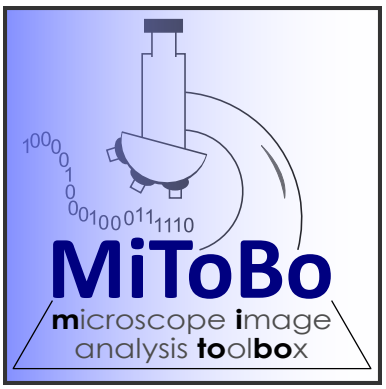
Martin Luther University Halle-Wittenberg
Institute of Computer Science
Pattern Recognition and Bioinformatics

User and Programmer Manual



MiToBo - Microscope Image Analysis Toolbox

Version 0.9

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 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 ***M****icroscope* ***I****mage Analysis* ***To****ol****Bo****x* developed at the Martin Luther University Halle-Wittenberg, tries to enhance ImageJ with regard to these aspects. `MiToBo` completely separates the implementation of image processing techniques and algorithms from any interface using these implementations, as for example ImageJ plugins or commandline tools. In `MiToBo` each image processing 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 image processing techniques the basic concept in `MiToBo` are 'operators'. Each operator is associated with input data objects, output data objects and certain configuration parameter objects that allow to specify how the operator works on the input data to produce the output data. Given this principal concept any image analysis pipeline may be interpreted as a directed graph. The operators constitute the nodes of this graph and the data flow between operators is represetend by edges connecting the different operator nodes.

`MiToBo` builds on ImageJ image datatypes and does not interfere with ImageJ's plugin interfaces, i.e. allows to program ImageJ-compatible plugins based on the `MiToBo` operator concept. However, in addition to the ImageJ interfaces which focus on plugin and script development, `MiToBo` also 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

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

data exchange and operator handling, and establishes a powerful fundament for adding new sophisticated features to ImageJ.

One of these features already available today in MiToBo is the automatic documentation of image analysis pipelines with regard to the operations that are carried out on the input data and related parameter configurations. MiToBo allows to document this pipeline in terms of 'history graphs'. With each output data object such a graph is saved in XML format which contains a detailed documentation of all parameter settings and data modifications and is especially helpful for longterm parameter documentation.

Another quite interesting feature which soon might gain significant interest is the graphical programming of image processing applications based on ImageJ and MiToBo. The MiToBo 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.

The MiToBo operator concept 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 operator concept might be helpful for developers of image processing applications and by this further extend ImageJ's selection of valueable features.

This manual is organized in three parts. The first part gives a short introduction to MiToBo combined with some notes on installation and dependencies. The second part is dedicated to users who are mainly interested in using MiToBo plugins or commandline tools for their own work to benefit from the automatic documentation capabilites of MiToBo. The third 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, more information about MiToBo data types and also about programming with MiToBo in general. 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!

Part I

Introduction and First Steps

Chapter 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,

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

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. In Sec. 2.1 the installation instructions for simply using MiToBo plugins and commandline tools are given, in Sec. 2.2 the necessary steps for using the MiToBo's API and its operators in your own code can be found.

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[1]. 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 the plugins directory for ImageJ by passing the option '-Dplugins.dir=<DIR>' to the Java virtual machine when starting ImageJ
3. include all other jars from the archive into your CLASSPATH variable

[1] www.informatik.uni-halle.de/mitobo

4. download external jars required by MiToBo to the same folder and include them also 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' menu (starting with prefix mtb_)

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

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 2.1, i.e. you mainly need to 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). Then the sources can be compiled in one of the following two ways:

A) using 'ant'

B) using 'eclipse'

Compiling MiToBo using 'ant'. To compile MiToBo with the Java tool 'ant', first of all enter the directory '$MITOBO/etc'. There you can find the file 'ant_config_templ.xml' which you need to copy to a file called 'ant_config.xml'. Open the new file with your favorite text editor and change all variable definitions according to your local system. Afterwards 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.

Chapter 3

Main Features: Operators, Plugins and History Graphs

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. This success is mainly stems from user-friendly and intuitive graphical user interfaces and a large variety of algorithms and plugins included in ImageJ that cover many important areas of image processing and analysis.

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 certain 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 and data structures. MiToBo defines an image analysis pipeline as a sequence of operations subsequently 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 implemented in 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 items from given input data. Accordingly, all image processing and analysis algorithms in `MiToBo` are implemented in terms of operators, i.e. each Java class implementing a certain technique or functionality is an operator.

Technically the concept is realized by defining a common superclass for all image analysis modules denoted '`MTBOperator`' from which all implemented 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. 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. This concept 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.

History Graphs. The self-documentation of image analysis processes in `MiToBo` is directly embedded into the operator concept. Operators as well as input and output data objects provide internal functionality to store process data during the course of an analysis pipeline which later on can be stored in terms of a *history graph* 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, also segmentation results like regions or contours and histograms or other numerical data are common. For `MiToBo` images the history graphs are automatically saved to disk together with the images themselves by `MiToBo`'s image open and save routines. Specifically, in addition to each image file a second file with extension '`.mph`' is written containing the history graph. See Figure 3.1 for an examplary visualization of such a graph.

