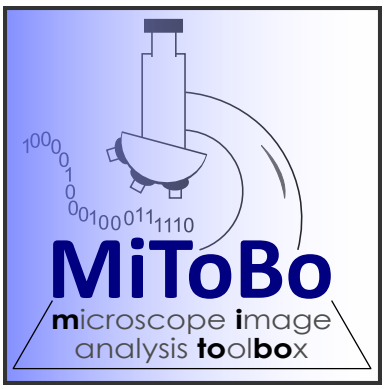
Martin Luther University Halle-Wittenberg
Institute of Computer Science
Pattern Recognition and Bioinformatics

User and Programmer Manual



MiToBo - Microscope Image Analysis Toolbox

Version 0.9.6

written by

The MiToBo Development Team

Markus Glaß Oliver Greß Danny Misiak

Birgit Möller Stefan Posch

Licensing information.

This manual is part of MiToBo - the Microscope Image Analysis Toolbox.

Fore more information on MiToBo visit http://www.informatik.uni-halle.de/mitobo/.

MiToBo is a project at the Martin Luther University Halle-Wittenberg.

Institution:

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

Contact: mitobo@informatik.uni-halle.de
Webpage: www.informatik.uni-halle.de/mitobo

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

Contents

Chapter 1

Welcome to MiToBo!

ImageJ[1] is a widely-used Java toolkit for image processing and analysis. Particularly in biological, medical and biomedical applications ImageJ (*Image Processing and Analysis in Java*) has gained large interest. ImageJ provides the user with a flexible graphical user interface, with a large variety of basic built-in image processing operations and also with a huge collection of optional plugins downloadable from the web. From a programmer's point of view, however, the ImageJ API provides less flexibility to support easy plugin and application development. Especially easy data access and exchange between different modules or plugins below the graphical user interface appear worth to be improved.

`MiToBo`, which is the ***Mi**croscope **I**mage Analysis **To**ol**Bo**x* developed at the Martin Luther University Halle-Wittenberg, tries to enhance ImageJ with regard to these aspects. On the one hand `MiToBo` provides implementations of common image analysis techniques useful for microscope image analysis. On the other hand, the implementation of these image processing techniques and algorithms is completely separated from any interface accessing the implementations, e.g. like ImageJ plugins or commandline tools, which offers largest flexibility to both users and programmers of the library at the same time.

`MiToBo` is build on top of `Alida`[2] which is a library for automatic documentation of data analysis procedures. In `Alida`, which is an acronyme for **A**utomatic **L**ogging of Process **I**nformation in **D**ata **A**nalysis, each data analysis pipeline is interpreted as a process of modifying given input data by a series of operations to produce the desired output data. Accordingly, for implementing data analysis and processing techniques the basic concept in `Alida` are 'operators' which manipulate data. It is straightforward to associate a selection of subsequent and/or parallel operator calls with a directed graph data structure, where each operator is linked to a certain node of the graph, and input and output data is passed from one operator to another along the graph edges. This graph together with the individual configuration parameters of all involved operators is a sufficient base for lateron reconstructing all steps of data manipulation for a given result data object. `Alida` allows to extract such 'history graphs' explicitly and by this offers a powerful tool for data analysis process self-documentation.

[1] http://rsbweb.nih.gov/ij/

[2] http://www.informatik.uni-halle.de/alida

MiToBo adopts the concepts of Alida. Consequently, image analysis procedures are realized in terms of operators which can be applied to data objects, e.g. images, regions or image primitives like keypoints and lines, and yield certain results. A set of parallel or subsequent operator calls results in a processing graph documenting the complete analysis process and, hence, releases developers and users of image analysis algorithms from cumbersome manual parameter logging and analysis documentation tasks.

MiToBo builds upon ImageJ image datatypes and does not interfere with ImageJ's plugin interfaces, i.e. allows to program ImageJ-compatible plugins based on the Alida operator concept. Thus, in addition to the ImageJ interfaces which focus on plugin and script development, MiToBo defines unique interfaces for the underlying image processing modules, i.e. operators, in terms of input/output data and parameters, and also with regard to the way how operators can be invoked from other operators or user interfaces. This significantly improves data exchange and operator handling, and establishes – besides the already mentioned concept of auto-documentation – a powerful fundament for adding new sophisticated features to ImageJ.

One of these possible extensions which soon might gain significant interest is the graphical programming of image processing applications based on ImageJ and MiToBo. The operator concept sets the first fundamental building blocks towards this direction since all operators clearly document the types of their input data, output data and parameter objects. This is essentially the most important basis for automatically combining operators to form processing chains, including data compatibility checks and the verification of operator configurations.

Altogether MiToBo in combination with the underlying concepts of Alida aims to provide programmers of image processing applications with a maximum of usability and ImageJ compatibility, while at the same time keeping the overhead for meeting the MiToBo operator specifications as small as possible. We hope that the new perspectives MiToBo opens with its concepts might be helpful for developers of image processing applications and by this further extend ImageJ's selection of valuable features.

1.1 Main Features: Operators, Plugins and History Graphs

Besides providing modern image analysis tools and algorithms for microscope image analysis the overall goal of the MiToBo project is to ease microscope image analysis in terms of the development of appropriate algorithms and flexible user interfaces. These interfaces should, of course, not only be designed for experts, but also for researchers using image processing software as a tool rather than developing their own algorithms. MiToBo builds on top of ImageJ which has achieved large success and broad acceptance by researchers from many different disciplines who need to solve their individual image analysis problems.

From the programmer point of view, ImageJ yields a suitable base for developing image analysis tools in an integrated framework. The programmer does not need to take care of e.g.

image display and zooming as ImageJ has answered such questions already. However, providing a certain degree of usability and easy integration of new algorithms in terms of plugins is only one side of the medal. On the other hand the underlying software structures and interfaces should also provide a sufficient degree of comfort to the programmer. In particular, image processing algorithms and user interfaces should be clearly separated from each other and data exchange between different modules should be easy in terms of well-specified interfaces.

As ImageJ is not optimally designed with regard to some of these aspects, `MiToBo` does not exclusively focus on the development of image analysis tools for microscope images, but also optimizes underlying software in terms of data flow and data structures. In addition, self-documentation of image processing pipelines is natively supported.

Based on the underlying concepts of `Alida` (Automatic Logging of Process Information in Data Analysis) `MiToBo` defines an image analysis pipeline as a sequence of operations subsequently or in parallel applied to data that is handed over from operator to operator. Such a pipeline may be viewed as directed graph structure, where the nodes are linked to different operators and the data flow is indicated by edges between these nodes. From this interpretation of image analysis in general, several concrete design issues are derived that are embedded in the `Alida` core of `MiToBo` and provide users as well as programmers with enhanced image analysis tools and an improved infrastructure.

Operator concept. The interpretation of image analysis pipeline as a sequence of operations directly leads to the concept of *operators* implemented in `MiToBo`. All manipulations that are performed on image data are done by operators. Vice versa operators are the only actors that work on given data, modify the data or generate new data entities from given input data. Accordingly, all image processing and analysis algorithms in `MiToBo` are implemented in terms of operators with clearly specified input and output interfaces.

The operator concept is adopted from `Alida`. Technically `Alida` defines a common superclass for data analysis modules denoted '`ALDOperator`' from which the `MiToBo` main operator class `MTBOperator` is derived and from which all implemented `MiToBo` operator classes need to be derived. Furthermore, the concept incorporates a formal description of the interface of an operator, i.e. a unique formal specification of its inputs, outputs, and parameters by annotations. In addition, there is only one possibility to invoke operators which is a single public routine to be called from user side (refer to Chap. 6 for more details on operators).

Self-documentation and History Graphs. The `Alida` operator concept with its transparent interface specifications allows a unified handling of operators in various contexts, e.g., with regard to graphical programming or automatic or semi-automatic code generation where compatibility checks and operator calls have to be standardized. In addition, the restriction of operator invocation to a single available method also serves as a basis for another sophisticated

feature of MiToBo which is the fully automatic documentation of image analysis pipelines. Given the interpretation of image analysis pipelines as sequences of operators, all that remains to be done for process documentation is to log the calls of all operators as well as their input and output data and save their parameter settings. Together with information about the order of operator calls as for example represented in a directed graph data structure these data form a complete protocol of the pipeline and allow for longterm documentation of analysis processes. In particular, linked to specific result data objects, they allow to reproduce all results ever produced during the course of algorithm development, testing, or experimental evaluation.

The concept of self-documentation is realized in Alida, thus, it is also an integral part of MiToBo and directly embedded into the operator concept. Operators provide internal functionality to store process data during the course of an analysis pipeline which later on can be extracted in terms of a *history graph* file in XML format. Such a graph is associated with each data object being the result of a certain operator or sequence of operations. Most of the time these objects will be images, however, processing histories can also be associated e.g. with segmentation results like regions or contours as well as histograms or any other numerical data. See Figure 3.1 for an examplary visualization of such a graph.

Altogether MiToBo, the *Microscope Image Analysis Toolbox*, yields an extension to ImageJ, which in particular provides programmers with enhanced functionality and unified interfaces for developing their image processing algorithms. In addition, users benefit from MiToBo's extensions towards an integrated documentation of all analysis pipeline and compatibility to ImageJ as underlying integrated software framework.

1.2 Installation

There are two common ways to work with MiToBo:

a) just using the plugins included in MiToBo for improving your work and easing image processing tasks (Sec. 1.2.1),

b) using the MiToBo API to write operators, plugins and other image analysis applications on your own (Sec. 1.2.2)

Depending on which use case you choose, and if you desire to compile MiToBo on your own or not, the installation instructions differ in some parts. More details about the necessary installation steps for both use cases can be found below.

1.2.1 Using MiToBo's plugins and tools

If you are mainly intended to use the MiToBo plugins and commandline tools without writing new code on your own, you just need to get the most recent binary archive including the MiToBo

packages and plugins from the download section of the MiToBo website[3]. Once you got the archive, please follow these steps:

1. extract the archive to a directory of your choice
2. copy the plugins' jar called 'Mi_To_Bo.jar' into your ImageJ plugins directory; you can specify a custom plugins directory for ImageJ by passing the option '-Dplugins.dir=<DIR>' to the Java virtual machine when starting ImageJ
3. include all jars from the archive and the MiToBo jar into your ImageJ plugins directory or into your CLASSPATH variable
4. download external jars required by MiToBo to the same folder and, if necessary, include them into the CLASSPATH; you will find a list of required jars on the webpage
5. start ImageJ and find the MiToBo plugins in the 'Plugins>MiToBo' menu (starting with prefix mtb_)

Note that the archive also contains script files for Linux and Windows to run ImageJ including the MiToBo plugins. After adaption of the file to the individual user's environment, plugins directory and CLASSPATH are set automatically, and ImageJ including MiToBo can be invoked by simply running the scripts.

MiToBo ImageJ toolbar macros. Since release 0.9.6 MiToBo supports the ImageJ toolbar, i.e. offers a configuration file for fast access of its plugins via a toolbar button. To add the MiToBo toolset to ImageJ, MiToBo is shipped with a macro configuration file. It can be found in the zip archives and is called MiToBo_Tools.txt. Simply copy the file into the folder macros/toolsets of your ImageJ installation and run ImageJ. Then you can select the MiToBo toolset from the list of available sets by clicking on the arrows button on the right side of the ImageJ standard toolbar. Selecting 'MiToBo_Tools' will add a button with the MiToBo logo to the toolbar allowing for direct access to all MiToBo plugins (Fig. 1.1).

