A block-model and an intrinsic visual editor for ImageJ workflows

Aliakbar Jafarpour

jafarpour.a.j@ieee.org

Abstract: Here we introduce a block (XML) model for a typical functionality of the image analysis program ImageJ. Such cascadable blocks and their associated interconnections are then used to define executable workflows for generic application-level programs. This model brings programming with ImageJ to a simpler and a more intuitive level and enables end-users to design workflows with more clarity, less effort, and no knowledge of (Macro) programming. It avoids unnecessary fusion of different types of prior knowledge and makes end-users more independent in generation, interpretation, and validation of results. The program is available at https://github.com/nurlicht/ImageJ-Workflow

1. Introduction

The Program *ImageJ* provides end-users and especially cell biologists with invaluable image analysis functionalities. Repeated use, extension, and adaptation of these functionalities often requires a special-purpose *application-level* program; typically an ImageJ Macro.

Writing a Macro is a relatively high-level programming task. Many end-users with little or no knowledge of programming need only a short training before writing their Macros for *simple* tasks. In reality, handling, analysis, and interpretation of important real-life images is associated with additional challenges. While the Macro language can still be used in such cases, it requires less trivial algorithms and data structures and also the (indirect) use of ImageJ classes. This is a serious challenge for the average end-user. On the other hand, clear understanding and controlled use of prior biological knowledge for developing a Macro or the interpretation of results is a challenge for a programmer trying to develop a Macro for the end-users.

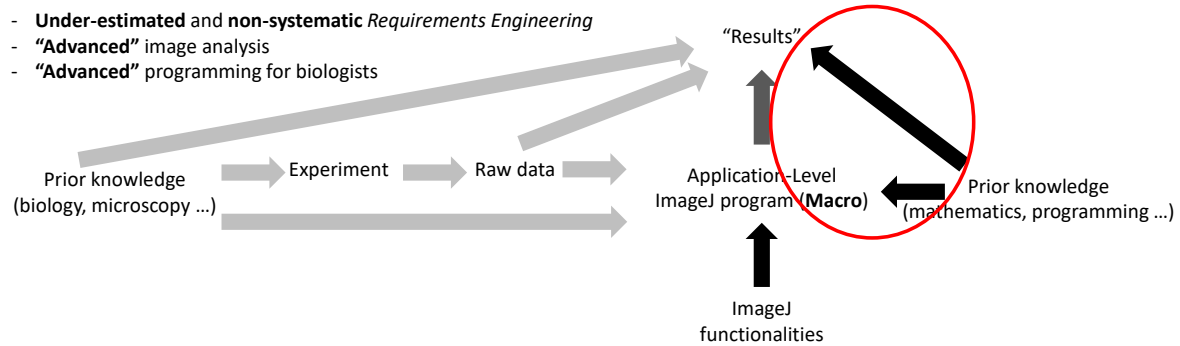

Figure 1: *Conventional* information flow diagram depicting the development and use of application-specific (bio-) image analysis programs. Generation, interpretation, and validation of Results depend on nontrivial communication between programmers and biologists.

The Macro language is a powerful and helpful tool developed in a language-centered (Java-centered) design. The goal here is 1) to develop an alternative programming environment in an

application-centered design, and 2) to separate the assignments of professional programmers from those of end-users (in terms of Requirements Engineering, validation, use of prior knowledge …)

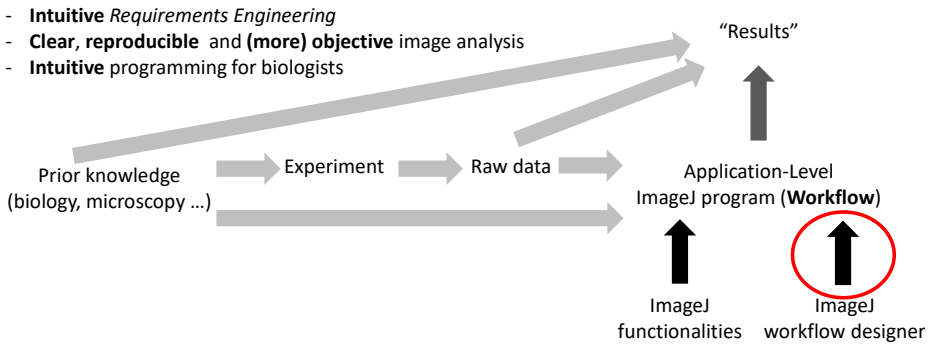

Figure 2: *Proposed* information flow diagram depicting the development and use of application-specific (bio-) image analysis programs. Generation, interpretation, and validation of Results are done by the end-user. Programmers focus on developing easy-to-use processing blocks without unnecessary overlap with end-users.