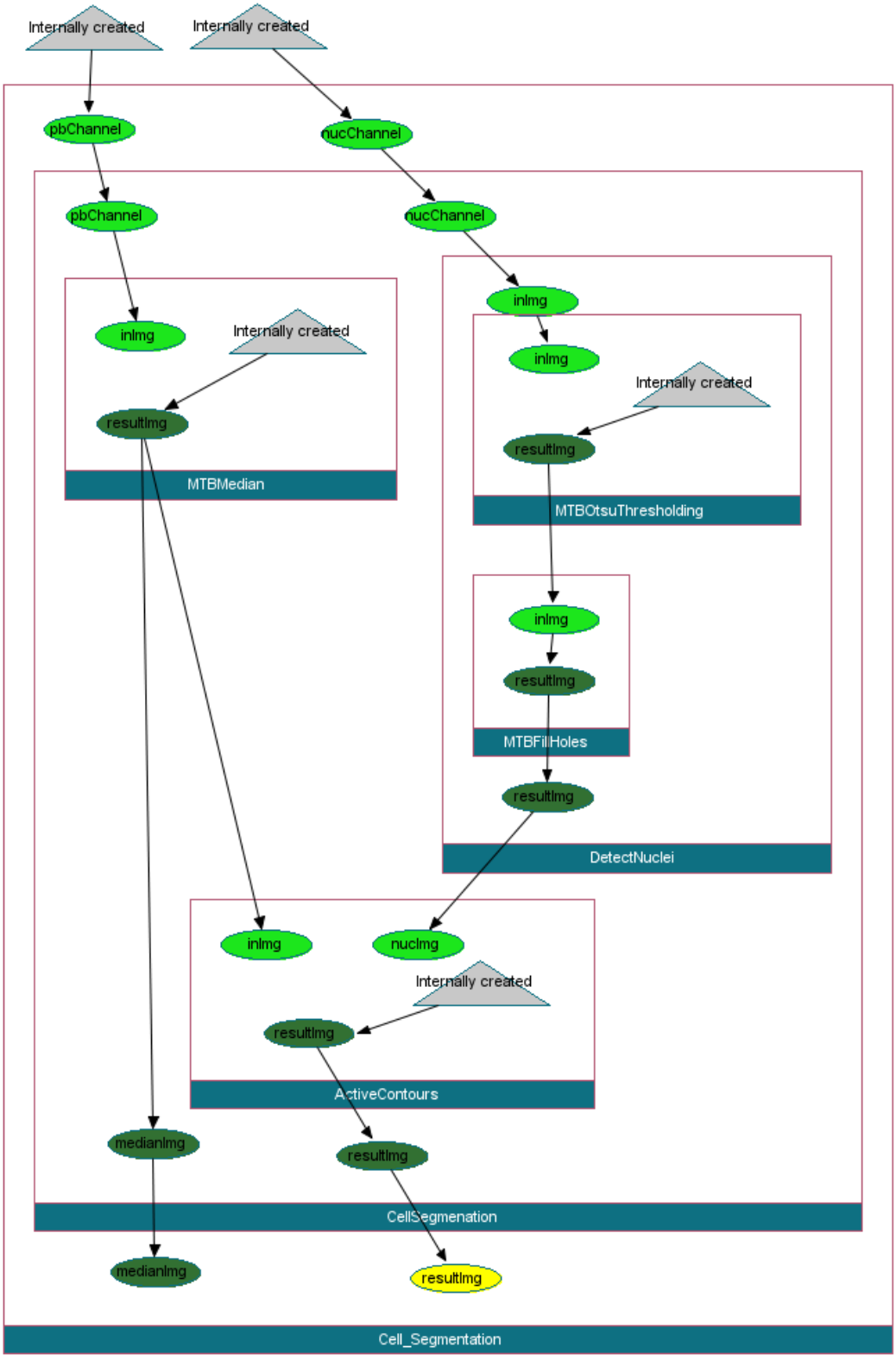


Figure 3.1: A `MiToBo` history graph: the directed acyclic graph represents the application of nested operators. Operators are depicted as rectangles, input and output ports as ellipses filled in light or dark green, respectively. The grey triangles relate to newly generated data objects, and the yellow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

The processing history comprises the originating data and all operations applied to the data as well as the order of application. Furthermore, several data properties like the location of the original file on disk or in a database and the type of the data are recorded as well as data type specific information (cf. Chap. 7). For operators their parameters and the version of the software are stored. This history extends across the application of plugins and commandline tools as the information is directly linked to data items independent of which user, e.g. plugin or commandline tool, invokes the operator.

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 in terms of an integrated documentation of all analysis pipeline and compatibility to ImageJ as underlying integrated software framework.

Part II

MiToBo: The user's view

Chapter 4

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, page 8 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 which shows a screenshot of `mtbchisio` (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 maybe connect 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, that correspond 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` and `Save_Image` 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.mph`'. The extension '`.mph`' indicates a `MiToBo` *processing history* file. When later on reading the image using `Open_Image`, `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. I.e., 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 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 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 clashes. 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
Open_Image/ Save_Image	MITOBO_IMAGEDIR	Directory where images are expected. For example checked if the following two variables are not set.
Open_Image	MITOBO_OPENDIR	Directory where browsing starts the first time.
Save_Image	MITOBO_SAVEDIR	Directory where browsing starts the first time.