1.2.2 Using MiToBo's operators and API

If you are interested in using MiToBo operators and its API directly in your own code to write new image processing modules or flexible self-documenting plugins and applications there are again two ways to do that. The more simple one is to consider MiToBo as a blackbox. In this case the installation instructions are essentially the same as in Section 1.2.1, i.e. you mainly need to

[3] http://www.informatik.uni-halle.de/mitobo

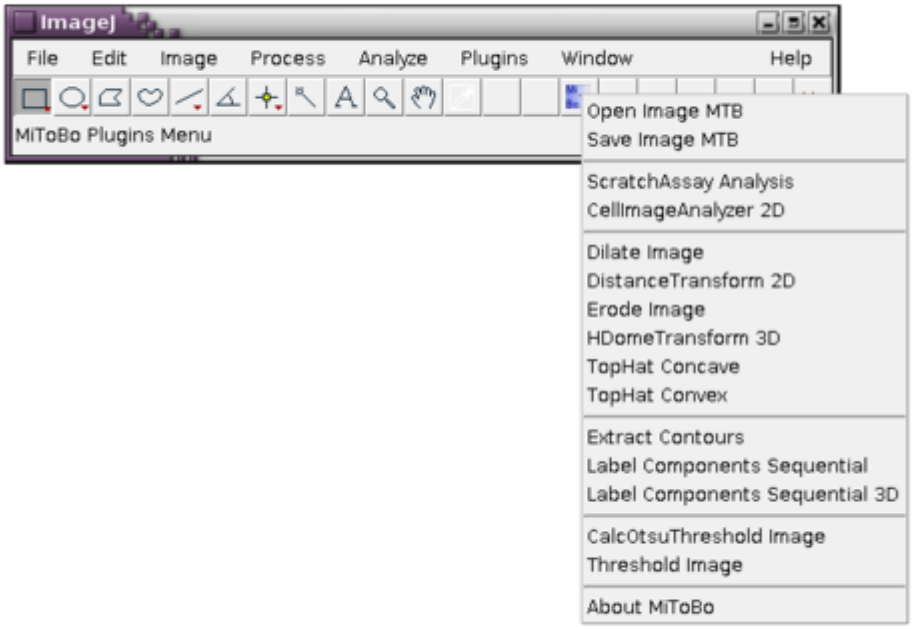


Figure 1.1: Screenshot of the `MiToBo` toolset menu in the ImageJ toolbar.

make sure that the `MiToBo` jar file and all additional jars are in your CLASSPATH when you compile and run code that uses `MiToBo` operators or datatypes.

Besides writing your own image processing applications based on the existing `MiToBo` packages and operators there might appear the necessity to extend the core or to adapt `MiToBo`'s functionality to your specific needs. Hence, it might be favorable to be able to compile `MiToBo` on your own. In this case you should download the `MiToBo` zip file including the source files from the webpage of `MiToBo`. Extract the file to a directory of your choice (to which we will refer to as root directory '$MITOBO' in the following). Explanations regarding the contents of the zip file can be found in Appendix B. The sources can be compiled either using 'ant' or within integrated development environments like 'eclipse'.

Compiling `MiToBo` using 'ant'. To compile `MiToBo` with the Java tool 'ant', simply run 'ant' from the root directory of your `MiToBo` installation.

Compiling `MiToBo` using 'eclipse'. If you prefer to use an integrated development environment (IDE) like 'eclipse' or 'netbeans', you need to create a new Java project by importing the `MiToBo` sources extracted from the zip file.

1.2.3 Native libraries

Some parts of `MiToBo` rely on native C++ library code. The implementation of explicit active contours (commonly denoted as *snakes*) is in parts based on the `CGAL Computational Geometry`

Algorithms Library[4], i.e. on some few classes of the CGAL polygon package. These classes are released under the GNU Lesser General Public License in version 2.1. For the ease of usage MiToBo is shipped with native libraries including binaries of these classes which need to be available when running related MiToBo plugins and operators. You can find the native libraries for different architectures in a lib directory in the binary and source archives of MiToBo. To make them available to ImageJ on linux machines copy them to a directory of your choice and add this directory to the environment variable LD_LIBRARY_PATH, on windows machines copy the corresponding .dll file into the system folder containing dynamic libraries.

1.3 About this Manual

This manual is organized in a short introductory section and two main parts. The introductory section – which you most probably have already read in the last few minutes – is dealing with some general remarks and basic installation notes, while the two main parts provide more details. The first part is dedicated to users who are mainly interested in using MiToBo plugins or commandline tools for their own work, e.g., to benefit from the automatic documentation capabilites of MiToBo or to simply adopt provided image analysis algorithms for own image processing tasks. The second part introduces the reader to some more internals of MiToBo, i.e. it provides the reader with more details about the operator concept and how to use it with own code, more information about specific MiToBo data types and also about programming with MiToBo in general. Note that only basics of the underlying Alida operator and self-documentation concept are presented. For more details about its functionality and technical implementation refer to the Alida manual to be found on the Alida webpage http://www.informatik.uni-halle.de/alida.

If anything remains to be clarified or if you have further notes and comments, just write an email to us at mitobo@informatik.uni-halle.de. We are happy to get in touch with you!

[4] http://www.cgal.org/

Part I

MiToBo: The user's view

Chapter 2

MiToBo Quick Start Guide

`MiToBo` brings a couple of image analysis plugins, basically developed for microscope image processing and analysis. After `MiToBo` was successfully installed (cf. Chap. 1.2) you can use the toolbox plugins like any other ImageJ plugin. To make use of the self-documentation to its full extent, it is indispensable to use the I/O plugins of `MiToBo` to open input data and save the resulting output data. During the image analysis process, a processing pipeline is generated, simply by using the included toolbox plugins. To get a complete self-documented history graph of the image process (refer to Chap. 3 for more details), store the desired output data via the provided `Save_Image_MTB` plugin.

For the basic use of `MiToBo` and its plugins, follow the steps mentioned below:

1. open the input data using the `Open_Image_MTB` plugin
2. use one or more plugins of the toolbox, to generate an image processing pipeline
3. save the resulting image(s) via the `Save_Image_MTB` plugin to store the output data and get a full history of the image process
4. use `chipory` (see Appendix A) to view the history graph and get an overview of the processing steps, the data flow from the input to the output data and the applied parameters

At the moment, self-documentation is only supported by using plugins from `MiToBo` itself. Maybe it will be possible in the near future to use ImageJ plugins as well to get a self-documented image analysis process, due to the developments in ImageJ 2.0[1]. Furthermore, custom plugins can be implemented with support for self-documentation (cf. Chap. 8).

[1] http://imagejdev.org/

Chapter 3

History Graphs

One of the main features of `MiToBo` is its capability of self-documenting image processing pipelines. The operator concept allows to get a detailed internal log of all data manipulations, which can subsequently be used to convert the process history into a directed graph data structure denoted *history graph* in the following.

The `MiToBo` operator concept defines operators as the only places where data are processed and manipulated. Each operator receives a number of input objects, which for example may be images or segmentation results like regions. The behaviour of an operator is controlled by parameters, where typical examples are the size of a structuring element or a threshold. An operator produces output data, in particular images, but also for example numerical data, regions or contours.

In `MiToBo` an image analysis pipeline 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. The invocation of operators can be of pure sequential nature or subsume parallel processing steps. In addition, nested application of operators is possible. Given this principle each analysis pipeline and its data flow may be interpreted and visualized as a directed acyclic graph (cf. Fig. 3.1 for an example).

A `MiToBo` history graph basically consists of operators and data nodes which are connected by edges indicating the flow of data. Within the graph each operator is depicted as a rectangle with the operator's classname in the bottom line, as can be seen from Fig. 3.1 showing a screenshot of `chipory` (see Appendix A). For each input and output data object the operator 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. In Fig. 3.1 the result image 'resultImg' produced in the `MTBMedian` operator is e.g. handed over to the `ActiveContours` operator as well as returned directly to the calling

operator `CellSegmentation`. Each port of an operator has an individual name indicating the input or output object associated with the port. This allows to distinguish between ports if one operator defines multiple input ports as is the case for the `ActiveContours` operator.

In addition to operators and ports there are also data nodes in the graph corresponding 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 with light grey. In Fig. 3.1 two data objects are created outside of the processing pipeline as a result of reading images (at the top of the figure) and are passed as input data objects to the `Cell_Segmentation` plugin. Additionally, three more images are created by the operators `MTBMedian`, `MTBOtsuThresholding` and `ActiveContours` which in all three cases form the resulting data objects of these operators and are passed to the outside via output ports.

Fig. 3.1 shows the history graph for the output object 'resultImg' of the operator `Cell_Segmentation`, where the corresponding port is shown as a yellow ellipse at the bottom of the figure. This history subsumes the calls of seven operators in total where some of these calls are nested. The outmost operator is `Cell_Segmentation` which is implemented as a `MiToBo` plugin, indicated by the underscore in its name (cf. Chap. 8). This plugin calls the `CellSegmentation` operator implementing the actual algorithms. For cell segmentation two input images are required whereas one of these images is median filtered by `MTBMedian` while the second one is fed into the `DetectNuclei` operator. Inside of that operator first `MTBOtsuThresholding` is called, and the binary result image is subsequently post processed applying `MTBFillHoles`. Its result is handed back to the calling `DetectNuclei` operator and also directly propagated further back to the `CellSegmentation` operator. This operator finally calls the `ActiveContours` operator which generates one of the two result images of `CellSegmentation`. The second result image is the median filtered image which is also returned to the calling plugin as mentioned above.

The history data is stored in XML format using *graphml* with some `MiToBo` specific extensions in a file accompanying the actual data object file. When reading and writing images using `MiToBo`'s Open_Image_MTB and Save_Image_MTB plugins, history files are automatically considered. For example, for an image stored in the file '`example.tiff`' its history data is automatically saved to the accompanying file '`example.ald`'. The extension '`.ald`' indicates a `MiToBo` *processing history* file and in fact is derived from `Alida`, which is responsible for processing history in `MiToBo`. When later on reading the image using Open_Image_MTB, `MiToBo`'s open operator checks for an accompanying file, and if one is found it is read and the corresponding history data is linked to the image object. This allows to trace the processing history of an object in the long run and even when the processing pipeline was interrupted by intermediate savings to disk.

Note, the identity of images is *not* preserved in the processing history across file boundaries. If two (or more) input images for the current top level operator, which is implementing the plugin functionality (in Fig. 3.1 this would be the operator `Cell_Segmentation`), are loaded from the

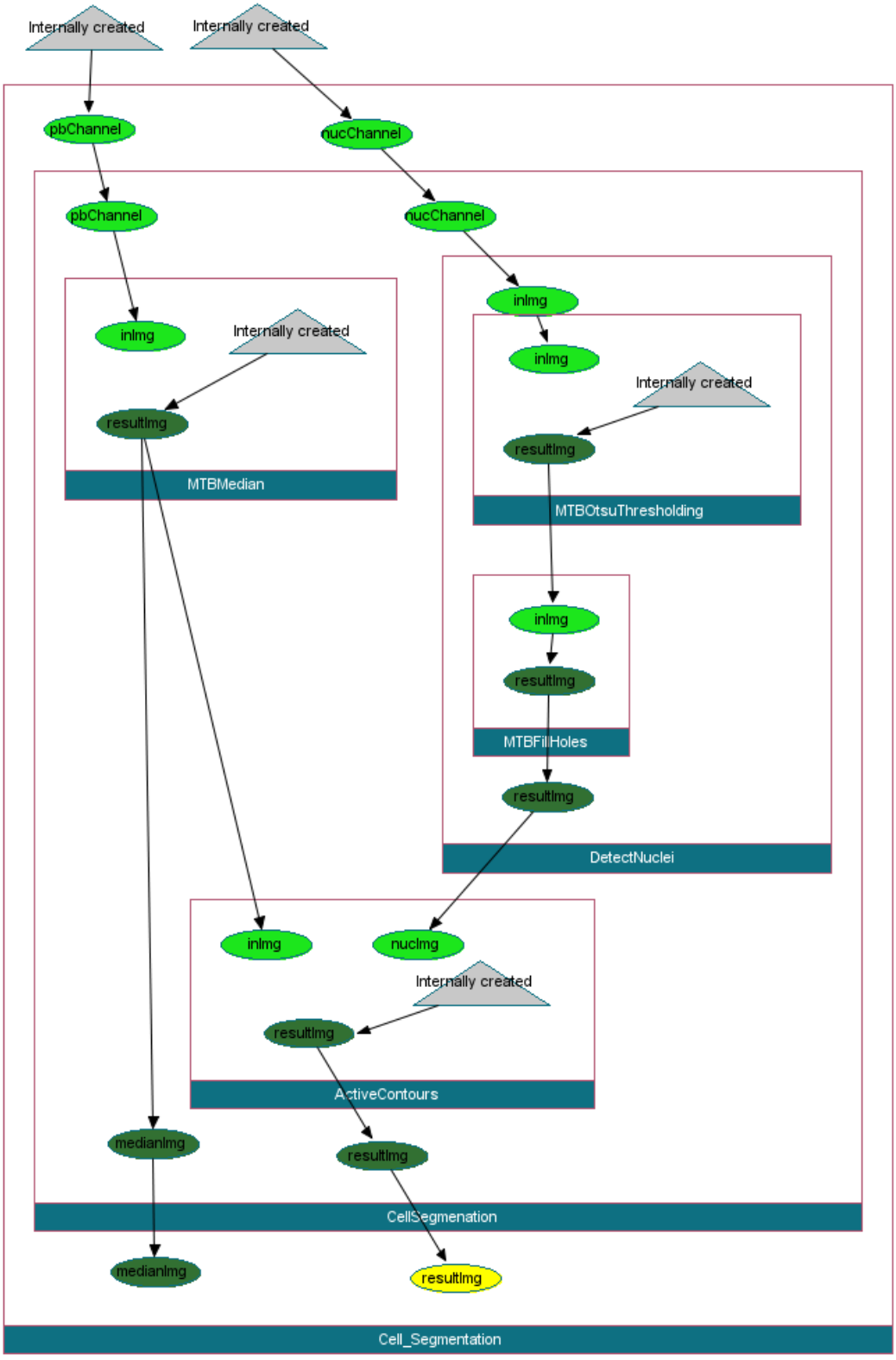


Figure 3.1: A `MiToBo` history graph: the directed acyclic graph represents the application of nested operators. Operators are depicted as rectangles, input and output ports as ellipses filled in light or dark green, respectively. The grey triangles relate to newly generated data objects, and the yellow ellipse indicates the result data object to which this history graph is linked to.

same image 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.

