. As default implementation `Alida` supplies the programmer with class `de.unihalle.informatik.Alida.version.ALDVersionProviderReleaseJar` which returns the `Alida` release identifier included in the current Alida.jar.

Alternatively, the class `ALDVersionProviderCmdLine` can be used. It allows reading version data from the environment. The class extracts version data from another JVM property named `version`. Hence, invoking `Alida` with the following options,

---
−Dalida.versionprovider_class=\
      de.unihalle.informatik.Alida.version.ALDVersionProviderCmdLine −Dversion=4711
---

will insert the version ID `4711` into extracted processing history files.

Since version 2.3 Alida ships with native support for extracting version information from Git repositories. To extract version information from a Git repository use the following provider class:

−Dalida.versionprovider_class=\
      de.unihalle.informatik.Alida.version.ALDVersionProviderGit
---

To allow the provider to work properly, the environment variable `'GIT_DIR'` needs to be set to the Git repository directory. Note that it needs to point directly to the .git directory, not only to the parent folder. If the repository cannot be assessed, Alida looks for a local revision file. If this is neither available, a dummy version string is returned.

3.6   Configuring `Alida`

As described in Sec 2.7, `Alida` allows to configure some of its properties and general behaviours at run-time. To grant programmers easy access to this functionality for their own operators as well, the `Alida` library includes helper classes with a flexible API for run-time configuration from the environment. In particular, it provides the class `ALDEnvironmentConfig` in the package `de.unihalle.informatik.Alida.helpers` to ease run-time configuration via the probably most common ways of individual configuration, i.e., in terms of environment variables and properties of the Java Virtual Machine.

The most general method, available in two versions, in class `ALDEnvironmentConfig` is

---
public static String getConfigValue(\
                    String prefix, String operator, String propname)

public static String getConfigValue(String operator, String propname)

These methods retrieve corresponding values from either environment variables or JVM

properties, in exactly this order. The strings passed as arguments to the routines are concatenated and properly formatted according to the usual naming conventions in `Alida` and the operating system, respectively (cf. Sec. 2.7). In detail, given a *prefix*, *operator* and *property*, the method is searching for an environment variable named `'PREFIX_OPERATOR_PROPERTY'`, or subsequently for a JVM property named `'prefix.operator.property'`. Note that in the `Alida` context the string `'prefix'` in the first method signature should usually be set to `'alida'`. In the second method, where the prefix argument is missing, this is done by default.

If the way how a certain property is specified, i.e., either as JVM property or environment variable, is known in advance, specific methods can be used:

---

public static String getEnvVarValue(String operator, String propname)

public static String getJVMPropValue(
                            String prefix , String operator, String propname)

public static String getJVMPropValue(String operator, String propname)

The first one just looks for environment variables, always assuming the prefix `'ALIDA'`. The other two methods are exclusively accessing JVM properties where the first call allows to specify a custom prefix, while the second one assumes `'alida'` by default.

Note that if for a certain requested property no configuration values are provided by any of these ways, all methods return `'null'` values and the requesting classes are supposed to fall back to default settings as defined by the programmer of an operator. In general, there is no limitation for an operator to access configuration variables of its choice. Usually they should just be properly documented in the Javadoc of the corresponding operator class.

The naming of environment variables and properties is not strictly regularized and left free to the programmer of an operator. However, it is strongly recommended to adhere to the `Alida` naming conventions as this helps to avoid name clashes. In particular, have your variables start with prefix `'alida'` and let the second part be the name of the class or operator using the variable. The third part is then the actual property. Make sure that new variables and properties do not collide with variables predefined in `Alida` which are listed in Sec 2.7.

A Graph-Visualization: `Chipory`

Processing histories are stored in XML format using *GraphML* with some `Alida` specific extensions as mentioned in Sec. 3.5. To display histories we extended *Chisio*[10] to handle the `Alida` specific extensions, yielding `Chipory`.