There is already at least one example for such a programming environment; KNIME. However, here we seek to develop a light-weight environment without unnecessary components or unfamiliar interfaces that can be executed intrinsically (by ImageJ itself) as a Java Plugin. Furthermore, the model used here is based on speicifc inputs/outputs encountered in ImageJ, and this special-purpose design guides the user through known meaningful steps, rather than offering too many choices.

The main focus in this contribution is the block diagram model ("business logic"). The current version of the program supports some blocks associated with commonly-used functionalities, as exemplified in Figure 3. Creation of new blocks and use of existing programs are addressed in Sections 5 and 9, respectively.

Additional functionalities for visualization and interaction with user have also been provided in the current version for the sake of completeness. These parts, however, can be easily replaced or developed further without changing the model (a Model-View-Controller-like architecture).

2. The Block-Diagram model

A typical functionality in ImageJ (say, Median Filtering or Particle Analysis) is modelled as a *Block;* with its inputs and outputs as *Pins*. A pin can assume only a few basic types of objects typical of ImageJ functionalities; namely 1) an image; 2) a set of ROIs; 3) a table; 4) a text; and 5) a file.

The collection of all input- or all output-pins forms a *Port*; thereby modelling a Block with two (*input* and *output*) ports. A *Link* connects a pin of an output port to a pin (of similar type) of an input port. The collection of all Blocks and interconnecting Links forms a *BlockDiagram*. Figure 3 illustrates a block diagram and the associated functionalities. Figure 4 illustrates the same dependency between corresponding programming elements (classes) of *Pin*, *Port*, *Block*, *Link*, and *BlockDiagram*.

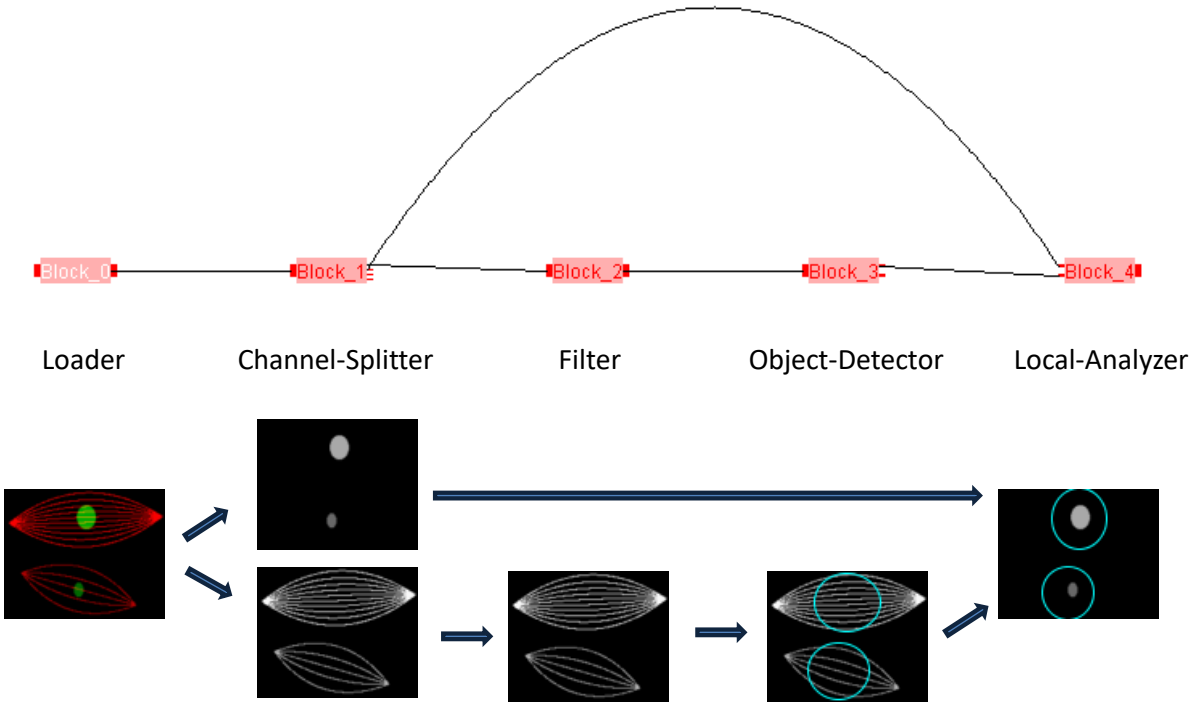

Figure 3: (Top) The block-diagram of a common bio-image analysis workflow, and (Bottom) Cartoons illustrating block functionalities. The goal is to identify and to analyze green spots. However, if only the green spots are associated with a red structure. So, the (bigger) objects in red should be first detected, and corresponding green spots can only then be analyzed.