Chapter 4

Important notes

Wrapper datatypes. `Alida` does only have some few restrictions on which data types are allowed as input and output parameters of operators. These restrictions are due to the implementation of the *port hash*, which stores the connection of data objects to input/output ports of operators. Further information can be found in Sec. 7.1. For the concept of the hash it is essential that objects can uniquely be identified during the run of a processing pipeline. The only unique object identifier in Java that can be used for this purpose is an object's reference. As a conseqeunce, input and output of operators may be of any Java class, which is uniquely identifiable, which excludes only primitive data types, interned strings, and cached numerical objects.

For Java data types not allowed as input and output data datypes wrapper classes have to be used instead. `MiToBo` already implements some wrapper classes for frequently used native datatypes. In particular, in '`MiToBo.datatypes`' you can find wrapper implementations for the following Java data types:

- `java.lang.Double`
- `java.lang.Integer`
- `java.lang.String`

MTBImage and ImagePlus. Inputs and outputs of operators are usually individual objects that can uniquely be identified given their reference, and that are independent of any other object being manipulated within the same processing chain. One exception from this rule, however, is given by image objects. In particular, although `MiToBo` defines its own image types, the implementation of these types depends on ImageJ datatypes as far as possible. The main reason for this is compatibility with the graphical user interface of ImageJ which should be accessible to `MiToBo` plugins and operators without cumbersome casting. The detailed description of `MiToBo`'s

'MTBImage' is to extensive and not required in the following part. Thus it is postponed to Sec. 10.3.

As as consequence of these links between 'MTBImage' and 'ImagePlus' objects of both types cannot be seen completely independent of each other. Conversely, in most cases an image object present in ImageJ's graphical interface will have an associated MiToBo image object in the underlying processing chain, and vice versa. Thus, the same image might one time be handed over to an operator as object of type 'ImagePlus', and another time as object of type 'MTBImage'.

For proper documentation of the image's processing history it is essential to associate all operator calls on both instances of the physical image with the same underlying object in the history database. To this end image objects are internally wrapped into a box object that explicitly links corresponding objects of type 'MTBImage' and 'ImagePlus' to each other and allows for correct tracing of the history independent of the concrete image object instance passed to an operator. Please read Sec. 10.3 for proper use of images in MiToBo to prevent data inconsistencies between corresponding 'MTBImage' and 'ImagePlus' objects.

Chapter 5

Configuring MiToBo

Several of MiToBo's plugins support individual configuration by the user. For example initial files or directories the plugin should work on can be specified by the user. The probably most common way of individual configuration is to pass specific path or flag settings to MiToBo plugins by environment settings as outlined in the next section.

5.1 Environment variables and properties

MiToBo plugins support three 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

c) ImageJ preferences as specified in the file ~/.imagej/Prefs.txt

This order reflects the priority of the three options, i.e. environment variables overwrite JVM properties, and the latter ones overwrite ImageJ preferences. If for a certain plugin no configuration values are provided by any of these three ways, default setting of corresponding internal variables is completely plugin dependent.

In general there is no limitation for a plugin to define configuration variables. Usually they should be properly documented in the Javadoc of the corresponding class. Some variables of general interest, however, are listed below as almost all users might be interested in using them.

The naming of the environment variables and properties is also not strictly enforced. However, it is strongly recommended to adhere to the MiToBo naming convention as this helps to avoid name conflicts. In MiToBo all variables start with prefix 'MITOBO' (which is automatically added to the variable name by the library fuctions). The second part of the name is usually the plugin using the variable, and the third part is the actual variable.

Example:
Imagine a plugin called 'Dummy_Plugin' which defines a variable 'Input'.
The environment variable that will be checked by the plugin is then denoted by:

`MITOBO_DUMMY_PLUGIN_INPUT`

Following ImageJ property naming conventions the
corresponding preference and also the JVM property is denoted by:

`mitobo.dummy_plugin.input`

Besides plugin specific variables there may exist variables of global interest shared by different plugins. In their names the second part is simply missing, like in

`MITOBO_IMAGEDIR / mitobo.imagedir`.

When defining such variables, however, special care has to be taken for ensuring that such variables are interpreted the same wherever they are used. And even more important, it needs to be thoroughly verified that the variables were not already defined elsewhere which might result in strange behavior of certain plugins.

5.2 List of Important variables and properties

Below you find a table listing variables and properties of presumably common interest.

Plugin(s)	Variable	Meaning
	MITOBO_SVNROOT	Path to local SVN checkout, optionally used to get release information in self-documentation.
Open_Image_MTB Save_Image_MTB	MITOBO_IMAGEDIR	Directory where images are expected. For example checked if the following two variables are not set.
Open_Image_MTB	MITOBO_OPENDIR	Directory where browsing starts the first time.
Save_Image_MTB	MITOBO_SAVEDIR	Directory where browsing starts the first time.

Table 5.1: List of Important variables and properties

Part II

The programmer's view

The following two Chapters are taken directly from the `Alida` Manual and included for convenience of reading.

Chapter 6

Alida operators

The heart of Alida's concept are operators that implement all data analysis capabilites. In this chapter we discuss the operator interface, how Alida operators can be invoked from self-written code, and finally how new operators can easily be implemented.

6.1 Data processing and operators

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 operators is expected to operate 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 a subtype of processing, e.g. should experimental measurements be summarized by their mean or their median, a mask-size of a filter to be applied to an image, or 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 a input or as a parameter. Therefore, Alida does not destinguish between input data and input parameters. A parameter of an operator may be optional.

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 the direction of the parameter, which obviously may be input or output. In cases,

where an input object is destructively modified, this parameter has the direction input and output. An example is a vector which is modified in place.

In addition to parameters giving the input data and configuration of the operator and output data representing the results of processing, an operator may use supplemental parameters. Examples include flags to control output or debugging information and intermediate results produced by an operator. 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.

The collection of all parameters is called the *interface* of the operator. Each operator defines its interface by Java annotations of the member variables which constitute the interface. Currently a modified version of the annotation `@Parameter` as under developement for ImageJ Version 2[1] is used. The relevant features defined for the `Alida` operator concept are

- the direction of the parameter, which may be IN, OUT, INOUT,
- a boolean indicating whether the parameter is supplemental,
- a boolean indicating whether the parameter is required or optional (and which is only interpreted for non-supplemental IN and INOUT, parameters)
- a label,
- a textual explanation of the parameter,
- a data I/O order by which parameters can be ranked for generic GUI or commandline interface generation, and
- an expert mode which, e.g., allows to hide parameters for advanced configuration from non-expert users.

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 is not be part of the data flow all classes are acceptable.

The application of operators may be nested as one operator may call one or more other operators. At the top of this hierarchy we have commandline tools or plugins. Their parameter settings are facilitated via files, GUIs, commandline or the console.

6.2 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 following method:

[1] http://imagejdev.org/

```
public final void runOp(boolean hidden)
  throws ALDOperatorException,ALDProcessingDAGException
```

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 would only be feasible from within the package of the operator.

```
1   CellSegmenation cellSegmenter = new CellSegmenation();
2
3   cellSegmenter.setVerbose( true);
4   cellSegmenter.sigma = this.sigma ;
5   cellSegmenter.maxIter = this.maxIter ;
6   cellSegmenter.gamma = this.gamma ;
7   cellSegmenter.nucChannel = this.nucChannel ;
8   cellSegmenter.pbChannel = this.pbChannel ;
9
10  cellSegmenter.runOp();
11
12  this.resultImg = cellSegmenter.resultImg;
13  this.medianImg = cellSegmenter.medianImg;
```

Figure 6.1: Example how to configure and invoke an operator.

An example of how to use an operator is given in Fig. 6.1. First a new instance of the operator is created, and subsequently input parameters and parameters are set. In this toy example the member variables are mainly assigned directly, typically, however, setter methods will be supplied by the operater, as is shown for the method `setVerbose()`. If all required input data and parameters have been assigned for the operator object, it can be invoked calling its `runOp()` method. This is the only legal way to invoke processing for an operator as this method takes care of the construction of the processing history. Upon invocation of `runOp()` the validity of parameters and inputs is checked. Validity requires for an operator that all required parameters and input parameters have values different from '`null`'. In addition the implementation of an operator may impose further constraints by overriding the method

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

which, e.g., may restrict the admissible interval of numerical parameters. Subsequent to successful validation the method

protected abstract void operate()
throws ALDOperatorException,ALDProcessingDAGException

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 as in the example, or by getter methods as implemented by the 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.

The method `runOp()` takes an parameter with name `hidden` of type `boolean`. If `hidden` is `true` then the invocation of the operator is hidden from the processing history. See Section 7.3 for more details.

An operator object may be reused to invoke processing several times, where inputs and/or parameters may be changed between subsequent calls of `runOp()`.

6.3 Implementing operators

To supply a uniform interface for applying operators which automatically takes care of the processing history, each operator is implemented by extending the abstract class `ALDOperator`. There are four issues which have to be taken care of when implementing an operator, namely

- the interface of the operator,
- the operation per se,
- constructors,
- and to indicate whether this operator prefers a complete processing history or a processing history according to data dependencies,

which are described in the following.

Operator interface. As mentioned, the interface of an operator is constituted by the parameters as well as input and output data. In addition, an operator may use supplemental parameters, e.g., to define variables to control output or debugging information as well as return of intermediate results. The output of an operator is expected to be independent of the values of these supplemental parameters. Hence, these are not stored in the processing history. The supplemental parameters are described in analogy to parameters.

Each operator needs to define its parameters by annotation of the corresponding member variables. Currently a modified version of the annotation `@Parameter` as under developement for

ImageJ Version 2[2] is used. For `Alida` only the attributes `required`, `direction`, `supplemental`, `label`, and `description` are used.

The field `description` of the parameter gives a textual explanation and the `label` may be used for display purposes. For `IN` and `INOUT` parameters the field `required` defines whether this parameter is required or optional. Via the Java inheritance mechanism an operator inherits all parameters defined in its super classes.

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.

For generic generation of user interfaces labels and descriptions of annotated parameters are used. Given unique labels parameters can be referenced from graphical user interfaces or as commandline arguments in console by their labels. The description is used in tool tips and in conjunction with help options to provide basic support for users. In addition, 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 commandline tools some parameters might be supposed to appear earlier in the help system than others. Such a ranking can be achieved by specifying an order. Smaller values refer to a high importance of the corresponding parameter, larger values to minor importance.

Finally `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`. `Alida`'s graphical operator runner allows to switch the view of parameters between standard and advanced accordingly.

Fig. 6.2 shows an example how descriptors are defined. Here the operator defines one required input and one output parameter of type `Image`. In the example the parameter declarations are shown in lines $3 - 10$.

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 appropriate setter- and getter-methods. To invoke the processing within an operator, i.e. to run its `operate()` routine, the final method `runOp()` needs to be called by the user of an operator.

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

[2] http://imagejdev.org/2011/04/01/imagej-v200-alpha1

```
1  public class MTBMedian extends MTBOperator {
2
3      @Parameter( label= "inImg", required=true,
4        type = Type.INPUT, description = "Input image")
5      Image inImg;
6
7      @Parameter( label= "resultImg",
8        type = Type.OUTPUT, description = "Result image")
9      Image resultImg;
10
11   public MTBMedian() throws ALDOperatorException {
12   }
13
14   protected void operate() throws ALDOperatorException {
15     System.out.println( "MTBMedian::operate");
16
17     //Image in = (Image)(this.getInput( "inImg"));
18     Image res = new Image();
19     // no actual work yet, add your operator code here
20     resultImg = res;
21   }
```

Figure 6.2: Example how to define the interface and standard constructor of an operator, as well as the more or less empty `operate()` method.

may be used to augment data objects, e.g., by a filename or URL specifying the origin of the data.