Table 5.1: List of Important variables and properties

Part III

The programmer's view

Chapter 6

MiToBo operators

The heart of MiToBo's concept are operators that implement all image analysis capabilites. In this chapter we discuss the operator interface, how existing MiToBo 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. All data passed into or returned from an operator is of type MTBData. Examples are images (e.g., MTBImage) or sets of regions comprising a segmentation result (e.g., MTBRegion2DSet), but also a threshold computed or to be applied is conceivable. An operator receives zero or more input objects. Operators with zero inputs are operators which for example create an image for given parameters or read an image from file.

The behaviour of operators is controlled or configured via parameters. Any Java class is accepted as a parameter type. Typical examples for parameters are the size of a kernel or structuring element, a filter mask, constants (or parameters) which weight energy terms, step width of optimizers, or the maximal number of iterations for a gradient descent algorithm. A threshold may either be considered as an input object or a parameter of the operator.

An operator produces zero or more output objects. The types of these objects are the same as for input objects since in virtually all cases an output of one operator may act as the input to other operators. An operator with zero output objects will, e.g., write an image to disk.

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 plugins or commandline tools. Their input and output as well as 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:

- `public final void runOp( MTBOperator callingOperator)`
  `throws MTBOperatorException,MTBProcessingDAGException`

After return from that routine the results can be retrieved from the operator.

Each operator defines its interface describing input and output arguments, parameter arguments, and also supplemental arguments. Each of these items is described by an object of type `MTBOpArgumentDescriptor` specifying

- a name,
- the Java class,
- a textual explanation of this data or parameter,
- a boolean indicating whether the argument is required or not, and
- a default value, which may be '`null`.

An example how to use an operator is given in Fig. 6.1. The generic methods `setInput(...)` and `setParameter(...)` can be used to set input data and parameters and are provided by each operator. Note that setting of input data and parameters may throw an exception if wrong input or parameter names are used, or if the value to be set is of an incorrect class type. For numeric parameters the Java standard widening operations for primitive numeric datatypes are applied in analogy – see Tab. 6.1. Note that a double literal, e.g. 0.1, is not accepted for a parameter of type `Float`. Depending on the implementation of a specific operator also constructors or dedicated methods to set individual inputs or parameters may be available.

Parameter type	accepted type
Double	Double, Float, Long, Integer, Short, Byte,
Float	Float, Long, Integer, Short, Byte,
Long	Long, Integer, Short, Byte,
Integer	Integer, Short, Byte,
Short	Short, Byte
Byte	Byte

Table 6.1: Widening applied when setting parameters of an operator.

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

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

If all required input data and parameters have been set 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 `runOp()` sets all required but unset parameters and inputs to their default values. Note, if an argument equals '`null`' it is considered as unset. Subsequently the validity of parameters and inputs is checked. Validity requires for an operator that all required parameters and inputs have values different from '`null`'. In addition the implementation of an operator may impose further constraints by overriding the method

- `public void validateCustom() throws MTBOperatorException`

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

- `protected abstract void operate()`
  `throws MTBOperatorException,MTBProcessingDAGException`

is invoked. Each operator is supposed to implement this method as it does the actual work of the operator. After return from `runOp()` the resulting output data can be retrieved from the operator using the generic method

- `public final MTBData getOutput( String name )`

or, if implemented, specialized methods for this purpose. Note, the value of the operator parameters and/or inputs may have changed upon return from `runOp()` due to setting of default values

(or modification in the `operate()` method). `runOp()` may throw an exception if validation of inputs and parameters or data processing itself fails.

The argument `callingOperator` of `runOp()` should be a reference to the operator which calls this operator as a nested operation. If `callingOperator` equals '`null`' the history mechanism will find the operator called only if one of its input or output data is referenced by the enclosing operator(s). Otherwise the operator called will not be included into the processing history. The reference is always '`null`' in the top most call of an operator as in our example in Fig. 6.1.

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

In addition to parameters, input and output data, an operator may use supplemental arguments, e.g. flags to control output or debugging information as well as for returning intermediate results. By definition the setting of supplemental arguments may not influence the processing results returned as output data. Consequently, supplemental arguments are not documented in the processing history.

6.3 Implementing operators

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

- the interface of the operator,
- the operation per se,
- and constructors,

which are described in the following.

Operator interface. The interface of an operator is constituted by the parameters as well as input and output data. In addition, an operator may use supplemental arguments, 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 arguments. Hence, these are not stored in the processing history. The supplemental arguments are described in analogy to parameters.

Each operator needs to define descriptors for all parameters, inputs, outputs, and supplemental arguments. All input and output argument classes are derived from the abstract class `MTBData`. Parameters and supplemental arguments may be of any Java class. The value of each parameter is recorded upon invocation of an operator via its `runOp()` method using the method `toString()` of the parameter class for later display in the processing history.

An operator inherits all descriptors defined in super classes. If multiple descriptors with identical names are defined along the inheritance hierarchy, the last one is used, i.e. the nearest to the operator at hand.

Fig. 6.2 shows an example how descriptors are defined. Here the operator defines no parameters and supplemental arguments. The key to the mechanism adopted by `MiToBo` is the method `collectArguments()` defined in the base class `MTBOperator`. Each subclass of `MTBOperator` is required to override this method, otherwise the inheritance of arguments will be broken. As a minimum, `collectArguments()` needs to invoke the method `collectArguments()` of its direct super class. Subsequently, further arguments may be added by invoking the method

- ```
  protected final void addDescriptors(
              MTBOpArgumentDescriptor[] parameterDescriptors,
              MTBOpArgumentDescriptor[] inputDescriptors,
              MTBOpArgumentDescriptor[] outputDescriptors,
              MTBOpArgumentDescriptor[] supplementalDescriptors).
  ```

The base class of all operators, `MTBOperator`, defines as its only argument the supplemental verbose flag which is by default set to '`false`'.