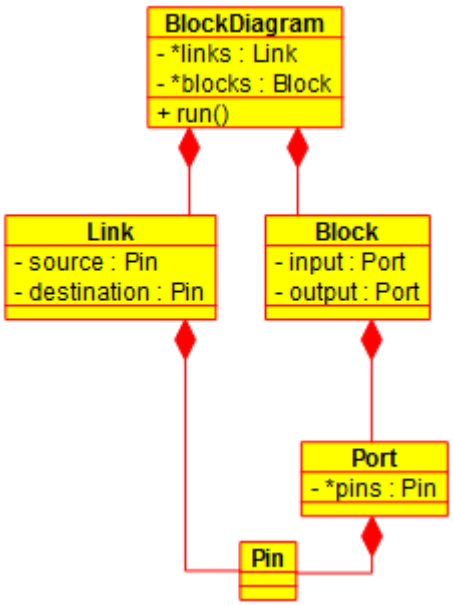

Figure 4: Class diagram illustrating the employed programming functionalities (classes) and their associations.

3. Major aspects of a Block

The major aspects of a block are its structure, behavior, and execution, as depicted in Figure 5. The class Block includes some structural information. However, some of the codes related to the

structure and the entire code related to behavior and execution have been outsourced (implemented by other classes and their subclasses).

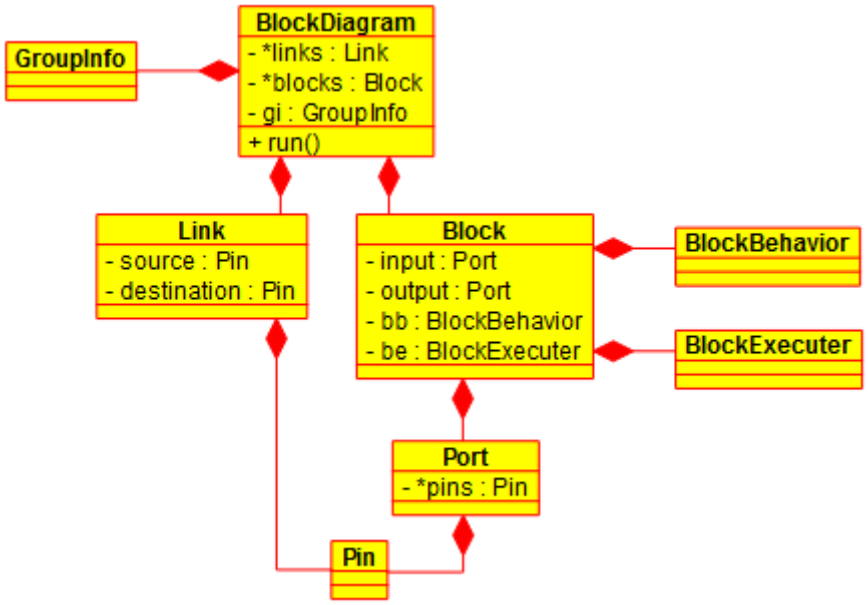

Figure 5: A more complete Class diagram (compared to Figure 4) illustrating the additional outsourced functionalities of a block; *BlockBehavior* and *BlockExecuter*. This diagram also shows the class *GroupInfo*; used in group-analysis of multiple images in multiple folders with a given block diagram.

3.1. Structure

The number of images, ROI sets, Tables, Texts, and Files at each port are quantified with the *degeneracy* vector. A Channel Splitter block, for instance, has only one input pin of type image and some (say 4) output pins of type image. The degeneracy vectors are di = {1, 0, 0, 0, 0} and do = {4, 0, 0, 0, 0}. A Particle Analyzer block has an input pin of type image and two output pins of type Roi Set and Table, respectively. The corresponding degeneracy vectors are then di = {1, 0, 0, 0, 0} and do = {0, 1, 1, 0, 0}. For the Loader and Saver blocks, the user can decide arbitrary degeneracies for one port (The degeneracy of the other port will be the same).