Constructors. For automatic code generation and documentation capabilites as well as generic execution of an operator the operator class needs to implement a public standard constructor as shown in the example, line 12. This is, however, not necessary if the operator is only to be used explicitly by the programmer as shown in example of Fig. 6.1. Further convenience constructors may be implemented which additionally set parameters.

Important note: It is strongly recommended that an operator does not rely on initializations, e.g. of private fields, that are performed in a 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 (see next paragraph).

Annotations. An operator may be annotated with the `@ALDAOperator` annotation provided by `Alida`. Some functionaliy incorporated into `Alida` requires this annotation to identify the class as an `Alida` operator. As an example consider the generic commandline tool and its graphical counterpart which initially register all available operators according to this annotation, see Chap. **??** for more details. If an operator is annotated with `@ALDAOperator` a public standard constructor has to be supplied, otherwise a compiling error will result. Furthermore, abstract classes can not be annotated with `@ALDAOperator`.

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

- `NONE`, to prohibit generic execution completely
- `SWING`, to allow generic execution via GUI only,
- `CMDLINE`, to allow generic execution via commandline only, and
- `ALL`, to allow for generic execution in general.

Furthermore operators can be categorized in the two classes `STANDARD` and `APPLICATION`. 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.

Preference for 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`, which works in all cases, but potentially includes further 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 7 for details.

Helpful tools for operator development There is a tiny commandline tool to query the interface of a compiled operator by invoking the tool on the operator's class, e.g.,

```
1 java de.unihalle.informatik.Alida.tools.PrintOperatorInterface
2                         de.unihalle.informatik.MiToBo.morphology.ImgDilate
```

from the commandline. The above example yields

```
1  Interface of ALDOperator: ImgDilate
2  Required input parameters
3    Parameter <inImg> (IN) required, type: MTBImage,
4                                   label: Input Image, (Input image)
5    Parameter <masksize> (IN) required, type: int,
6                                   label: Masksize, (Masksize)
7  Output parameters
8    Parameter <resultImg> (OUT) required, type: MTBImage,
9                                   label: Result Image, (Result image)
10 Supplemental parameters
11   Parameter <verbose> (IN supplemental), type: Boolean,
12                                   label: Verbose, (Verbose flag)
```

Another commandline tool, `de.unihalle.informatik.Alida.tools.GenerateGetterSetter`, prints standard getter and setter methods onto standard output which can easily be included into the Java source code if no IDE for code development is used.

Chapter 7

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 (Chapt. 6), 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 calls directly to the manipulating operators. These links essentially form the base to later on build the history graph representation for each object ever seen by any of the `Alida` operators during a processing chain.

7.1 Basics of the history concept

The key to log all operator calls 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 build 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 call 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 has to take care of the data stored in the port hash or the correct logging of operator calls. 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.

7.2 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* [1]. It is based on the XML schema definition of *GraphML* [2] 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:

[1] http://xmlbeans.apache.org/
[2] http://graphml.graphdrawing.org/

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

7.3 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(true)` as mentioned in Section 6.2. This effectively excludes this invocation from any processing graph for an object which indeed depends on this operator invocation. One example where this might be adequate is a tool for interactively choosing and applying a global threshold on an image. Here a thresholding operator will be successively invoked for varying thresholds until the user is satisfied with the result. If all operator invocations would be documented in the history graph it would be cluttered with intermediate thresholding operations. This hiding of operator invocation can be ignored as the processing graph is created using the static method `createGraphmlDocument()` of class `ALDProcessingDAG` as described in the class documentation, e.g. for debugging purposes.

The second mechanisms 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 calls of further operators are also included into the graph unless the invocation was hidden. For example, in the processing graph shown in Fig. 3.1 including the `ALDOpNode` for `CellSegmentation` will recursively also include the nested operator invocations of `MTBMedian`, `ActiveContours`, and `DetectNuclei`. Considering the object returned by `CellSegmentation` via the output port `resultImg` this is adequate without question, as is evident from the data flow depicted.

However, the situation is different with regard to the object returned via the output port `medianImg` as its value does not depend on the manipulations of the operators `ActiveContours` and `DetectNuclei`. In this case 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. The history will then only include the invocation of `MTBMedian`, but not `ActiveContours` and `DetectNuclei`.

A typical example where data dependencies are not suited to yield a sensible history graph is the operator `MTBMedian`. The data dependencies indicate a dependency of the output object from an internally created data port, however, do not reflect the dependency of the input image. This is indeed not possible if `MTBMedian` is to not modify its input data and, thus, returns the filtered image in a newly created data object. This example shows, that the generation of a processing graph will yield a sensible history in very rare cases using the mode `DATADEPENDENCIES`.

Therefore 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`. If this variable is set to `false` for the operator `CellSegmentation` and the history is constructed in mode `OPNODETYPE`, the history for the object returned via the port `resultImg` will include all three nested operator invocations. Contrary, the history for the object returned via `medianImg` will only contain the invocation of `MTBMedian`. 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.

7.4 Software version handling

Documenting the processing history for data items requires not only to log all operator calls and their parameter settings, but also to remember the software versions of the operators. Consequently the method `runOp()` retrieves upon invocation the current software version of an operator. Indeed, where this version is queried from needs to be specified by the user. Popular options are for example SVN, CVS or Git repositories, 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`

which all version provider classes have to extend. `ALDVersionProvider` mainly defines the method

```
public String getVersion()
```

returning a string object containing the software version. The concrete implementation of `ALDVersionProvider` to be used for version information retrieval can be specified at runtime by JVM properties or environment variables (cf. Section 5). 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 the standard error channel and no version information is added to the generated documentation.

Internally, a factory named `ALDVersionProviderFactory` extracts the desired implementation from the given environment variables or JVM properties and creates corresponding objects. Note that by default a dummy version provider is initialized, i.e. the version is always set to *"unknown"*. Anyway, as default implementation `Alida` supplies the programmer with class `de.unihalle.informatik.Alida.version.ALDVersionProviderCmdLine` which allows reading version data from the environment. The class extracts version data from another JVM property named `version`. Hence, envoking `Alida` processes with the following options,

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

will insert the version ID *"4711"* into extracted history graphs.

Chapter 8

Implementing plugins

The ImageJ plugin concept is a very powerful concept to access the full ImageJ and third-party APIs. Based on this concept, the whole `MiToBo` API is also accessible via plugins. This provides a huge range of flexibility to use common plugins as well as special developments and algorithms for image analysis and processing.
The implementation of plugins in `MiToBo` is very easy and is effected according to the conventions of ImageJ. Writing your own plugins is possible in one of the following two ways:

A) as standard ImageJ `PlugIn` / `PlugInFilter`

B) as `MiToBo` `MTBOperatorPlugInFilter`

Important note: For both ways the ImageJ rules take effect. Only **.class** and **.jar** files in the '$MITOBO/plugins' folder with at least one underscore in their name will be accessible from the graphical user interface of ImageJ.

8.1 Implement `PlugIn` or `PlugInFilter`

The implementation of plugins in this way follows the same rules as mentioned in the standard ImageJ plugin development[1]. In principle two types of plugins are supported:

`PlugIn` - do not require an image as input
`PlugInFilter` - require an image as input

Both types of plugins can easily be combined with `MiToBo` operators (cf. Chap. 6) and data types (cf. Chap. 10). To use an operator, a new operator object has to be instantiated and input data and parameters have to be set. Then the operator can be invoked by its `runOp()` method where the results can be retrieved from the operator object after returning from this method. Figure 8.1 shows a small example plugin using the `MTBMedian` operator.

[1] http://www.imagingbook.com/index.php?id=102

```
1  import ij.ImagePlus;
2  import ij.plugin. filter .PlugInFilter;
3  import ij.process.ImageProcessor;
4  import de.unihalle.informatik.MiToBo.operator.*;
5  import de.unihalle.informatik.MiToBo.datatypes.images.MTBImage;
6
7  public class MTB_Median_IJPlugin implements PlugInFilter {
8      // plugin input image
9      private MTBImage inputImage;
10
11     // Implementing standard ImageJ setup method.
12     @Override
13     public int setup(String arg0, ImagePlus imp) {
14         // create a new MTBImage data type object from the ImagePlus input image
15         this.inputImage = MTBImage.createMTBImage(imp);
16         // allow 8 and 16-bit gray value images
17         return DOES_8G + DOES_16;
18     }
19
20     // Implementing standard ImageJ run method.
21     @Override
22     public void run(ImageProcessor impIP) {
23         // create a new median operator object and set the input via its
24         // constructor
25         MTBMedian medianOp = new MTBMedian(this.inputImage);
26         // set input and parameters explicit if no constructor exists
27         // MTBMedian medianOp = new MTBMedian();
28         // medianOp.setInput( "inImg", this.inputImage);
29
30         // invoke the operator
31         medianOp.runOp(false);
32         // get the output of the operator, in this case the filtered image
33         MTBImage resultImage = medianOp.getResImage();
34         // display the filtered image via ImageJ
35         resultImage.show();
36     }
37 }
```

Figure 8.1: Example how to implement a standard ImageJ plugin using an `MiToBo` operator.

This way of plugin implementation should be used if the plugin uses only operators to modify and create data, i.e. that the processing history reflects the complete chain of operations.

Important note: Using this kind of plugin implementation, the plugin itself is not included in the history graph, but nested operators are included. To also include the plugin itself, it needs to be implemented as operator.

8.2 Implement `MTBOperatorPlugInFilter`

Here a plugin can be implemented as `MiToBo` operator (cf. Chap. 6). The plugin extends the abstract class `MTBOperatorPlugInFilter`, which implements the `PlugInFilter` class of ImageJ, and overwrites the `setup()` and `operate()` method, as well as the ImageJ `PlugInFilter run()` method. Being an operator, the plugin must define its operator descriptors (cf. Sec. 6.3).

Important note: An input image of the plugin is implicitly defined in the `MTBOperatorPlugInFilter` class. By default, this input image is stored by an `MTBImage` object reference named `mtbInput`.

The input data and parameter settings of the plugin are located in the `run()` method. To use other operators inside the plugin, these operators should be instantiated in the `operate()` method and invoked by their `runOp()` method. Afterwards, the results can be retrieved from the operators and the plugin output can be set. Finally these additional operators are called via the `runOp()` command within the plugin `run()` method. After returning to `run()` the results of the plugin can be processed or displayed.
Figure 8.2 shows a small implementation example of a plugin, implemented as `MiToBo` operator. To show only the basic implementation, the plugin only passes its parameters to the median operator.

Important note: Using this way of implementation, the plugin appears as an operator in the history graph.

This way of implementation should be used, if the plugin passes its parameter to more than one operator or modifies the input image in some other way.

```
1  import ij.ImagePlus;
2  import ij.process.ImageProcessor;
3  import de.unihalle.informatik.MiToBo.operator.*;
4  import de.unihalle.informatik.MiToBo.datatypes.images.MTBImage;
5  import de.unihalle.informatik.MiToBo.exceptions.*;
6
7  public class MTB_Median_MTBPlugin extends MTBOperatorPlugInFilter {
8
9      @Parameter(label="resultImg", type=Type.OUTPUT,
10         description="Filtered_image")
11     private MTBImage resultImg;
12
13     /**
14      * Standard constructor.
15      */
16     public MTB_Median_MTBPlugin() throws MTBOperatorException {
17         // nothing to do here
18     }
19
20     /**
21      * Implementing standard ImageJ setup method.
22      */
23     @Override
24     public int setup(String arg0, ImagePlus imp) {
25         // create MTBImage from ImagePlus or retrieve MTBImage that is connected with ImagePlus
26         // and set as input image
27         this.mtbInput = MTBImage.createMTBImage(imp);
28
29         // allow 8 and 16-bit gray value images
30         return DOES_8G + DOES_16;
31     }
32
33     @Override
34     protected void operate() throws MTBOperatorException,
35         MTBProcessingDAGException {
36
37         // maybe call some other operators
38
39         // crate a new median operator object and set the input via its constructor
40         MTBMedian medianOp = new MTBMedian(this.mtbInput);
41         // invoke the operator
42         medianOp.runOp(false);
43         // set plugin output from median filter result
44         this.resultImg = medianOp.getResImage());
45     }
```

```
46      /**
47       * Implementing standard ImageJ run method.
48       */
49      @Override
50      public void run(ImageProcessor impIP) {
51          try {
52
53              // call the operate method to call additional operators or operations
54              this.runOp(false);
55              // display the filtered image via ImageJ
56              this.resultImg.show();
57          } catch (MTBOperatorException e) {
58              e.printStackTrace();
59          } catch (MTBProcessingDAGException e) {
60              e.printStackTrace();
61          }
62      }
63  }
```

Figure 8.2: Example how to implement a `MiToBo` `MTBOperatorPlugInFilter` using an `MiToBo` operator.