In the example the parameter declarations are shown in lines $6 - 22$. Two descriptor arrays are defined for the input and output data, respectively, and are added to the interface using the method `addDescriptors()` as shown in lines 20 and 21.

An operator must never override the member variables

- `parameterDescriptorsAll`
- `inputDescriptorsAll`
- `outputDescriptorsAll`
- `supplementalDescriptorsAll`

as these are used by `MiToBo` to represent all arguments.

Optional parameters of an operator can easily be realized where the value of '`null`' indicates an unset parameter or input data. Furthermore, an operator can be implemented which receives a value either as input or as parameter. An example is a thresholding operator where the threshold may be considered as a parameter set by the user, or may result as output from an operator automatically computing a threshold.

Operator functionality. The method `operate()` implements the functionality of the operator. All data passed into and got back from the operator have to be passed via the arguments of
  ```

```
1  import de.unihalle.informatik.MiToBo.operator.*;
2  import de.unihalle.informatik.MiToBo.exceptions.*;
3
4  public class MTBMedian extends MTBOperator {
5
6    @Override
7    protected void collectArguments() throws MTBOperatorException {
8      super.collectArguments();
9
10       MTBOpArgumentDescriptor[] inputDescriptors =
11         new MTBOpArgumentDescriptor[] {
12           new MTBOpArgumentDescriptor( "inImg", MTBImage.class,
13                       "Input image", true, null) };
14
15       MTBOpArgumentDescriptor[] outputDescriptors =
16         new MTBOpArgumentDescriptor[] {
17           new MTBOpArgumentDescriptor( "resultImg", MTBImage.class,
18                       "Result image", true, null) };
19
20     addDescriptors( null, inputDescriptors,
21               outputDescriptors, null);
22   }
23
24   MTBMedian() throws MTBOperatorException { }
25
26   protected void operate() throws MTBOperatorException {
27     System.out.println( "MTBMedian::operate");
28
29     MTBImage in = (MTBImage)(this.getInput( "inImg"));
30     MTBImage res = new MTBImage();
31     // no actual work yet, add your operator code here
32     this.setOutput( "resultImg", res);
33   }
34 }
```

Figure 6.2: Example how to define the interface of an operator.

the operator which may be set and retrieved with the appropriate set and get methods. To invoke the processing within an operator, i.e. run its `operate()` routine, the final method `runOp()` needs to be called by the user of an operator.

Constructors. Each operator class requires to implement the standard constructor which might be empty as given in the example, line 24. This is required for automatic code generation capabilites. Further convenience constructors may be available which additionally set parameters, inputs, and/or outputs.

6.3.1 Helpful tools for operator development

There is a tiny commandline tool to query the interface of a compiled operator by invoking e.g.

```
java cmdTools.development.PrintOperatorInterface
                    de.unihalle.informatik.MiToBo.morphology.ImgDilate
```

from the commandline which yields

```
1  Interface of MTBOperator: ImgDilate
2      Param <masksize> required, type: Integer
3      Input <inImg> required, type: MTBImage
4      Output <resultImg> required, type: MTBImageByte
5      Supplemental <verbose> not required, type: Boolean
```

Another commandline tool, `cmdTools.development.GenerateGetterSetter`, prints standard getter and setter methods onto standard output which can easily be included into the Java source code.

Chapter 7

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. All 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 two 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. 3). To log all data manipulations taking place while running through an image analysis pipeline, MiToBo needs to monitor all operator calls and associate them with the data objects manipulated by the operators. To this end the data objects need to provide a pool of basic properties and methods which need to be specified in some common base class that does not exist in ImageJ.

7.1 Basic concepts of data types in MiToBo

Image processing pipelines in MiToBo build on the idea of operators that manipulate data objects. According to the specification of MiToBo operators (Chapt. 6), data objects that are to be

manipulated by a certain operator have to be passed to this operator as its input objects. Resulting data objects will be passed back to the user as output objects of the operator. Both types of objects share the property that they have to be of type `'MTBData'`. As mentioned above this is necessary to allow `MiToBo` to link the data objects to the manipulating operators, and based on this information to later on build up the history graph representation.

The class `'MTBData'` is an abstract class to be found in the package `'mitobo.operator'`. It basically defines the framework for self-documentation and provides the data objects with methods to save their processing history to an XML file or read the history from such a file:

- `public void writeHistory( String filename )`
  writes the object's processing history in XML format to the specified file; the file can be opened with `mtbchisio` to view the history graph (Appendix A)
- `public void readHistory( String filename )`
  reads the processing history of an object from the specified file

As outlined above, all data types to be used as input or output objects of `MiToBo` operators need to be subclasses of 'MTBData'. Apart from that, however, there are no restrictions on the implementation neither any other specifications that need to be met by the programmer. Accordingly, at this point `MiToBo` implements common functionality for all data objects while at the same time still granting a maximum of flexibility to the programmer.

However, the common data type interface for self-documentation is not the only new feature of `MiToBo` data types. As another optional feature `MiToBo` offers the concept of data type *properties* to the programmer for further characterizing certain objects in the processing history and also in general. Further, it supports easy ways to read and write data objects from and to files by using XML representations and XML beans. Both features will be outlined in more detail in the following two sections. The last section will give a brief overview of `MTBImage`, the basic image data type of `MiToBo`.

7.2 Data type 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.

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

Table 7.1: Examplary properties and its values for an object of type `MTBImage`.

Additional information can be added to input and output objects by defining object *properties* which 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 `mtbchisio`, 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 7.1.

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 convinience 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, it is, however, not automatically set. This task 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.

7.3 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.

7.3.1 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'

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

(cf. Chapt. 2). 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 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.

7.3.2 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 7.1 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