3.2. Behavior

This is where the functionality of a block is defined. As in the rest of the program, significant emphasis has been put on class inheritance to facilitate readable, flexible, and extensible object-oriented programs. The *BaseClass* is first inherited by the concrete class *BlockBehavior* to implement the functionalities needed for all possible "behaviors". It is then inherited by the class *FileBehavior*, before being inherited further by *SaverBehavior* and *LoaderBehavior*. Similarities between other blocks have also been captured to form additional levels of class inheritance.

Setting the parameters of a block can be done in three stages: 1) fixing parameters by the programmer in class definitions; 2) setting parameters by the workflow-designer ("process-developer") while creating a block, and 3) setting parameters by the end-user when running a block-diagram.

The first two ways of setting parameters can be done with the *configure()* method of the class BlockBehavior (with normal or *null* arguments, respectively). The third can be done with the class *run()* of the same class.

3.3. Execution

The functionalities for managing the execution of a block (waiting for all input pins to receive valid signals before running the block) have been implemented in the class BlockExecuter. An instance of this class needs to be added to any instance of a Block (sub-) class.

At the beginning (after a reset of the block diagram), only a Loader block has a ready input port. All other blocks should wait until some data (usually images) are loaded and processed by preceding blocks.

In principle, the execution of a block diagram can be done in two ways; comparable with compiled and interpreted languages. The program encoded by the blocks and links may be decoded and converted to an ordinary (Java) program. Alternatively, as done here, the decoded program may be run by executing individual blocks and delivering generated signals (via links) to other blocks.

3.4. Beyond a simple run

An important aspect of this application-oriented design is to apply the same workflow to a group of images. Group analysis is facilitated by a class *GroupInfo* for keeping track of input and output files. Other considerations, to be implemented in the future versions, are iterative and (run-time-) interactive blocks.

4. Categories of Blocks

A generic block can accept different types of data at its pins. However, it is more convenient to categorize blocks into {Loader, Saver, Operator}. A Loader block is the only type that accepts files or folders (and nothing else) as input. A Saver block is the only block that accepts files or folders (and nothing else) as output. An Operator block accepts files neither as input, nor as output. Some of the benefits of this categorization are as follows:

- In Group Analysis of multiple images in a folder, only Loader and Saver blocks need to keep track of successive files. All other blocks, simply operate on input data (and are reset by the Block Diagram after each iteration), without keeping track of or even knowing about repeated operations.
- The functionalities for handling files are limited to a few classes, rather than being repeated in unlimited number of classes.
- An Operator block can have different numbers of pins at its input and output ports. Loader and Saver blocks; however, have the same number and corresponding type of data at input and output. Once the number of pins at one port is specified, the number/types of pins of the other port can be figured out automatically.

5. Creating a new Block

One simply needs to add ports; add a *BlockBehavior*; and add a *BlockExecuter* to a new sub-class of *Block*, as depicted in Figure 6. *Block-Executer* does not need any customization, but it is possible to customize it for other problem-specific purposes. The most important features for creating a new *BlockBehavior* are overriding the two methods *config()* and *run()*.

With the same fixed behavior, the structure of a block may still be customized (dynamic structure). For some Blocks, the number/types of pins are fixed (ex. MedianFilter). For some others, the number/types of pins can be set while defining a block diagram (ex. Loader and Saver). A dynamic Structure is achieved when the method *addPort()* in the Block (sub-)class is used with null argument for the port degeneracy.