Chapter 9

Implementing commandline tools

The strict separation of code and interfaces in `MiToBo` allows operators to be easily accessed from different kinds of programs and user interfaces. Besides writing plugins which is actually the standard in ImageJ (cf. Chap. 8), the implementation of commandline tools is also straightforward in `MiToBo`.

Implementing a commandline tool can be done in either of two possible ways:

a) implementing a standard Java class with `main()` routine that directly uses operators

b) implementing the commandline tool itself as an operator where the `main()` routine just instantiates, configures and calls the actual commandline tool object

The first option a) is reasonable if the commandline tool does nothing more than calling certain operators. In particular, the commandline tool should not add any additional functionality related to image processing to the operators' functionalities. Contrary, if the commandline tool itself provides functionality in addition to the operators, it is advisable to implement the commandline tool itself as an operator following option b) to ensure proper self-documentation.

In practice commandline tools will most of the time just hand over commandline arguments to existing operators without adding new functionality by themselves, i.e. option a) is more common. Accordingly, below we will discuss the implementation of an example commandline tool for image thresholding which follows that strategy.

Part of the implementation is shown in Fig. 9.1. The class of the commandline tool basically contains the `main()` method which handles commandline arguments and operator calls directly. Commandline argument parsing is in this case done using the external library `jargs`[1] which is not part of the `MiToBo` distribution.

At the top of the `main()` method some commandline options are defined which directly relate to parameters of the thresholding operator invoked later. Subsequently the actual parsing

[1] http://jargs.sourceforge.net/

```
package cmdTools.segmentation;
import jargs.gnu.CmdLineParser;

/**
 * Commandline tool for thresholding plain images, stacks and ...
 */
public class ImageThresholdTool {

    public static void main(String [] args) {
        CmdLineParser parser = new CmdLineParser();
        CmdLineParser.Option oThreshold = parser.addDoubleOption('t',"threshold");
        CmdLineParser.Option oFGValue = parser.addDoubleOption('f',"fgvalue");
        CmdLineParser.Option oBGValue = parser.addDoubleOption('b',"bgvalue");

        // parse arguments
        try {
           parser.parse(args);
        }
        catch ( CmdLineParser.OptionException e ) {
           ...
        }
        // remaining arguments ok?
        String[] otherArgs = parser.getRemainingArgs();
        if (otherArgs.length != 2) {
            ...
        }
        // retrieve options
        Double threshold = (Double)parser.getOptionValue( oThreshold, null);
        Double fgValue = (Double)parser.getOptionValue(oFGValue, Double.POSITIVE_INFINITY);
        Double bgValue = (Double)parser.getOptionValue( oBGValue, Double.POSITIVE_INFINITY);

        try {
            ReaderImageMTB reader = new ReaderImageMTB(otherArgs[0]);
            reader.runOp(null);
            MTBImage inImg = reader.getResultMTBImage();

            ImgThresh thresOp = new ImgThresh();
            thresOp.setInputImage(inImg);
            thresOp.setThreshold(threshold);
            thresOp.setFGValue(fgValue);
            thresOp.setBGValue(bgValue);
            thresOp.runOp(false);

            MTBImage resultImg = thresOp.getResultImage();
            WriterImageMTB writer= new WriterImageMTB(otherArgs[1], resultImg);
            writer.runOp(false);
        } catch ( Exception e) {
            ...
        }
     }
}
```

Figure 9.1: Example code of a commandline tool using `MiToBo` operators.

is done and the values of the options are copied to local variables. The `try-catch` block in the lower part of the listing contains the calls of `MiToBo` operators. First the input image is loaded using the operator `ReaderImageMTB`. Subsequently the thresholding operator `ImgThresh` is initialized and invoked. While for the `ReaderImageMTB` class a convenience constructor defining all required inputs and parameters is available, in case of the `ImgThresh` class all configuration settings have to be done explicitly. After running the thresholding operator the last three lines in the `try-catch` block show how the result data is retreived from the operator (using its `getResultImage()` method) and then saved to disk by using the operator `WriterImageMTB`.

Note that the history graph associated with the output image will not contain any link to the commandline tool itself as it is not implemented as an operator. The history of resulting image that is written to disk will rather include only all explicit operator calls.

Chapter 10

MiToBo data types and their implementation

MiToBo defines a set of its own data types. Besides new image data types improving the ImageJ image classes, these include for example regions and contours and some other data type primitives frequently used with regard to image analysis applications. Most data types can be found in the package 'mitobo.datatypes' and its subpackages. To allow for easy identification of the data types the classnames of the data types in MiToBo always start with 'MTB', like in 'MTBRegion2D' or 'MTBImageDouble'.

There are three main reasons why MiToBo implements its own data types and not simply builds on top of the data types provided by ImageJ. The first reason is given by the fact that the handling of data objects in ImageJ is solved only in a rudimentary fashion, at least with regard to the API. As there are only some few explicit datatypes apart from images in ImageJ, data access or exchange is often cumbersome. Accordingly, MiToBo tries to enhance the usability and flexibility of image processing modules by defining its own data types, and by this tries to overcome some limitations nowadays present in ImageJ.

The second reason for introducing new data types is specific to MiToBo and its feature of self-documentation (cf. Chapt. 1.1). While MiToBo operators in principal support almost all available kinds of objects as inputs and outputs for operators, some few object types remain that cannot be handled natively within our concept. Among those data types are for example Java's wrapper classes like 'Integer' and 'Double', and – more generally speaking – all classes that implement the comparison of objects based on equality of object values. If objects of these kinds should be used as operator inputs or outputs this can only be done by wrapping them in data types providing object identification independent of the current value. Accordingly, for some basic and frequently used data types MiToBo implements such data object wrappers.

Another aspect regarding self-documentation is the fact that sometimes proper documentation of operator configurations requires more than just logging an input or output object's type

and current value. There might be other object parameters that are worth to be documented, e.g. like certain image-specific properties in case of images. To support the documentation of such object properties MiToBo defines a basic data type class supporting management and automated documentation of additional object properties.

Finally, in image processing workflows quite often the necessity arises to save intermediate and final analysis results in an appropriate way that in best case also offers the possibility to reinitiate the processing pipeline from that data again. MiToBo meets such requirements by enhancing its own data types with easy ways to read and write data objects from and to files based on XML representations and XML beans.

We will now discuss the different features and motivations of MiToBo data types in more detail. MiToBo's data type base class 'MTBData' and its features will be outlined in the following section. The subsequent section discusses XML input and output schemes for some data types and explains how the concepts can be implemented for new data types. The last section of this chapter will give a brief overview of MTBImage and its subclasses, probably the most basic but also most important data types of MiToBo.

10.1 MiToBo's data type definition and object properties

MiToBo allows to represent image processing pipelines as graph data structures, i.e. history graphs. In particular, for each data object being the result of an image analysis process composed of a series of data manipulations by MiToBo operators, the history graph allows to backtrace each single intermediate processing step subsuming all interactions with other objects and the parameter settings of the involved operators. While these data, together with the overall structure of the graph, already draw a detailed picture of the process pipeline, sometimes extended information about manipulated and generated data objects, i.e. input and output objects of the operators, are of interest that rise above the default data, like name, object class and package.

MiToBo defines the data type super class 'MTBData' to support adding specific information to input and output objects of operators. The class mainly offers the concept of data type *properties* to data types derived from this class, allowing programmers to further characterize objects in the processing history and also in general. Properties of operator inputs and outputs are embedded in the history graph representation. Each time a data object passes an output port, i.e. is taken out of an operator, the properties will be associated with the corresponding data port node in the graph. When later on viewing the graph with chipory, the properties can then be displayed as additional information of the corresponding ports.

A property is basically given as a pair of key and value and is supposed to specify object characteristics. For example in case of the MiToBo image data types properties subsume information like image and pixel sizes in all dimensions and the units of the axes. Examplary key value pairs are shown in Table 10.1.

Property	Value
location	"/home/user/images/microscope.tif"
StepsizeX	"1"
StepsizeZ	"0.5"
UnitX	"cm"
...	...

Table 10.1: Examplary properties and its values for an object of type `MTBImage`.

For setting and getting object properties '`MTBData`' defines two methods:

- `public void setProperty( String key, Object obj )`
  allows to set a property named 'key' to the string representation of 'obj'
- `public String getProperty( String key )`
  returns the string describing the value of the property named 'key'

Internally the properties are stored in a hashtable of the Java type '`Hashtable<String, String>`' to be found in the package `java.util`. Accordingly, keys and values are represented as strings. Nevertheless, for convenience an arbitrary object can be handed over to the set routine as shown above. It is automatically converted to a string via its `toString()` method that consequently should return an informative description of the object at hand.

The programmer of a new `MiToBo` data type is in general allowed to choose arbitrary names for the object properties without any restrictions, apart from one exception. There is one property predefined for all `MiToBo` data types which is the property denoted '`location`'. The location of a data object defines the place of origin where the data object is coming from. This can be the place where it is physically stored, i.e. the name of a file on disk or an URL, or it can point to a virtual location if the object was generated by an operator in the course of the processing pipeline. Note that although this property is by default attached to all data types extending 'MTBData', it is, however, only set automatically for `MiToBo` images. For other data types setting the location to proper values remains to the responsibility of the programmer of the specific data type. To set and read the location of an object methods are available:

- `public void setLocation( String location )`
  sets the object location to the given string
- `public String getLocation()`
  returns the current location of the object

Note that there are no automatic checks to ensure that property names are unique. Thus, if the `setProperty` method is called on a property which is already defined its previous value will be overwritten. This is particularly true for the property '`location`', so this key should never be used by the programmer within another context than intended to omit confusion.

10.2 Input and output using XML schemata

In most image analysis projects sooner or later the question appears how to save result data to disk for later use and reference. One common way for saving data to disk in a generic form is to use XML representations. `MiToBo` supports this kind of representation, i.e. provides methods for its data types to save objects to XML files and later on read them again from the corresponding files. Of course, there are plenty of ways to generate XML files starting from plain text-based output, going further to the direct use of XML document generators and parsers, and ending up with high-level interfaces and libraries like XMLBeans[1] for Java.

In `MiToBo` we suggest to use XMLBeans as they disengage the programmer from cumbersome tasks like invoking parsers on her/ his own and subsequently analyzing resulting XML documents. In short, using XMLBeans to load and save data objects requires to generate an XML schema definiton for the data type which specifies its internal structure, in particular its member variables. Once the schema is available corresponding XML wrapper classes can be generated using the `org.apache.xmlbeans.impl.tool.SchemaCompiler` included in XMLBeans, i.e. in the archive `xmlbean.jar`. The wrapper classes then can be used within the `MiToBo` data type classes to implement well-arranged I/O routines.

`MiToBo` already includes XML schemata definitions and wrapper classes for some basic data types. We will outline below how XML schemata can easily be defined for new data types, how the corresponding Java wrapper classes can be generated and how they can be used in operators and plugins.

10.2.1 Basics of `MiToBo` and XMLBeans

XML schemata and all other related definitions required to generate XML wrapper classes can be found in the `MiToBo` archive in the directory '$MITOBO/share/xmlschemata/mtbxml'. The directory contains all data required to read and save `MiToBo` data types. Inside this directory there are the following basic files:

- `MTBXML.xsdconfig`, which contains some basic namespace and other related defines
- `MTBXML.xsd`, which lists all `MiToBo` data types for which XML schemata are available (and which you will have to edit if you define new data types and schemata on your own)
- `MTBXMLBase.xsd`, which defines some very basic data types like 2D points and other primitives frequently used to compose more complex data types

All other files to be found there are schemata for `MiToBo` data types. They all have in common that their filenames begin with '`MTBXML`' followed by the name of the `MiToBo` data type

[1] http://xmlbeans.apache.org/

without the leading 'MTB'. E.g., for the data type 'MTBContour2DSet', which is a container for a set of contours, the corresponding schema would be termed 'MTBXMLContour2DSet.xsd'.