```
<!-- 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 7.1: Example XML schema for the data type `MTBContour2DSet`.

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.

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. 7.1. Below some XML schemata files are imported, e.g. the

```
<?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 7.2: Examplary contents of the file `MTBXML.xsd`.

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 `$MITOBO/intjars/`. The option `-src` denotes the root path to where the source files are copied. Given this path and the target namespace as defined in the XML schemata files the sources will be placed in `$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 otherwise 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 7.3

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 explicitly 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 7.4. Initially an object 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.

```
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 7.3: XML wrapper class based write method for `MTBContour2DSet`

```
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 7.4: Example method for translating `MTBContour2D` types into XML

7.4 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 and thus documentation of the history or processing chain of an image

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.

7.4.1 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 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 7.4.3.
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.

7.4.2 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 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
- `MTBImageShort` for `short`-type pixel values
- `MTBImageInt` for `int`-type pixel values
- `MTBImageFloat` for `float`-type pixel values
- `MTBImageDouble` for `double`-type pixel values
- `MTBImageRGB` for three-dimensional `byte`-type pixel values, one for each color channel red, green and blue

These image types share the same interface, but we have to differentiate between `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. As long as you are working with `MTBImage` only, there is no difference between them. But if you have to fall back to `ImagePlus`(e.g. when displaying images), you should keep in mind the difference described in the following two paragraphs.

MTBImages with corresponding ImageJ types. If you change the values of an `MTBImage` that simply wraps a corresponding `ImagePlus`, the changes are applied to that `ImagePlus` directly, because `MTBImage` and `ImagePlus` share the same data arrays. Table 7.2 lists the subtypes of `MTBImage` and their corresponding ImageJ image types.

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

Table 7.2: `MTBImage` types with corresponding ImageJ types.

MTBImages without corresponding ImageJ types. `MTBImage`s which cannot be represented by corresponding ImageJ types keep their own data arrays and are not linked to an `ImagePlus` object. 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 of another `MTBImage` to that datatype. Nevertheless an `ImagePlus` object is often needed, usually for visualization. `MTBImage` provides the function `getImagePlus()` to an `ImagePlus`. 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.
Table 7.3 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 7.3: `MTBImage` types without corresponding ImageJ types.

7.4.3 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 `MTBImage`s 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 (let's call it *Slice*) in this array (called *Stack*) are referenced by 1 to N, where N is the number of slices. `MTBImage` uses always indexing that is known by every programmer, starting from 0 to $(N_{\text{dim}} - 1)$.

- `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 an "actual" slice or z-stack with the following functions:

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

7.4.4 `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 `WriterMTBImage` and `ReaderMTBImage` 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 TIFF format, and another file in GraphML format, which is an XML-based format to store the image processing history. This file is automatically loaded when the image is opened using `MiToBo`'s `ReaderMTBImage` operator. The processing history files can be examined using `mtbchisio` (Appendix A).
`MiToBo` uses the ImageIO-Ext library (https://imageio-ext.dev.java.net/) to read and write 64-Bit floating point (`double`) TIFF images. This library depends on the Java Advanced Imaging (JAI) library (https://jai.dev.java.net/) and the JAI ImageIO library (https://jai-imageio.dev.java.net/). The latter defines readers for many formats, but does not support 64-Bit TIFF images. To remove dependencies from the JAI library, `MiToBo` provides a modified version (`mtb-image-io-ext.jar`) of the TIFF part of the ImageIO-Ext library to remove dependencies from the JAI library. The JAI ImageIO library is still required though.

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.

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. 7). 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(null);
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 only passes its parameters to exactly one operator and no other operator inside the plugin modifies the input image.

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: The input descriptor of the plugin is implicit defined in the `MTBOperatorPlugInFilter` class. By default, the input image is stored in an `MTBImage` object 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 callec 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 parameter 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      protected void collectArguments() throws MTBOperatorException {
10         super.collectArguments();
11         // define input, output, parameters and supplemental arguments
12         // if some arguments not exist, they can be leaved out
13         MTBOpArgumentDescriptor[] outputDescriptors =
14           new MTBOpArgumentDescriptor[] {
15             new MTBOpArgumentDescriptor(
16             "resultImg", MTBImage.class, "Filtered_image", true, null)
17           };
18         // add descriptors
19         addDescriptors(null, outputDescriptors, null, null);
20     }
21
22     /**
23      * Standard constructor.
24      */
25     public MTB_Median_MTBPlugin() throws MTBOperatorException {
26         // nothing to do here
27     }
28
29     /**
30      * Implementing standard ImageJ setup method.
31      */
32     @Override
33     public int setup(String arg0, ImagePlus imp) {
34         // allow 8 and 16-bit gray value images
35         return DOES_8G + DOES_16;
36     }
```

```
37      @Override
38      protected void operate() throws MTBOperatorException,
39          MTBProcessingDAGException {
40
41          // maybe call some other operators
42
43          // crate a new median operator object and set the input via its constructor
44          MTBMedian medianOp = new MTBMedian(this.inputImage);
45          // invoke the operator
46          medianOp.runOp(null);
47          // set plugin output from median filter result
48          this.setOutput("resultImg", medianOp.getResImage());
49      }
50
51      /**
52       * Implementing standard ImageJ run method.
53       */
54      @Override
55      public void run(ImageProcessor impIP) {
56          try {
57              // set the input data
58              this.setInput("inImg", this.mtbInput);
59              // call the operate method to call additional operators or operations
60              this.runOp(null);
61              // get the output of the operator, in this case the filtered image
62              MTBImage resultImage = (MTBImage) this.getOutput("resultImg");
63              // display the filtered image via ImageJ
64              resultImage.show();
65          } catch (MTBOperatorException e) {
66              e.printStackTrace();
67          } catch (MTBProcessingDAGException e) {
68              e.printStackTrace();
69          }
70      }
71  }
```