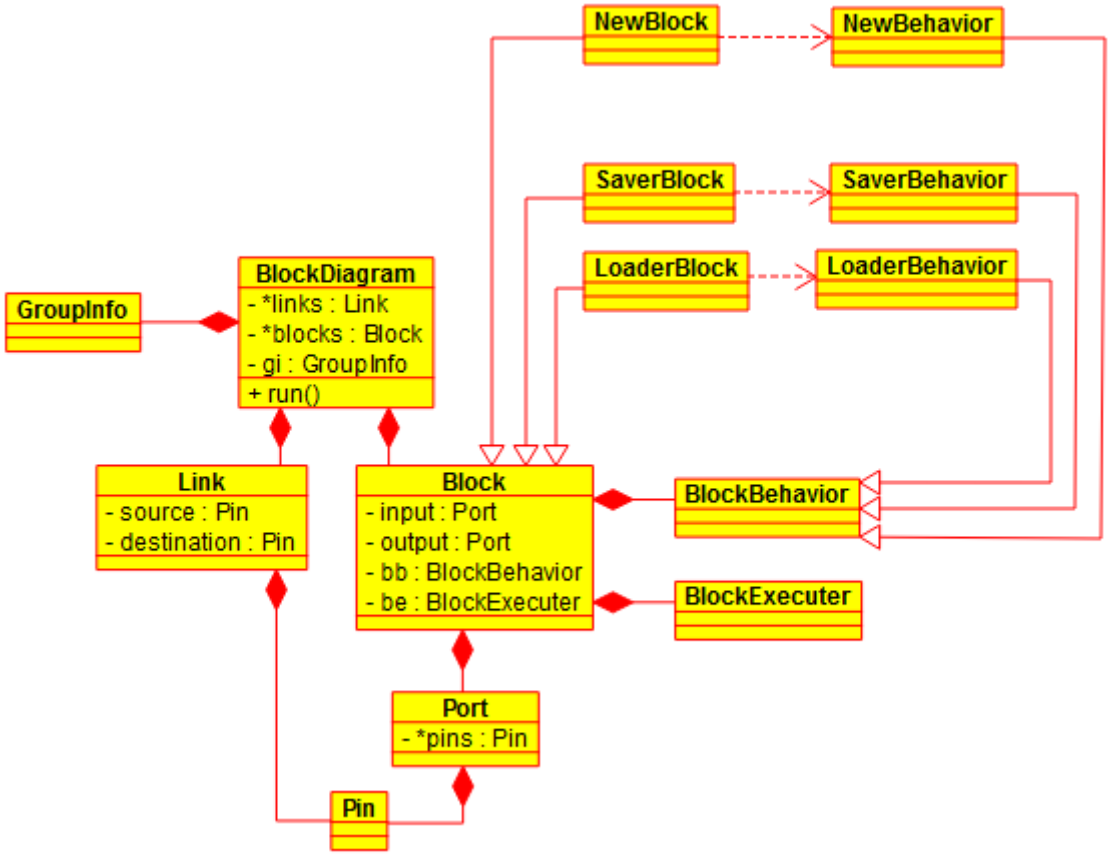

Figure 6: A more complete Class diagram (compared to Figure 5) illustrating the inheritance of the parent-class Block by sample child-classes LoaderBlock, SaverBlock, and NewBlock. Also shown are sample child-classes LoaderBlockBehavior, SaverBlockBehavior, and NewBlockBehavior. The classes shown here are the major components of the functionalities needed *to model* an application-level ImageJ program ("business logic").

6. Visualization and guided design

Visualization of a given block is facilitated by the corresponding *BlockComponent*, as depicted in Figure 7. BlockComponent performs the required calculations to show a given number of input/output pins and connecting a link to the right pin. Functionalities for handling mouse-events have also been implemented, but not used in the current version (favoring guided design with many multiple-choice dialog boxes).

At the heart of visualization is the class *BlockDiagramVisualizer*. The positions of blocks and links are calculated automatically and used to display them in the form of an *Arc-Diagram*, as shown in Figure 3 (Top). Consequent blocks, however, are connected using straight lines (rather than arcs). Given the mouse-selectability of block components, it is possible to modify the visualization for user-editable positioning of blocks and links.

Figure 7 shows the most complete version of the class diagram. The majority of classes can be partitioned into three groups bordered by blue lines and labeled with M (Model), V (View), and C (Controller); reminiscent of the Model-View-Controller design pattern.