The basic procedure for generating new wrapper classes is the following one:

1. Generate an XML schema description in the directory '$MITOBO/share/xmlschemata/mtbxml'.
2. Generate the source code and class files using the SchemaCompiler from XMLBeans and build a jar archive to be linked to your data type code.

10.2.2 XML schema definitions

To define an XML schema for a data type it is necessary to specify the basic structure of the data type in XML. We will discuss this on the example of the MiToBo data type MTBContour2DSet, for which input and output routines based on XMLBeans are available. MTBContour2DSet is a simple container for objects of type MTBContour2D. It basically consists of a list of contours, but further defines a bounding box which may be interpreted as the region-of-interest in an image where the contours of the set are located. A 2D contour itself is given by a set of 2D points and a list of inner contours included in the given one (for more details have a look in the Javadocs of 'mitobo.datatypes.MTBContour2D' and 'mitobo.datatypes.MTBContour2DSet').

In Figure 10.1 we first of all show the overall structure of the XML schema description for the data type 'MTBContour2DSet' from the file 'MTBXMLContour2DSet.xsd' which we will discuss in detail now. At the top of the file there is the header with basic declarations. The most important one is the target namespace declaration. As long as you use the XML files in the context of MiToBo the target namespace should always be set to

```
targetNamespace=‘‘http://informatik.unihalle.de/MiToBo_xml’’
```

The target namespace amongst others specifies where the generated source files will be placed. While this is obligatory, the next section in the XML file is optional. By this it is possible to include XML schemata from other XML files, for example schemata for basic MiToBo XML data types from the file 'MTBXMLBase.xsd', as in this case. In this example we will use 2D points from that file later on.

Once header and imports are declared the actual schema declarations remain to be added. In this file there are three of them. At first we declare the schema of the XML data type for which the file is actually created and which is 'MTBXMLContour2DSetType' in this example. As a contour set consists of multiple contour objects we further add the type 'MTBXMLContour2DType', and as each of these contours consists of a list of 2D points we also add the type 'MTBXMLPointVectorType' to describe the point list. The latter two declarations could also be moved to separate files. However, as they are only used in the context of the contour set so far, they reside in this file for the sake of simplicity.

```
<!-- header -->
<?xml version="1.0" encoding="UTF-8"?> <xs:schema
   targetNamespace="http://informatik.unihalle.de/MiToBo_xml"
   xmlns="http://informatik.unihalle.de/MiToBo_xml"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   elementFormDefault="qualified"
   attributeFormDefault="unqualified">

<!-- import basic XML types -->
<xs:import namespace="http://informatik.unihalle.de/MiToBo_xml"
        schemaLocation="MTBXMLBase.xsd">
</xs:import>

<!-- definition of the XML type -->
<xs:complexType name="MTBXMLContour2DSetType">
  <xs:sequence>
   <xs:element name="xMin" type="xs:double"/>
   <xs:element name="yMin" type="xs:double"/>
   <xs:element name="xMax" type="xs:double"/>
   <xs:element name="yMax" type="xs:double"/>
   <xs:element name="contour" type="MTBXMLContour2DType"
               minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="MTBXMLContour2DType">
  <xs:sequence>
   <xs:element name="points" type="MTBXMLPointVectorType"
               minOccurs="1" maxOccurs="1"/>
   <xs:element name="inner" type="MTBXMLContour2DType"
               minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="MTBXMLPointVectorType">
  <xs:sequence>
   <xs:element name="point" type="MTBXMLPoint2DType"
               minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
</xs:schema>
```

Figure 10.1: Example XML schema for the data type `MTBContour2DSet`.

Structure of data type declarations. Each XML schema declaration consists of a tag with name `complexType` located in the namespace `xs` which is further detailed with the attribute `name` stating the actual name of the XML data type to be defined. Subsequently the actual type definition follows which in this file is in all three cases an element of tag type `sequence`, i.e. all data types are composed of a set of various objects. For example it is stated that '`MTBXMLContour2DSetType`' consists of a sequence of objects of type '`MTBXMLContour2DType`' from which we can have a minimum number of zero and a maximum number of 'unbounded':

```
<xs:element name="contour" type="MTBXMLContour2DType"
            minOccurs="0" maxOccurs="unbounded"/>
```

In addition the declaration contains four items of type `xs:double` that refer to the bounding box of the contour set mentioned above. From the second declaration for the type '`MTBXMLContour2DType`' we can see that an object of that type consists on the one hand of a set of points stored in a vector, i.e. which is of type `MTBXMLPointVectorType`, and on the other hand of another set of contours forming its inner contours. Note that XML data types can be defined in a recursive fashion like it is common for data types in other languages, too.

The final declaration within the file states that objects of type '`MTBXMLPointVectorType`' consist of a sequence of objects of type '`MTBXMLPoint2DType`'. The XML schema declaration for this type is not defined in the given file, but is imported from the file '`MTBXML.xsd`' as stated above.

Generating Java sources and wrapper classes. Once the schema declaration is available, source and class files have to be generated. To this end the schemata declarations need to be added to the list in the file `MTBXML.xsd` which might for example look as follows:

At top of the file we have again a header for the schema declaration with some declarations similar to the ones from the file in Fig. 10.1. Below some XML schemata files are imported, e.g. the file `MTBXMLContour2DSetType.xsd` that was discussed above. In the lower part of the file we find the final Java XML data type declarations for all XML schemata. The names declared here yield the names for some of the Java classes to be generated from the schemata. The actual generation is invoked by calling the XMLBeans SchemaCompiler as follows:

```
java -Xmx256m org.apache.xmlbeans.impl.tool.SchemaCompiler \
     -out $MITOBO/intjars/mtbxml.jar \
     -src $MITOBO/src/ MTBXML.xsd MTBXML.xsdconfig
```

When running the SchemaCompiler make sure that `xbean.jar` is in your Java classpath. The option `-out` specifies the target output jar archive which per default should located in `$MITOBO/intjars/`. The option `-src` denotes the root path to where the source files are copied.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
   targetNamespace="http://informatik.unihalle.de/MiToBo_xml"
   xmlns="http://informatik.unihalle.de/MiToBo_xml"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   elementFormDefault="qualified"
   attributeFormDefault="unqualified">

<xs:import namespace="http://informatik.unihalle.de/MiToBo_xml"
    schemaLocation="MTBXMLPolygon2DSet.xsd">
</xs:import>

<xs:import namespace="http://informatik.unihalle.de/MiToBo_xml"
    schemaLocation="MTBXMLContour2DSet.xsd">
</xs:import>

<xs:element name="MTBXMLPolygon2DSet" type="MTBXMLPolygon2DSetType">
</xs:element>

<xs:element name="MTBXMLContour2DSet" type="MTBXMLContour2DSetType">
</xs:element>
</xs:schema>
```

Figure 10.2: Examplary contents of the file `MTBXML.xsd`.

Given this path and the target namespace as defined in the XML schemata files the sources will be placed in `$MITOBO/src/de/unihalle/informatik/MiToBo_xml`. Simultaneously, corresponding class files will be generated and directly put into the given jar archive. Note that the source code files are mainly helpful for API documentation and do not need to be edited or modified in any way. The resulting jar archive is the one that you need to place in your classpath and which contains the Java wrapper classes to be used for reading and writing the corresponding data types.

In case of the example discussed above, for the schema declaration from file `MTBXMLContour2DSet.xsd` the following Java classes will be generated:

- MTBXMLContour2DSetDocument.java
- MTBXMLContour2DSetType.java
- MTBXMLContour2DType.java
- MTBXMLPointVectorType.java

For each type declaration from the file a corresponding type class is generated. In addition, the class in file `MTBXMLContour2DSetDocument.java` links the new data type definitions to Java XML, i.e. implements parsers and other things related to the handling of XML documents.

Using XML wrapper classes for data type I/O. The wrapper classes generated above can be used to implement I/O routines for the corresponding `MiToBo` data type. In case of the `MTBContour2DSet` type the write method based on the XML wrapper classes for example is displayed in Figure 10.3

```
1  public void write(String filename) throws MTBException {
2          try {
3            BufferedWriter file =
4                    new BufferedWriter(new FileWriter(filename + ".xml"));
5            // generate XML documents
6            MTBXMLContour2DSetDocument xmlContourSetDocument =
7                    MTBXMLContour2DSetDocument.Factory.newInstance();
8            MTBXMLContour2DSetType xmlContourSet =
9                    xmlContourSetDocument.addNewMTBXMLContour2DSet();
10
11           // transfer the contours to XML
12           MTBXMLContour2DType [] cList=
13                   new MTBXMLContour2DType[this.contourSet.size()];
14           for (int p = 0; p < this.contourSet.size(); p++) {
15             MTBContour2D contour = this.contourSet.elementAt(p);
16             cList[p]= this.getContour2DAsXml(contour, null);
17           }
18           xmlContourSet.setContoursArray(cList);
19           xmlContourSet.setXMin(xMin);
20           xmlContourSet.setYMin(yMin);
21           xmlContourSet.setXMax(xMax);
22           xmlContourSet.setYMax(yMax);
23
24           // write the xml file
25            file.write(xmlContourSetDocument.toString());
26            file.close();
27         } catch (Exception e) {
28           System.err.println("Exception" + e);
29         }
30         // write processing history
31         writeHistory(filename);
32       }
```

Figure 10.3: XML wrapper class based write method for `MTBContour2DSet`

At first an XML document corresponding to the data type '`MTBContour2DSet`' needs to be instantiated together with an object of the related XML data type '`MTBXMLContour2DSetType`'. Subsequently the different ingredients of a contour set are transferred to XML which are in detail all contours and the coordinates of the bounding box. Each single contour is explic-

itly translated to XML which in this case is done by the helper function `getContour2DAsXml` to be found in the same class. Its basic definition is shown in Figure 10.4. Initially an ob-