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 {
            ReaderMTBImage IO = new ReaderMTBImage(otherArgs[0]);
            IO.runOp(null);
            MTBImage inImg = IO.getResultImage();

            ImgThresh thresOp = new ImgThresh();
            thresOp.setInput("InputImage", inImg);
            thresOp.setParameter("Threshold", threshold);
            thresOp.setParameter("FGValue", fgValue);
            thresOp.setParameter("BGValue", bgValue);
            thresOp.runOp(null);

            MTBImage resultImg = (MTBImage)thresOp.getOutput( "ResultImage");
            WriterMTBImage writer= new WriterMTBImage(otherArgs[1], resultImg);
            writer.runOp(null);
        } 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 `ReaderMTBImage`. Subsequently the thresholding operator `ImgThresh` is initialized and invoked. While for the `ReaderMTBImage` 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 the `getOutput()` method) and then saved to disk by using the operator `WriterMTBImage`.

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 will rather include only all explicit operator calls. The fact that the tool itself is not an operator is also the reason for passing '`null`' to all `runOp()` calls inside the `main()` routine in this example.

Chapter 10

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`.

10.1 Plugin Configuration

For user specific configuration of plugins MiToBo supports environment variables and JVM properties as well as ImageJ preferences (Section 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: `mtbchisio`

Processing histories are stored in XML format using *graphml* with some `MiToBo` specific extensions as mentioned in Chapter 4. To display histories we extended Chisio[1] to handle the `MiToBo` specific extensions yielding `mtbchisio`.

A.1 Installation and invocation of `mtbchisio`

`mtbchisio` is not strictly part of `MiToBo` but supplied as an add-on at the `MiToBo` website[2]. A single zip-file is provided for running `mtbchisio` 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 `mtbchisio`, e.g., by

```
java org.gvt.ChisioMain [directory]
```

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

A.2 Using `mtbchisio`

`mtbchisio` is based on Chisio, a free editing and layout tool for compound or hierarchically structured graphs. In `mtbchisio` 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 `mtbchisio` 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] www.informatik.uni-halle.de/mitobo