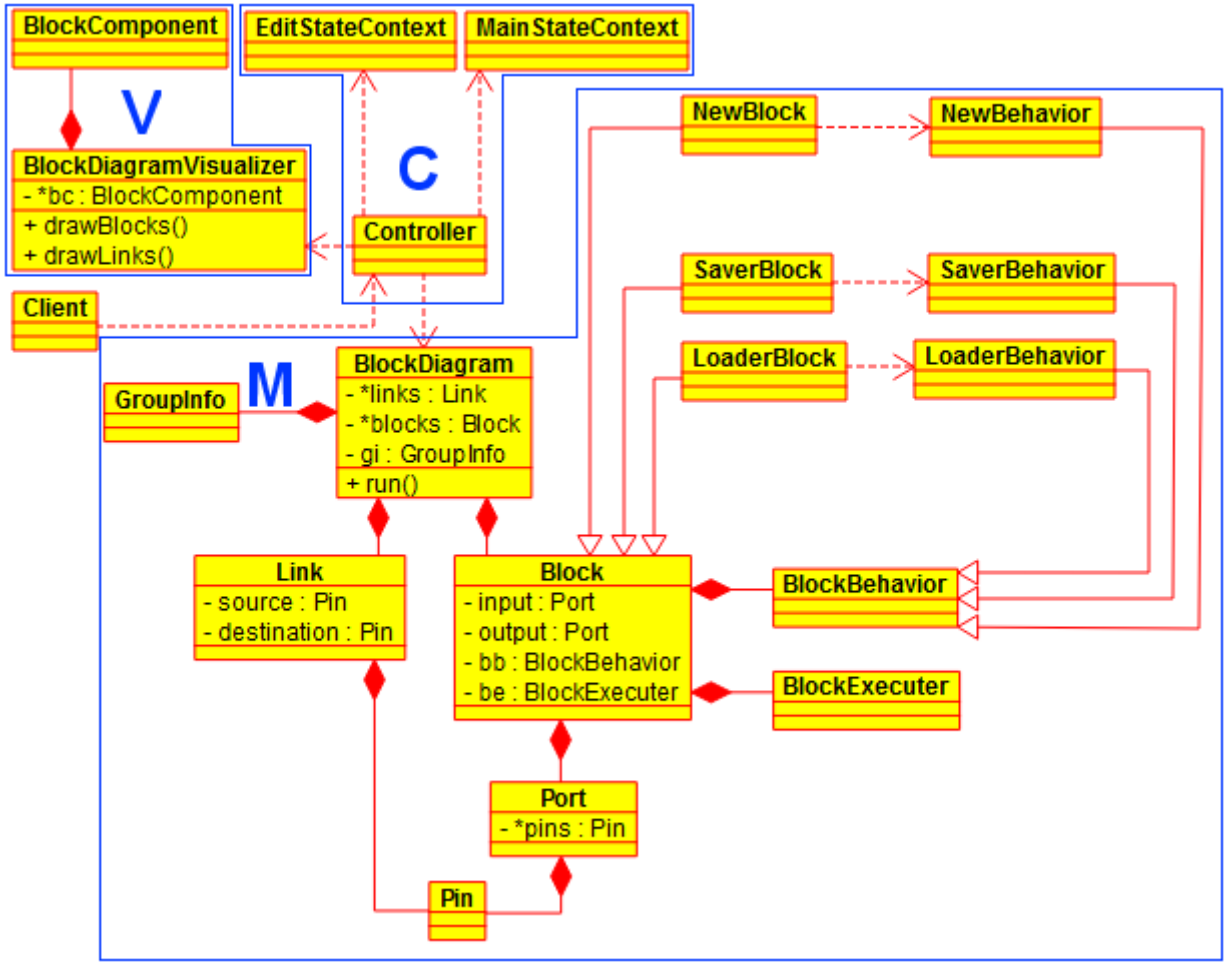

Figure 7: The most complete Class diagram (compared to previous three figures) illustrating the additional functionalities needed *to view* (V) and *to control* (C) a block diagram. The possible states of the main menu (loading/creating a block diagram, editing it, running it …) and the possible states of editing a block diagram (adding/deleting blocks and links …) are implemented with the State Design Pattern and demonstrated here only with two representative classes *MainStateContext* and *EditStateContext*. Additional functionalities are represented with the class *Controller*. Visualization of a block diagram is achieved using classes that are represented here with *BlockComponent* and *BlockDiagramVisualizer*. The majority of classes can be partitioned into three groups bordered by blue lines and labeled with M (Model), V (View), and C (Controller); reminiscent of the Model-View-Controller design pattern. For the sake of simplicity and clarity, only major classes and interconnections have been displayed.

7. XML model of a block diagram

All classes Block, Port, and Pin have a string array called *params* (adjustable) and an associated string array for *params* labels (fixed). The XML code for any object type starts with its name, the name of its associated objects; the *params* array (with the *params* label as a *tag* and *params* value as the *text content*); followed by its *child* objects (ports under a block and pins under a port). Keeping track of related objects by their names reduces the necessity of multiple objects having a copy of each other.

The XML model includes the full structural properties of a BlockDiagram. Furthermore, it includes some behavioral properties as well. The *params* array of a pin of a Loader block, for instance, includes the selected file for loading. Saving and reusing such pieces of information makes it possible to run a program by loading an XML file without asking for what the user had specified before. The *params* array and hence XML file include execution information, as well (for example, if the input port is ready). These pieces of information are obviously problematic after loading a (previously

executed) block diagram. The class *BlockDiagramInitializer* makes sure that irrelevant execution data from previous runs are reset before running a loaded XML file.

The two classes *EncodeXML* and *DecodeXML* can be used to save a BlockDiagram as (the content of) an XML file and vice versa. Appendix I demonstrates a sample XML model of a block diagram.