A.1 Installation and invocation of `Chipory`

`Chipory` is not strictly part of `Alida`, but supplied as an add-on at the `Alida` website[11]. A single zip-file is provided for running `Chipory` on Linux systems with 32- or 64-bit as well as on Windows systems. The only difference is one system dependent jar-file as detailed in the installation instructions provided in the zip archive. Essentially, all system-independent and one appropriate system-dependent jar-file have to be included into the CLASSPATH. Invoke `Chipory`, e.g., by

java org.gvt.ChisioMain [directory]

The optional directory supplied as an argument denotes the path where `Chipory` starts to browse when reading or writing files. If omitted, the current working directory is used.
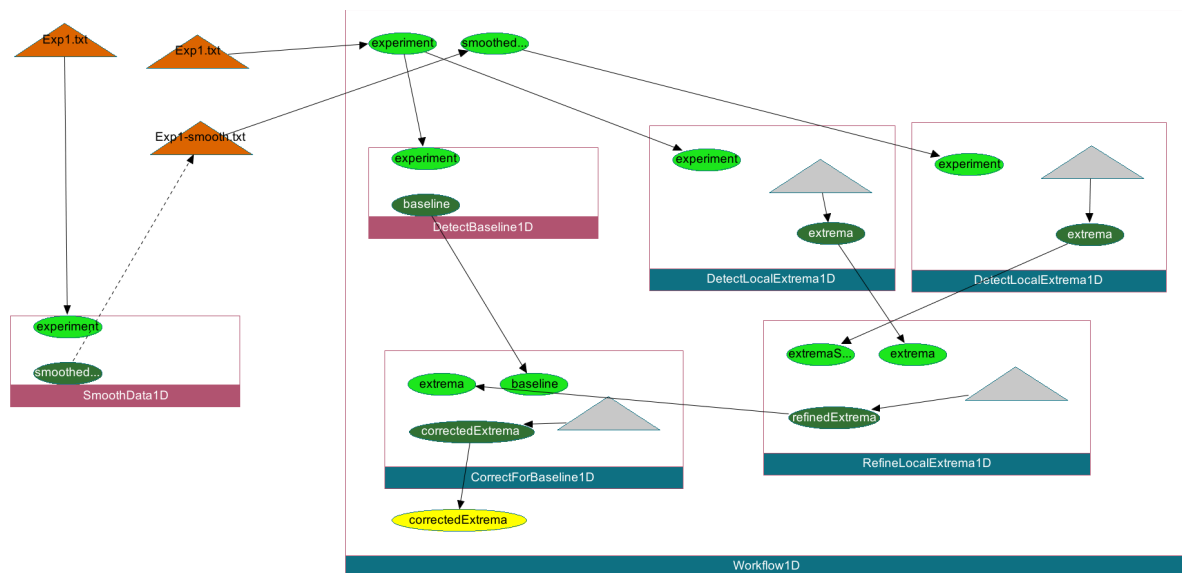

Figure 20: Example processing graph.

---

[10]http://sourceforge.net/projects/chisio
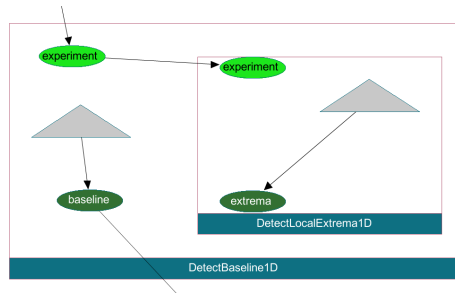[11]https://alida.informatik.uni-halle.de

Figure 21: Screenshot of `Chipory` for a part of the same processing graph as shown in Fig. 20, however, the collapsed instance of the `DetectBaseline1D` has been uncollapsed.

A.2   Using `Chipory`

`Chipory` is based on *Chisio*, a free editing and layout tool for compound or hierarchically structured graphs. In `Chipory` all editing functionality was conserved, however, is not required for inspecting a processing graph in virtually all cases. *Chisio* offers several automatic layout algorithms where `Chipory` chooses the Sugiyama as default as this is most adequate for the hierarchical graph structure of processing histories. In the following we explain a tiny part of *Chisio's* functionality and the extensions supplied by `Chipory`. For more details on *Chisio* see the User's manual of *Chisio* which is included in the `Chipory` package and also easily found in the web.