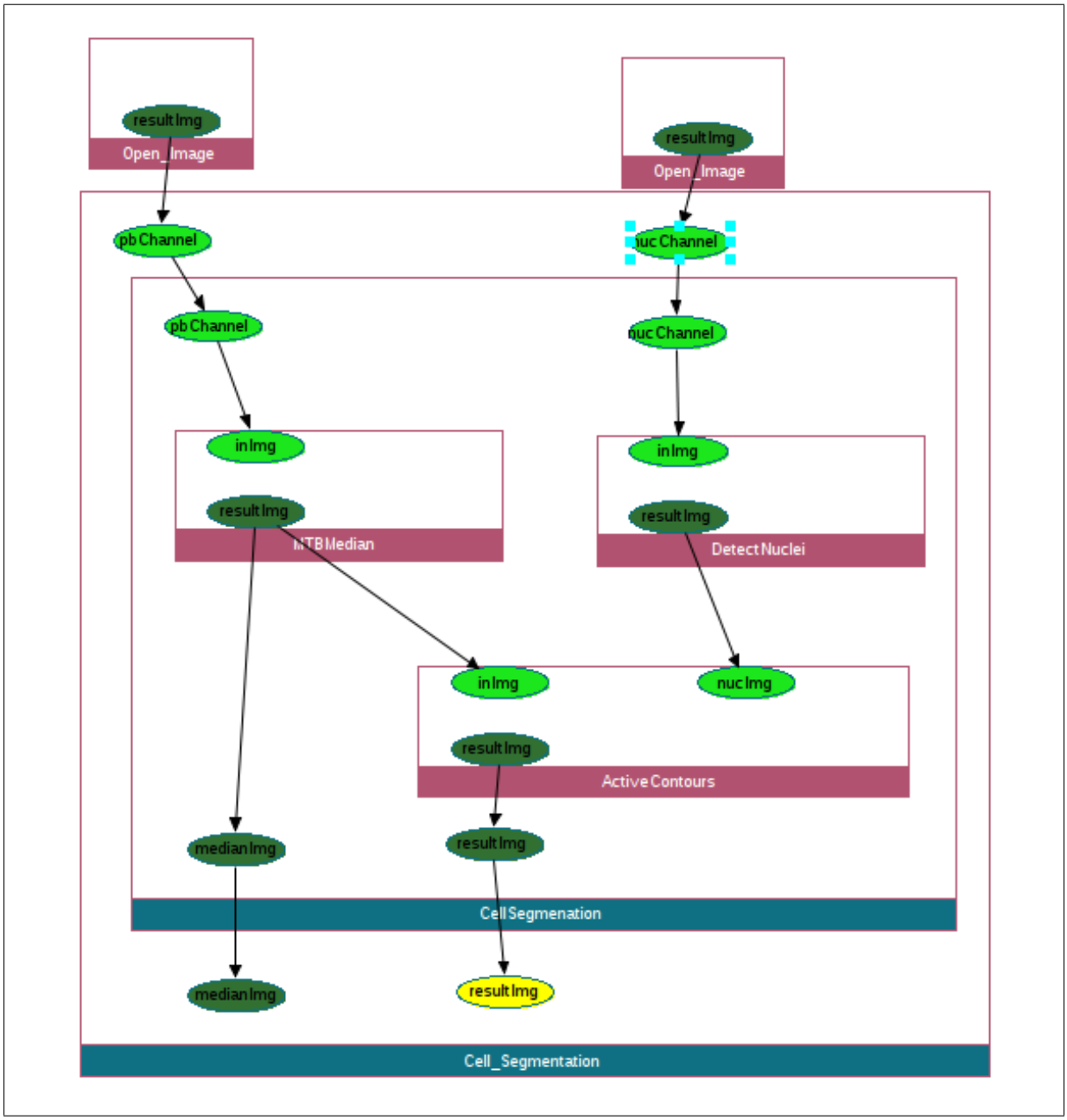


Figure A.1: Screen shot of `mtbchisio` for the output port named `resultImg` of the operator `Cell_Segmentation`. Some operators are collapsed indicated by dark red operator names.

Chisio's functionality and the extensions supplied by `mtbchisio`. For more details on Chisio see the User's manual of Chisio which is included in the `mtbchisio` package and also easily found in the web.

As already described in Chapter 4 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 in Fig. A.1 where the nested operators within `CellSegmentation` have been collapsed. This example is very similar to the one discussed in Section 3, Fig. 3.1. The only difference is, that images are not read directly but using the operator `Open_Image`. Implementing input or output of data as operators is adequate, if these I/O operations depend on parameters, e.g. the selection of channels or set of images from a stack to be read. In Fig. A.1 these reading operators are collapsed, too. 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 thus any manual layout adjustments applied before are lost.

If we uncollapse both operator instances of the operator `Open_Image`, we find the the source of `pbChannel` was read from the file `pb.pgm` which has no prior history. The later fact is visually marked in Fig. A.2 by the grey shading of the corresponding triangular data port. On the other hand, `nuc.pgm` which is passed to `Cell_Segmentation` via the `nucChannel` port has a processing history associated which was read from the `.mph`-file accompanying the image data in `nuc.pgm`. This is indicated by the orange color of the data port. From this processing history we find the the median operation has been applied to an image `foo.pgm` which in turn had no associated history.

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 input port `resultImg` of the operator `Cell_Segmentation`.

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 by a double mouse click. Information displayed includes