```
1  public MTBXMLContour2DType getContour2DAsXml(MTBContour2D contour,
2                              MTBXMLContour2DType xmlC) {
3
4              MTBXMLContour2DType xmlContour;
5
6              // instantiate  contour
7               ...
8              // transfer  list  of contour points
9              MTBXMLPointVectorType xmlPlist=
10                 MTBXMLPointVectorType.Factory.newInstance();
11             Vector<java.awt.geom.Point2D.Double> points = contour.getPoints();
12             for (int i = 0; i < points.size (); i++) {
13                 MTBXMLPoint2DDoubleType xmlPoint = xmlPlist.addNewPoint();
14                 xmlPoint.setX((int)points.elementAt(i).getX());
15                 xmlPoint.setY((int)points.elementAt(i).getY());
16             }
17             xmlContour.setPoints(xmlPlist);
18
19             // inner contours
20             Vector<MTBContour2D> innerContours= contour.getAllInner();
21             MTBXMLContour2DType [] inList=
22                 new MTBXMLContour2DType[innerContours.size()];
23             for (int i = 0; i< innerContours.size(); ++i) {
24                  inList [i]= this.getContour2DAsXml(innerContours.get(i),null);
25             }
26             xmlContour.setInnerArray(inList);
27             return xmlContour;
28         }
```

Figure 10.4: Example method for translating `MTBContour2D` types into XML

ject of type '`MTBXMLPointVectorType`' is instantiated to which afterwards points of type '`MTBXMLPoint2DDoubleType`' are added. The coordinates of the single points are set according to the coordinates of the contour points. In a second phase all inner contours are added to the new XML object by calling the helper function recursively on all inner contours.

The read methods for the data type work similar and just the other way round. For details on these methods and related helper methods please take a look into the javadoc API documentation or directly in the source file `src/de/unihalle/MiToBo/datatypes/MTBContour2DSet.java`.

Concluding remarks. All XML schemata are part of your MiToBo distribution, however, related wrapper classes are not automatically compiled when calling standard MiToBo ant tasks. Rather precompiled class files are supplied via the jar named mtbxml.jar to be found in $MITOBO/intjars/. Accordingly, if you would like to permanently add new XML wrapper classes for a certain data type to MiToBo you need to generate a new jar archive manually and afterwards make sure that all users of your code replace the default version of the jar with your release. Note that for the wrapper sources are not included in the MiToBo sources, however. The API documentation for these classes is part of the MiToBo Javadocs.

10.3 MTBImage

MiToBo defines its own image classes, namely MTBImage and its subclasses, for the following reasons:

- Extended pixel value precision to support all primitive numeric datatypes of Java
- Easy access to image pixel data, but also to properties like physical pixel size etc.
- Functionality for MiToBo's operator concept, namely documentation of image properties

The image classes can be found in de.unihalle.informatik.MiToBo.datatypes.images. This subsection about MTBImage is roughly divided into the following parts. At first, some important details about the structure of MTBImage are given and which image types are available in MiToBo. An overview of most common methods for creation and manipulation of MTBImages follows. This subsection is closed by the description of MTBImage file IO and how it integrates in MiToBo's operator concept.

10.3.1 The ideas behind MTBImage

MTBImage was not developed to fully replace ImageJ's ImagePlus, but rather to wrap the ImagePlus objects if possible. The most convenient way to create a MTBImage from an existing ImagePlus object is to simply specify the ImagePlus as input to the method public static MTBImage.createMTBImage(ImagePlus img). The created MTBImage holds a reference to that ImagePlus object and stores the image size as well as physical pixel size and units if available. For fast pixel access, the MTBImage keeps direct references to the data array or arrays in case of a (hyper-)stack.

When a MTBImage is created from an ImagePlus, that MTBImage must uniquely be associated with the specified ImagePlus, as no new MTBImage is created, but the existing one is used. This case occurs very often, e.g. when an image window is selected from the ImageJ GUI and used as input to a MiToBo plugin. Therefore another reference is kept in the *properties* hash-table

of the ImagePlus to the MTBImage, which was initially created using the ImagePlus. When an ImagePlus with a reference to an existing MTBImage is passed to createMTBImage(ImagePlus img), the existing MTBImage is simply returned.

Another aspect of MTBImage is to think of an ImagePlus as a 5-dimensional image, which is the highest possible dimensionality of an image in ImageJ (hyperstack). To provide easy access to higher dimensional image data, methods exist to access data in 5D hyperstacks, 3D stacks and 2D images, which will be discussed in more detail in section 10.3.3.
MTBImage objects are designed in a similar way as ImageJ's ImageProcessor. You usually reference them by the abstract type MTBImage, while one of its subclasses is actually instantiated.

10.3.2 Subclasses of MTBImage: Image types

One reason to develop a new image type was the limitation of ImageJ images to 32-Bit pixel value precision. The need for a 64-Bit precision floating-point image type to store most accurate results was obvious. Also the lack of a (true) 32-Bit integer type in ImageJ can bear some problems, e.g. when consecutive labels are given to image regions especially in higher dimensional data. The instantiable subclasses are comprised of the name MTBImage and the Java data type of the pixel values. The following list shows the available image types in MiToBo:

- MTBImageByte for byte-type pixel values (unsigned as in ImageJ)
- MTBImageShort for short-type pixel values (unsigned as in ImageJ)
- MTBImageInt for int-type pixel values
- MTBImageFloat for float-type pixel values
- MTBImageDouble for double-type pixel values
- MTBImageRGB for three byte-type pixel values, one for each color channel red, green and blue (unsigned)

These image types share the same interface, but are differentiated by MiToBo image types that have a corresponding ImageJ type and thus simply wrap the ImagePlus, and those types that do not have a corresponding ImageJ type. If only MTBImage s are used, there is no difference in the interface. If ImagePlus has to be consulted (e.g. using ImageJ functions or displaying images), please keep in mind the difference described in the following two paragraphs.

MTBImages with corresponding ImageJ types. If values are changed of an MTBImage that simply wraps the corresponding ImagePlus, the changes are applied to that ImagePlus directly, because MTBImage and ImagePlus share the same data arrays. Table 10.2 lists the subtypes of MTBImage and their corresponding ImageJ image types.

MTBImage subtype	ImageProcessor of corresponding ImagePlus
MTBImageByte	ByteProcessor
MTBImageShort	ShortProcessor
MTBImageFloat	FloatProcessor

Table 10.2: MTBImage types with corresponding ImageJ types.

MTBImages without corresponding ImageJ types. MTBImages which cannot be represented by corresponding ImageJ types keep their own data arrays and are not linked to an ImagePlus object at creation. Images of such data types cannot be instantiated by the createMTBImage(ImagePlus img) method. These images are usually constructed from scratch by specifying datatype and image size, or by conversion from another MTBImage to that datatype. Nevertheless an ImagePlus object is often needed, usually for visualization. MTBImage provides the function getImagePlus() to obtain an ImagePlus. The ImagePlus created is firmly associated to the MTBImage. In the case of the data types discussed in this paragraph, a new ImagePlus of the ImageJ type that is supposed to provide the least loss of information is created. By the way, as MTBImage provides its own show() and updateAndRepaint() methods which use the getImagePlus method, you won't have to explicitly get the ImagePlus object for pure displaying purpose.
Always keep in mind, that a second image data object is kept in memory, once getImagePlus() or the displaying methods are called!
Table 10.3 describes the MTBImage types that do not have a corresponding ImageJ type and explains, how they are mapped to ImagePlus.

MTBImage subtype	ImageProcessor of created ImagePlus	Pixel value conversion
MTBImageInt	FloatProcessor	cast from int to float
MTBImageDouble	FloatProcessor	cast from double to float
MTBImageRGB	ColorProcessor	lossless encoding of three byte values to ImageJ's int color representation

Table 10.3: MTBImage types without corresponding ImageJ types.

10.3.3 Construction, data access and other useful functions of MTBImage

This subsection gives a short overview of the functions of MTBImage that are widely used when working with MiToBo. A full description can be found in the Javadoc API of MiToBo.

At first, methods to create new MTBImages are presented. As there are no visible constructors, you have to use the following static factory functions:

- `public static MTBImage createMTBImage(ImagePlus img)`
  creates a new `MTBImage` of the correct subtype, which is uniquely linked to the `ImagePlus`
- `public static MTBImage createMTBImage(int sizeX, int sizeY, int sizeZ, int sizeT, int sizeC, MTBImageType type)`
  creates a new `MTBImage` from scratch given the size and the datatype of the new image

The following methods can be used to create `MTBImage`s from existing ones:

- `MTBImage duplicate()`
  duplicates a `MTBImage`.
- `MTBImage convertType(MTBImageType type, boolean scaleDown)`
  creates a `MTBImage` of different datatype from the values of the source image

There are more methods (e.g. to create `MTBImage`s only from part of an image) and all these methods also exist with one more argument to specify, an `MTBOperator` object. For the sake of briefness, only the versions without that argument and only the most commonly used methods are presented here. Please refer to the API for the other methods.

Methods for image pixel data access are declared by the `MTBImageManipulator` interface, which is implemented by `MTBImage`. The behavior of data access methods is similar to ImageJ's `getPixel` and `putPixel` methods, which return or take an `int` to cover 8-Bit to 32-Bit values. `MTBImage` provides the same methods called `getValueInt` and `setValueInt`, with the only difference, that `int`s are casted and not reinterpreted in case of underlying floating point datatypes. Keep in mind, that like ImageJ methods `byte` types return and take values in the range [0,255] and `short` types in the range [0,65535]. To cover floating point types additional methods exist, which return or take `double` values. These methods are called `getValueDouble` and `putValueDouble`, which are the safest way to go with, if you cannot be sure which kind of images have to be processed.
A word to (hyper-)stacks: ImageJ holds an array of 2D images, no matter if the image is three-, four- or five-dimensional. 2D images (called *slices* in the following) in this array (called *stack*) are referenced by 1 to N, where N is the number of slices. `MTBImage` uses indexing that is known to every programmer, starting from 0 to $(N-1)$.

- `int getValueInt(int x, int y, int z, int t, int c)`
  returns the pixel value at (x,y,z,t,c) as `int`
- `void putValueInt(int x, int y, int z, int t, int c, int value)`
  sets the pixel value at (x,y,z,t,c) using an `int` as input value
- `double getValueDouble(int x, int y, int z, int t, int c)`
  returns the pixel value at (x,y,z,t,c) as `double`

- `void putValueDouble(int x, int y, int z, int t, int c, double value)`
  sets the pixel value at (x,y,z,t,c) using a `double` as input value

`MTBImageRGB` can be modified in the same way as color images in ImageJ, by encoding the color values to an `int` and then pass that `int` to the `putValueInt/Double` method. But `MTBImageRGB` further provides methods to get and set values of the different color channels separately, or even get and work on the `MTBImageByte`s that represent the separate color channels.

For work with 2D images or 3D stacks, there are equivalent methods that take only 2D (x,y) or 3D (x,y,z) coordinates. You can also use these methods to access certain slices (2D images) or z-stacks (3D images) of a (hyper-)stack. Therefore you can set internal variables of `MTBImage` to specify a "current" slice or z-stack with the following functions:

- `void setActualSliceCoords(int z, int t, int c)`
  sets the coordinates of the "current" slice
- `void setActualSliceIndex(int idx)`
  sets the index of the "current", meaning the index in the array of slices
- `void setActualZStackCoords(int t, int c)`
  sets the coordinates of the "current" z-stack (leaves "current" slice index unchanged)

The image data should be accessed by the above methods to develop algorithms for generic image types. The data access methods are kept as fast as possible (e.g. no further function calls), but be aware that for this reason the specified coordinates are not checked. This means that running out of the data arrays' bounds will cause an `ArrayOutOfBoundsException` that is not forced to be caught.

For fast processing of higher dimensional images, you should also be aware of how to iterate through the pixels. The usual ordering in `ImagePlus` hyperstacks is XYCZT, while `MiToBo`'s interface order is XYZTC. You should therefore iterate over the pixels of a `MTBImage` as shown in the example below:

```
MTBImage img = MTBImage.createMTBImage(100, 100, 100, 100, 100,
                                       MTBImageType.MTB_BYTE);

for (int t = 0; t < img.getSizeT(); t++)
  for (int z = 0; z < img.getSizeZ(); z++)
    for (int c = 0; c < img.getSizeC(); c++)
      for (int y = 0; y < img.getSizeY(); y++)
        for (int x = 0; x < img.getSizeX(); x++)
          img.putValueInt(x,y,z,t,c,255);
```

If slicewise processing is possible, you can simply iterate over all slices, which produces less lines of code and is the fastest way to access all pixels:

```
MTBImage img = MTBImage.createMTBImage(100, 100, 100, 100, 100,
                                       MTBImageType.MTB_BYTE);

for (int i = 0; i < img.getSizeStack(); i++) {
   img.setActualSliceIndex(i);