In Figure 20 an example processing graph extracted from a data analysis procedure based on demo operators is shown. As already described, instances of operators are depicted as rectangles, input and output ports as ellipses, and data ports as triangles. All three types of elements of a processing graph are implemented as *Chisio* nodes. A node may be selected with a left or right mouse click. A selected node may be dragged with the left mouse button pressed to manually adjust the layout. The size of a node representing operators is automatically adjusted to fit all enclosed ports and nested operators.

The name of an operator is displayed in a colored area at the bottom of its rectangle. If an operator node is uncollapsed it is shown in blue, if it is collapsed it is of dark red. This is shown in Fig. 20 where the operator `DetectBaseline1D` has been collapsed. A selected operator node may be collapsed or uncollapsed via its context menu or by a left double mouse click while pressing down the shift key. Collapsing makes all enclosed operator and data nodes invisible, thus, only the ports of a collapsed operator are shown. If the node is uncollapsed lateron, enclosed nodes are made recursively visible again, until a collapsed node is encountered. Uncollapsing additionally invokes the automatic layout algorithm, hence, any manual layout adjustments applied before are lost. If we uncollapse the operator `DetectBaseline1D` as shown in Fig. 21, we can inspect the data processing accomplished within this operator.

If a data port represents data read from file, the triangle is tagged by a string and colored
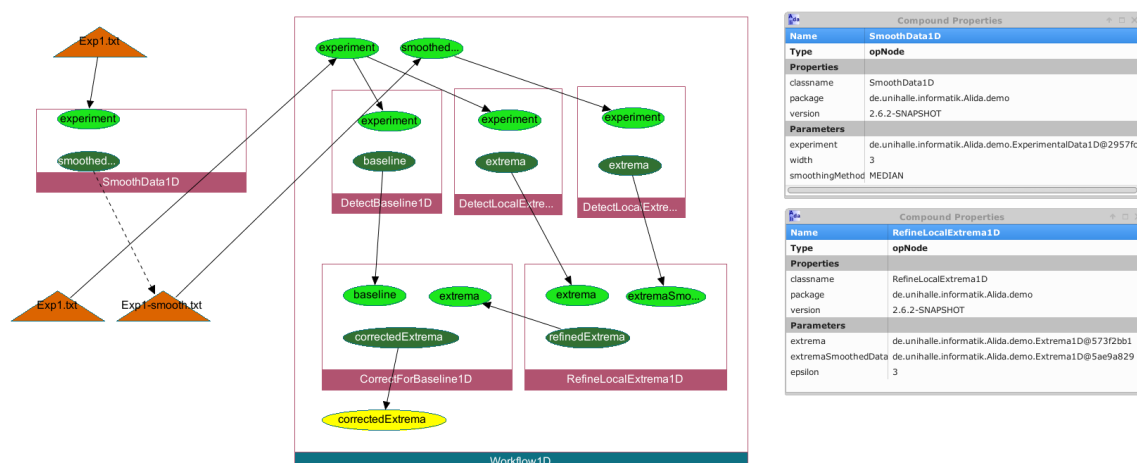
Figure 22: Screenshot of `Chipory` with details for the operators `SmoothData1D` and `RefineLocalExtrema1D`.

orange. As the data read from file `'Exp1-smooth.txt'` has a processing graph associated, this history is also included into the processing graph and connected by a dashed edge. This history was written to the file `'Exp1-smooth.txt.ald'` by previous processing using `Alida`.

Input and output ports are generally displayed with light and dark green ellipses, respectively. The single exception is the port for which the processing graph was constructed, which is depicted in yellow. In our example this is the output port `'correctedExtrema'` of the operator `CorrectForBaseline1D`.

More details for operators and ports may be inspected using the *Object properties* of *Chisio's* nodes. These are displayed in a separate window which for the selected node can be popped up using the context menu. The context menu is activated by a right mouse click. Alternatively the object properties window can be popped up by a double left mouse click.

Information displayed includes:

- name of the operator or port
- type of the node, e.g., `opNode` for operators
- for operators the parameter values at time of invocation
- for input and output ports the Java class of the object as it was passed into the operator along with the explanatory text of this port
- for output ports the properties of the object given when passed out of the operator, if it is of type `ALDData`.

In Fig. 22 this is shown for the operators `SmoothData1D` and `RefineLocalExtrema1D`.