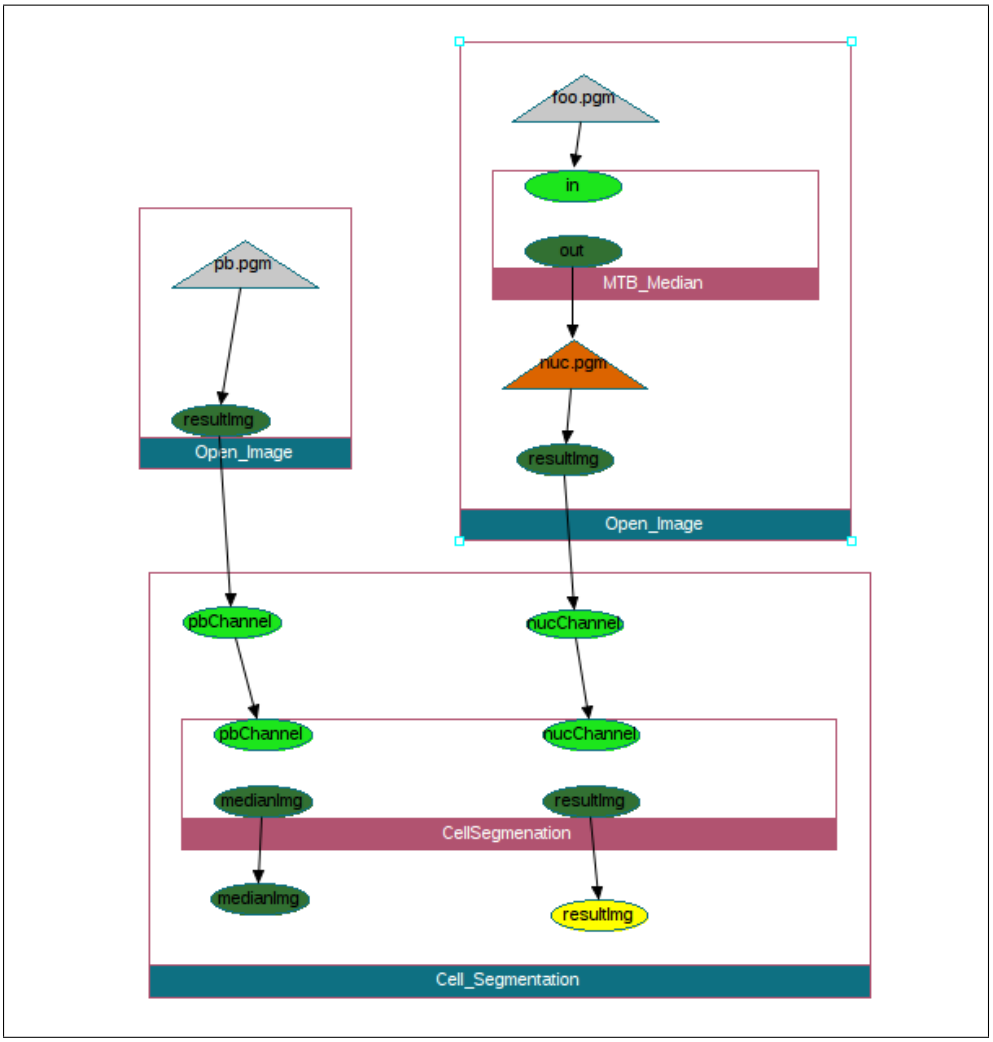


Figure A.2: Screen shot of `mtbchisio` for the same processing history as shown in Fig: A.1, however with a different set of collapsed operator nodes.

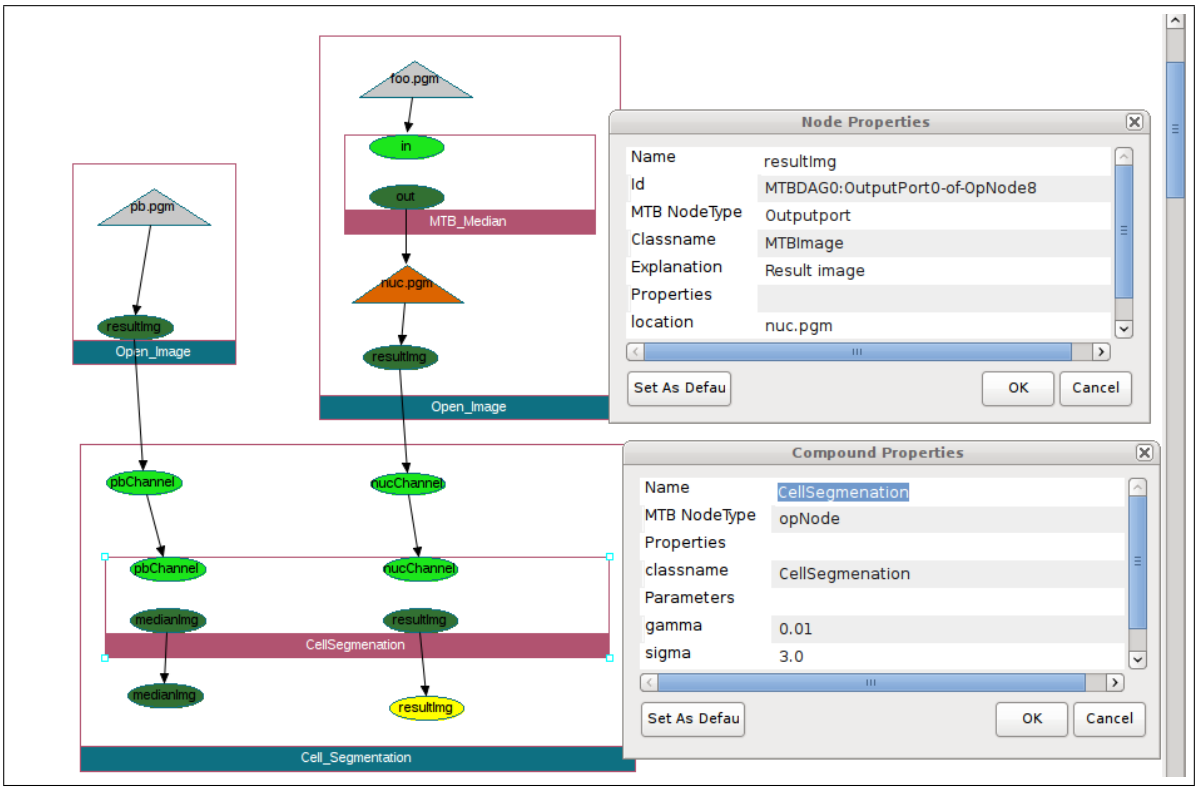


Figure A.3: Screen shot of `mtbchisio` with details for the operator `Cell_Segmentation` and the output port `resultImage` of one instance of the operator `Open_Image`.

- 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 `Open_Image` as it passed into our along with the explanatory text of this port
- for output ports the properties of the `Open_Image` valid when pass out of the operator.

In Fig. A.3 this is shown for the operator `Cell_Segmentation` and one output port with name `resultImage`.