8. Object-oriented design considerations

An MVC-like architecture; two cases of the State design pattern; several cases and layers of inheritance (including a BaseClass); and separation of concerns (ex. outsourcing block functionalities to BlockBehavior and BlockExecuter) are important aspects of the currently-implemented design.

Extensive inheritance used here has negligible computation cost at "run-time" (while defining or while executing the block diagram).

9. Use of existing Plugins and Macros

Developing a BlockBehavior for a new block can be simplified (or literally bypassed) by simply running an installed Plugin or Macro; rather than coding the actual functionality. In the case of filters (modifying a given image without deleting or generating new images/ROIs/Tables), the life is easy. In other cases, new images/ROIs/Tables may be generated that do not have a corresponding pin in any block. Using ImageJ classes such as *WindowManager*, *ResultsTable*, and *RoiManager*, it is possible to detect such new elements.

10. Future work

Validation of a block diagram (before running) and interactive inspection of results (after running) have been considered in the design and have been implemented to some extent. Further developments in these areas have high priorities.

The possibility of using pre-defined block diagrams as a new user-defined block (similar to a sub-vi in LabView or a sub-system in Matlab-Simulink) can be also be crucial for many users. The Composite design pattern can be helpful here.

A visual workflow can also facilitate process development in adaptive microscopy. Combined with the RXTX Java library and a serial-USB adapter, new blocks supporting USB interfacing can be developed.

Contributions

A.J. conceived and elaborated the idea; compiled and organized the Requirements; designed and continuously improved the software architecture; designed and implemented the required algorithms; performed several tests; and wrote the manuscript.

Acknowledgements

A.J. is grateful to the researchers at the *Zentrum für Molekulare Biologie der Universität Heidelberg* (ZMBH) and especially Dr. Holger Lorenz and Mr. Christian Hörth (from the ZMBH Imaging Facility) for being acquainted with problems in (image analysis of) cellular structures and processes.

Appendix I: Typical XML Model

Figure 8 shows a simple block diagram comprising a Loader and a Channel Splitter block and a link in-between. The associated XML model follows.