   for (int y = 0; y < img.getSizeY(); y++)
      for (int x = 0; x < img.getSizeX(); x++)
         img.putValueInt(x,y,255);
}
```

10.3.4 `MTBImage` IO and the `MiToBo` operator concept

`MTBImage` extends the `MTBData` class and therefore fully integrates in `MiToBo`'s operator concept. A difference to other `MTBData` types is the file input and output. `MTBImage` objects can be written to and read from disk using the `ImageWriterMTB` and `ImageReaderMTB` operators, which can be found in the package `de.unihalle.informatik.MiToBo.io.files`.
The output is comprised of two separate files: one image file in any supported format, and a file (*.ald) that contains the image's processing history (see chapter 3). This history file – if present – is automatically loaded when the image is opened using `MiToBo`'s `ImageReaderMTB` operator. The processing history files can be examined using `chipory` (Appendix A).
`MiToBo` uses the Bio-Formats Library (http://www.loci.wisc.edu/software/bio-formats) and thus allows reading and writing of files supported by Bio-Formats. Bio-Formats is a sophisticated image IO library targeted at the various file formats in bio-medical imaging. One great feature is the support of formats that satisfy the Open Microscopy Environment (OME) standard (http://www.openmicroscopy.org).

Chapter 11

Tools and helper classes

MiToBo provides certain classes not directly related to image processing, however, useful for doing things like time measurements or plugin configuration. Such tools can usually be found in the package `mitobo.tools.system`.

11.1 Plugin Configuration

For user specific configuration of plugins MiToBo supports environment variables and JVM properties as well as ImageJ preferences (Chap. 5). For accessing environment variables and properties/preferences MiToBo provides the class `mitobo.tools.system.EnvironmentConfig` which supports easy access to variables. It basically defines the following methods:

- `public static void`
  `setImageJPref(String plugin, String envVar, String val)`

  This method allows to set a preference in the ImageJ configuration file. It is saved to the user specific ImageJ configuration file (usually ~/.imagej/Prefs.txt). Note that the saving requires the ImageJ gui to be used as otherwise related methods for preference saving are not called.

- `public static String`
  `getConfigValue(String plugin, String envVariable)`

  With this method environment configurations can be accessed.

The second method follows the formerly defined priority ordering of the different configuration options, i.e. first looks for an environment variable with the given name, then checks for JVM properties and third for ImageJ preferences. If not all options are to be checked in this order, the following methods can be used alternatively:

- `public static String`
  `getEnvVarValue(String plugin, String envVariable)`

  This method allows to directly read environment variables.

- `public static String`
  `getJVMPropValue(String plugin, String envVariable)`

  This method allows to directly read JVM properties.

- `public static String`
  `getImageJPropValue(String plugin, String envVariable)`

  This method allows to directly read ImageJ preferences.

Note that in all cases the prefix 'mitobo.' for properties and preferences or 'MITOBO_' for environment variables, respectively, is internally added to the variable and property names. The programmer usually does not need to pay attention on this feature, it should solely be kept in mind by a user when setting values for properties and environment variables.

Appendix A

Graph-Visualization: `chipory`

Processing histories are stored in XML format using *GraphML* with some `Alida` specific extensions as mentioned in Chapter 7. To display histories we extended *Chisio*[1] 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[2]. A single zip-file is provided for running `chipory` on Linux systems with 32- or 64-bit as well as on Windows system. 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.

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 history 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

[1] http://sourceforge.net/projects/chisio
[2] http://www.informatik.uni-halle.de/alida

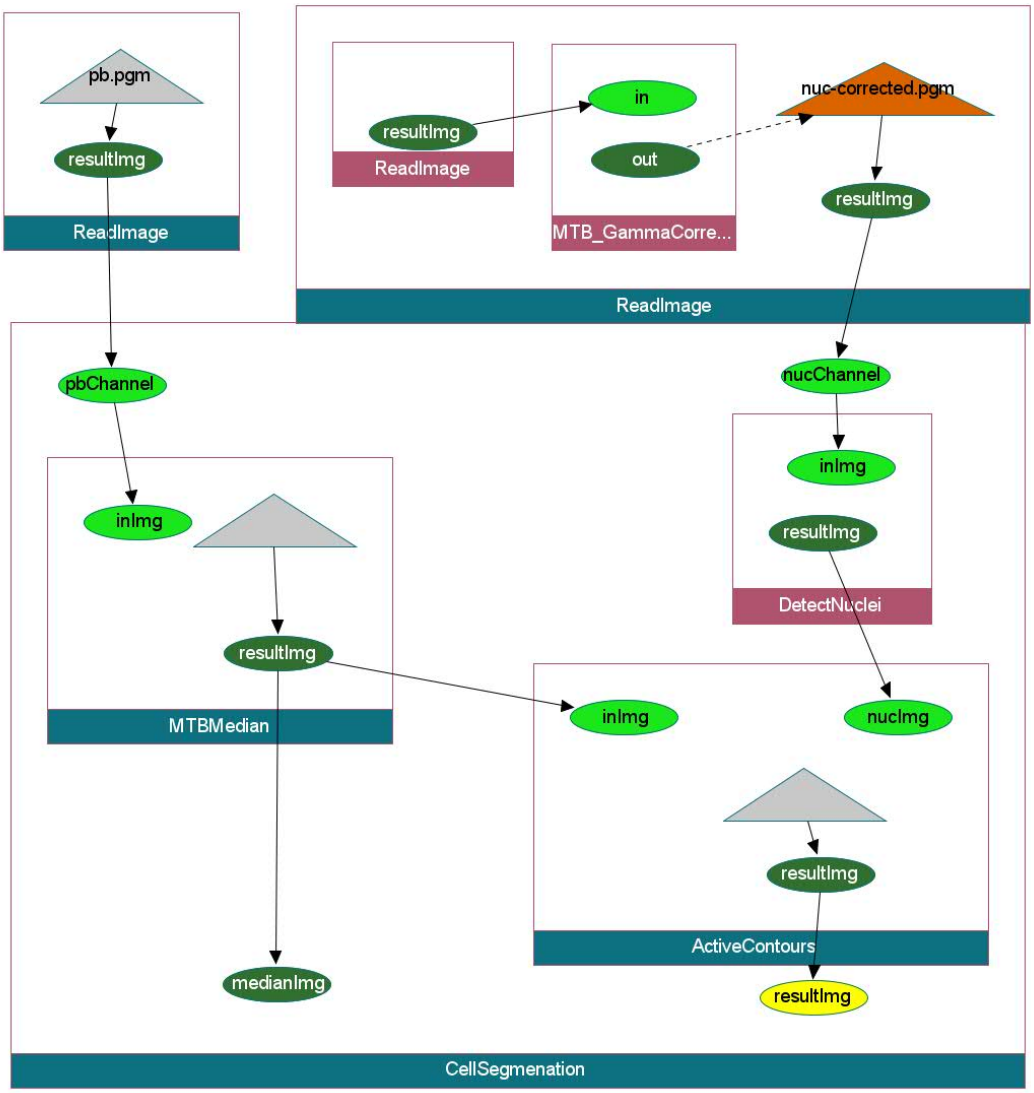


Figure A.1: Example processing graph.

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 A.1 an example processing graph extracted from an image analysis procedure implemented in `MiToBo` 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 history 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 a operator node is uncollapsed it is shown in blue, if it is collapsed it is of dark red. This is shown

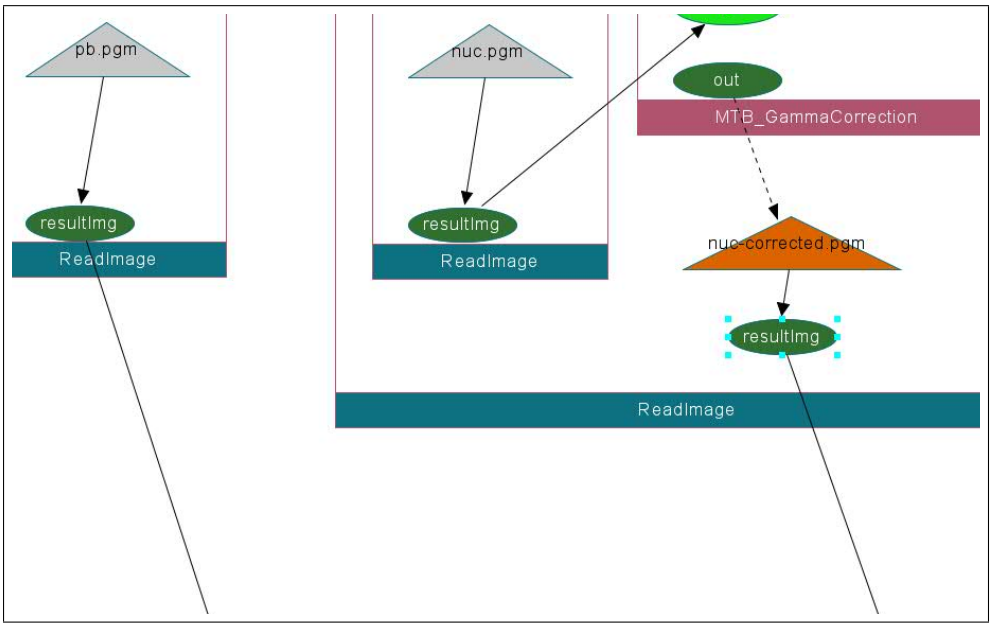


Figure A.2: Screen shot of `chipory` for a part of the same processing history as shown in Fig: A.1, however the collapsed instance of the `ReadImage` has been uncollapsed.

in Fig. 3.1 where the operator `DetectNuclei` has been collapsed. A selected operator node may be collapsed or uncollapsed by a left double mouse click. Collapsing makes all enclosed operator and data nodes invisible, thus, only the ports of a collapsed operator are shown. If the node is uncollapsed later on 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 right instance of the operator `ReadImage` as shown in Fig. A.2 we find that the source of the port `in` for the operator `MTB_GammaCorrection` was read from the file `nuc.pgm` which has no prior history. The later fact is visually marked by the grey shading of the corresponding triangular data port. On the other hand, `nuc-corrected.pgm` which is passed to `CellSegmentation` via the `nucChannel` port has a processing history associated which was read from the `.ald`-file accompanying the image data in `nuc-corrected.pgm`. This is indicated by the orange color of the data port.

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

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 mouse click.

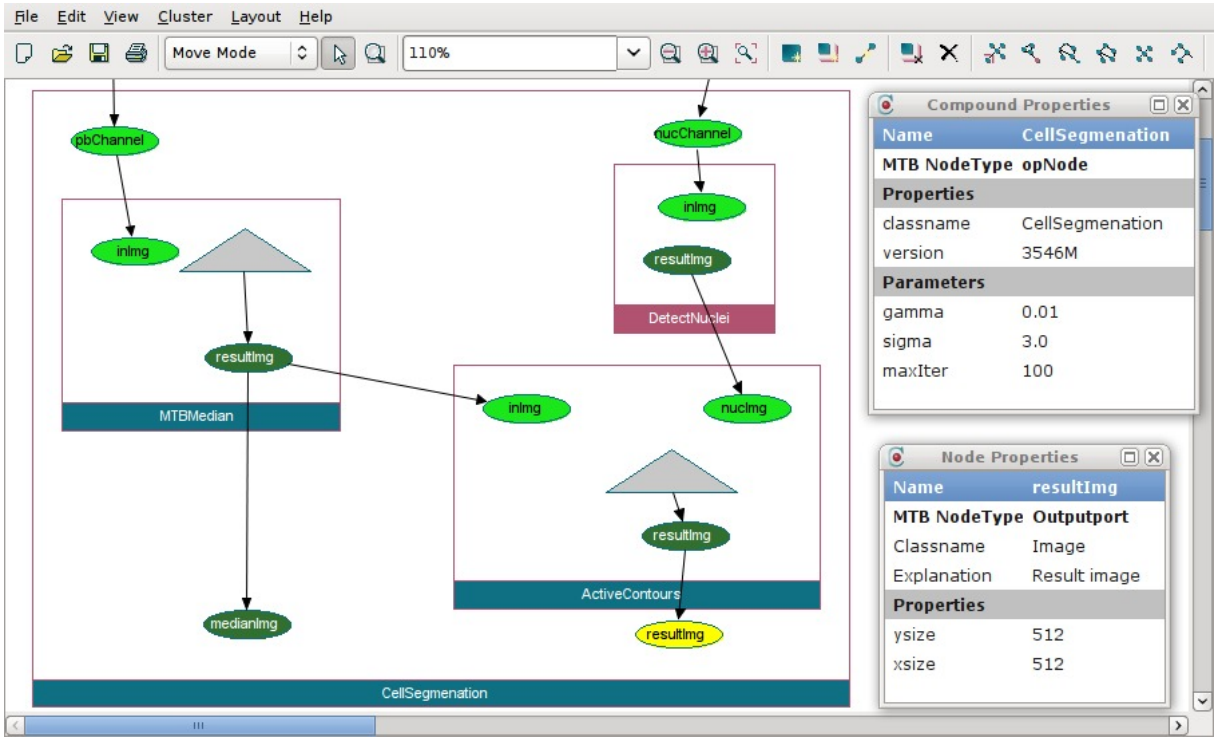


Figure A.3: Screen shot of `chipory` with details for the operator `CellSegmentation` and the output port `resultImage` of the same operator.

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 port the java class of the object as it passed into our along with the explanatory text of this port
- for output ports the properties of the object valid when pass out of the operator if it is of type `ALDData`.

In Fig. A.3 this is shown for the operator `CellSegmentation` and the output port `resultImage` of the same operator.

Appendix B

MiToBo-Resources: Files and Directory Contents

Once the MiToBo source archive was extracted to a directory of your choise which we denote with '$MITOBO' there you will find the following directories and files:

- $MITOBO/.
  contains license and ant build files
- $MITOBO/etc
  contains configuration files
- $MITOBO/share/README.mitobo-ant
  explains how to use ant with MiToBo
- $MITOBO/share/xmlschemata/mtbxml
  contains all files related to MiToBo's XML extensions
- $MITOBO/share/logo
  contains some nice pictures
- $MITOBO/share/scripts
  contains shell scripts for running MiToBo
- $MITOBO/src/de/unihalle/informatik/Alida/
  contains the source files and directories of Alida
- $MITOBO/src/de/unihalle/informatik/MiToBo/
  contains the source files and directories of MiToBo
- $MITOBO/src/plugins
  contains the sources of all MiToBo plugins

- $MITOBO/src/cmdTools
  contains sources of `MiToBo` commandline tools