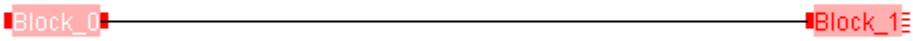

Figure 8: The XML model in this Appendix corresponds to this block diagram (including a Loader and a ChannelSplitter)

```
<?xml version="1.0" encoding="UTF-8"?>
<BlockDiagram>
 <Block>
  <Name>Block_0</Name>
  <Type>Loader</Type>
  <Class>LoaderBlock</Class>
  <Port>
   <Pin>
    <Name>Pin_0</Name>
    <Port_Name>Port_0</Port_Name>
    <Block_Name>Block_0</Block_Name>
    <Number_Links>0</Number_Links>
    <IO_Type>Input</IO_Type>
    <Valid_Structure>0</Valid_Structure>
    <Valid_Configuration>0</Valid_Configuration>
    <Rx_Ready>1</Rx_Ready>
    <Rx_Valid>0</Rx_Valid>
    <Tx_Ready>1</Tx_Ready>
    <Tx_Valid>0</Tx_Valid>
    <Data_Type>Images</Data_Type>
    <Signal_Type>String(Path_Images)</Signal_Type>
    <Object_To_String>D:\VP_Data\Multi_3DHyper.tif</Object_To_String>
    <Root_Folder>null</Root_Folder>
   </Pin>
  </Port>
  <Port>
   <Pin>
    <Name>Pin_1</Name>
    <Port_Name>Port_1</Port_Name>
    <Block_Name>Block_0</Block_Name>
    <Number_Links>0</Number_Links>
    <IO_Type>Output</IO_Type>
    <Valid_Structure>0</Valid_Structure>
    <Valid_Configuration>0</Valid_Configuration>
    <Rx_Ready>1</Rx_Ready>
    <Rx_Valid>0</Rx_Valid>
    <Tx_Ready>1</Tx_Ready>
    <Tx_Valid>0</Tx_Valid>
```

```
    <Data_Type>Images</Data_Type>
    <Signal_Type>Images</Signal_Type>
    <Object_To_String>img[Multi_3DHyper.tif (512x512x3x10x3)]</Object_To_String>
    <Root_Folder>null</Root_Folder>
   </Pin>
  </Port>
 </Block>
 <Block>
  <Name>Block_1</Name>
  <Type>Operator</Type>
  <Class>ChannelSplitterBlock</Class>
  <Port>
   <Pin>
    <Name>Pin_2</Name>
    <Port_Name>Port_2</Port_Name>
    <Block_Name>Block_1</Block_Name>
    <Number_Links>0</Number_Links>
    <IO_Type>Input</IO_Type>
    <Valid_Structure>0</Valid_Structure>
    <Valid_Configuration>0</Valid_Configuration>
    <Rx_Ready>1</Rx_Ready>
    <Rx_Valid>0</Rx_Valid>
    <Tx_Ready>1</Tx_Ready>
    <Tx_Valid>0</Tx_Valid>
    <Data_Type>Images</Data_Type>
    <Signal_Type>Images</Signal_Type>
    <Object_To_String>img[Multi_3DHyper.tif (512x512x3x10x3)]</Object_To_String>
    <Root_Folder>null</Root_Folder>
   </Pin>
  </Port>
  <Port>
   <Pin>
    <Name>Pin_3</Name>
    <Port_Name>Port_3</Port_Name>
    <Block_Name>Block_1</Block_Name>
    <Number_Links>0</Number_Links>
    <IO_Type>Output</IO_Type>
    <Valid_Structure>0</Valid_Structure>
    <Valid_Configuration>0</Valid_Configuration>
    <Rx_Ready>1</Rx_Ready>
    <Rx_Valid>0</Rx_Valid>
    <Tx_Ready>1</Tx_Ready>
    <Tx_Valid>0</Tx_Valid>
    <Data_Type>Images</Data_Type>
    <Signal_Type>Images</Signal_Type>
    <Object_To_String>img[C1-Multi_3DHyper.tif (512x512x1x10x3)]</Object_To_String>
```

```
  <Root_Folder>null</Root_Folder>
 </Pin>
 <Pin>
  <Name>Pin_4</Name>
  <Port_Name>Port_3</Port_Name>
  <Block_Name>Block_1</Block_Name>
  <Number_Links>0</Number_Links>
  <IO_Type>Output</IO_Type>
  <Valid_Structure>0</Valid_Structure>
  <Valid_Configuration>0</Valid_Configuration>
  <Rx_Ready>1</Rx_Ready>
  <Rx_Valid>0</Rx_Valid>
  <Tx_Ready>1</Tx_Ready>
  <Tx_Valid>0</Tx_Valid>
  <Data_Type>Images</Data_Type>
  <Signal_Type>Images</Signal_Type>
  <Object_To_String>img[C2-Multi_3DHyper.tif (512x512x1x10x3)]</Object_To_String>
  <Root_Folder>null</Root_Folder>
 </Pin>
 <Pin>
  <Name>Pin_5</Name>
  <Port_Name>Port_3</Port_Name>
  <Block_Name>Block_1</Block_Name>
  <Number_Links>0</Number_Links>
  <IO_Type>Output</IO_Type>
  <Valid_Structure>0</Valid_Structure>
  <Valid_Configuration>0</Valid_Configuration>
  <Rx_Ready>1</Rx_Ready>
  <Rx_Valid>0</Rx_Valid>
  <Tx_Ready>1</Tx_Ready>
  <Tx_Valid>0</Tx_Valid>
  <Data_Type>Images</Data_Type>
  <Signal_Type>Images</Signal_Type>
  <Object_To_String>img[C3-Multi_3DHyper.tif (512x512x1x10x3)]</Object_To_String>
  <Root_Folder>null</Root_Folder>
 </Pin>
 <Pin>
  <Name>Pin_6</Name>
  <Port_Name>Port_3</Port_Name>
  <Block_Name>Block_1</Block_Name>
  <Number_Links>0</Number_Links>
  <IO_Type>Output</IO_Type>
  <Valid_Structure>0</Valid_Structure>
  <Valid_Configuration>0</Valid_Configuration>
  <Rx_Ready>1</Rx_Ready>
  <Rx_Valid>0</Rx_Valid>
```

```
      <Tx_Ready>1</Tx_Ready>
      <Tx_Valid>0</Tx_Valid>
      <Data_Type>Images</Data_Type>
      <Signal_Type>Images</Signal_Type>
      <Object_To_String>img[C3-Multi_3DHyper.tif (512x512x1x10x3)]</Object_To_String>
      <Root_Folder>null</Root_Folder>
     </Pin>
   </Port>
  </Block>
  <Link>
   <Name>Link_0</Name>
   <Source_Pin_Name>Pin_1</Source_Pin_Name>
   <Destination_Pin_Name>Pin_2</Destination_Pin_Name>
  </Link>
</BlockDiagram>
```