ITKBOARD: A VISUAL DATAFLOW LANGUAGE FOR BIOMEDICAL IMAGE PROCESSING

Hoang D. K. Le, Rongxin Li, Sébastien Ourselin
BioMedIA, Autonomous Systems Laboratory

John M. Potter
Programming Languages & Compilers Group
School of Computer Science & Engineering
University of New South Wales, Sydney

Keywords: Dataflow programming, visual programming languages, image processing software, ITK, ITKBoard.

Abstract: Experimenters in biomedical image processing rely on software libraries to provide a large number of standard filtering and image handling algorithms. The Insight Toolkit (ITK) is an open-source library that provides a complete framework for a range of image processing tasks, and is specifically aimed at segmentation and registration tasks for both two and three dimensional images.
This paper describes a visual dataflow language, ITKBoard, designed to simplify building, and more significantly, experimenting with ITK applications. The ease with which image processing experiments can be interactively modified and controlled is an important aspect of the design. The experimenter can focus on the image processing task at hand, rather than worry about the underlying software. ITKBoard incorporates composite and parameterised components, and control constructs, and relies on a novel hybrid dataflow model, combining aspects of both demand and data-driven execution.

1 INTRODUCTION

Segmentation and registration are two common tasks that are conducted in processing biomedical images, such as those produced by Computed Tomography Imaging (CT scanners) and Magnetic Resonance Imaging (MRI scanners). *Segmentation* involves identifying and classifying features of interest within an image, and *registration* involves the alignment of corresponding parts of different images or underlying grids and meshes. A biomedical application being explored in our BioMedIA laboratory at CSIRO performs non-invasive modelling of the internal 3D structure of an artery. Potentially this can be used to customise the design of a good-fitting stent as used in angioplasty procedures. Segmentation is used to identify the arterial region of interest, and registration used to align the 3D model of the stent.

Researchers working in this area typically experiment with different image filtering algorithms, and modify parameters of the algorithms in attempting to achieve good images for diagnostic or other purposes. Their concern is with the kind of algorithm and the tuning process involved, in order to achieve accurate reconstructions from the images. They do not, in general, want to be concerned with the development of the underlying software.

The *Insight Toolkit* (ITK) is an open-source software toolkit for performing registration and segmentation. ITK is based on a demand-driven dataflow architecture, and is implemented in C++ with a heavy reliance on C++ templates to achieve efficient generic components. Only competent C++ developers can build image processing applications using ITK in the raw. Although the toolkit does allow Python, Java or Tcl scripting of ITK components, the level of programming expertise and time required is still too high for convenient experimentation with ITK. In particular, experimenters working with scripted models, still have a cognitive mismatch between the image processing problems they want to solve, and the task of writing scripts to run different experiments.

Our aim with *ITKBoard* is to provide a simpler platform for biomedical researchers to experiment on, leveraging the underlying strengths of ITK, but requiring little or no programming knowledge to use. More specifically, we aim to overcome some existing limitations such as a lack of efficiency in dataflow

D. K. Le H., Li R., Ourselin S. and M. Potter J. (2007).
ITKBOARD: A VISUAL DATAFLOW LANGUAGE FOR BIOMEDICAL IMAGE PROCESSING.
In *Proceedings of the Second International Conference on Software and Data Technologies - PL/DPS/KE/WsMUSE*, pages 13-21
DOI: 10.5220/0001339800130021
Copyright © SciTePress

model construction, difficulty in customising filter's properties for running different experiments, and lack of high-level constructs for controlling execution.

To this end, we have designed a visual dataflow language which provides extra features above and beyond ITK. The *novel contributions* of ITKBoard are: *visual construction* of ITK applications, by graphical manipulation of filtering models as reported elsewhere (Le et al., 2005)); a *hybrid dataflow* model, combining both *demand-driven* execution for images, and *data-driven* execution for filter parameters; explicit *visual parameterisation* of filters, with graphical input/output parameter dependencies between filters; *visual composition* of filters that can be saved and redeployed in other applications; explicit *visual control flow* with selection and repetition constructs; and explicit construction of expressions combining parameters through the visual interface.

The combination of features in ITKBoard is unique, and designed to suit experimentation in biomedical image processing. In particular, our hybrid dataflow model, incorporating both data-driven and demand-driven computation, is novel. The control constructs in our language are also interesting, being tailored specifically to cope with the underlying ITK execution model, combined with our hybrid model.

The paper is organised as follows. Section 2 provides a summary of the software architecture of ITK. In particular we discuss how the uniformity in the ITK design allows dataflow models to be "wired up", and filters to be executed on demand. The user-based perspective of ITKBoard, presented in Section 3, gives an indication of the ease-of-use that we achieve with the interactive visual layer that we place on top of the ITK model. In Section 4 we focus on the hybrid dataflow model that we employ in ITKBoard. Here we extend the demand-driven dataflow model of ITK, in which image data is cached at filter outputs, with a data-driven model for handling filter parameters. This adds a uniformity to handling parameters which is not directly provided in the ITK model. Further details of the ITKBoard architecture are outlined in Section 5: we consider wrapper and plug-in mechanisms for ITKBoard, composite filters, a simple way of expressing combinations of parameters, and explicit visual control flow constructs for selection and repetition. In Section 6 we discuss the contribution of our work, and compare it with other related work, before a brief conclusion in Section 7 where we also point to possible future work.

2 ITK: THE INSIGHT TOOLKIT

We provide a brief summary of the features of ITK which are relevant for the design of ITKBoard. There is a comprehensive software guide (Ibáñez et al., 2005) which should be consulted for further information.

ITK is based on a simple demand-driven dataflow model (Johnston et al., 2004) for image processing. In such a model, the key idea is that computation is described within process elements, or *filters*, which transform input data into output data. Data processing *pipelines* form a directed graph of such elements in which the output data of one filter becomes the input for another. In a demand-driven model, computation is performed on request for output data. By caching output data it is possible to avoid recomputation when there have been no changes to the inputs. Image processing is computationally intensive, and so redundant computations are best avoided. ITK has therefore adopted this lazy, memo-ised evaluation strategy as its default computational mechanism.

ITK is implemented in C++ and adopts standardised interfaces for the constituent elements of the dataflow model. In particular it provides a generic interface for *process objects* (filters). In ITK images and geometric meshes are the only types of data transformed by filters. They are modelled as generic data objects, which can be specialised for two and three dimensional image processing. Typically output data objects for a filter are cached, and only updated as necessary.

When inputs are changed, downstream data becomes out-of-date. Subsequent requests will cause re-computation of only those parts of the model which are out-of-date. As well as tracking the validity of the cached output images, ITK provides a general event notification mechanism, based on the *Observer* design pattern (Gamma et al., 1995) for allowing arbitrary customisation of behaviours that depend on the state of computation. This mechanism makes it feasible for us to trap particular events in the standard ITK execution cycle, and intersperse the standard behaviours with further updating as required by our ITKBoard model.

ITK uses data other than the objects (image data) that are transformed by filters. This other data is constrained to be local to a particular filter. Such data is usually used to parameterise the behaviour of the filters, or in some cases, provide output information about a computation which is distinct from the image transformation typical of most filters. Such parametric data are treated as properties of a filter, and there

is a standard convention in ITK for providing access to them with get and set methods. There is however, no standard mechanism in ITK to model dependencies between the parameters of different filters in ITK. This is one of our contributions with ITKBoard.

To build an ITK application, the basic approach requires all elements of the model to be instantiated as objects from the C++ classes provided by ITK. The toolkit does encourage a particular idiomatic style of programming for creating filter instances and constructing filter pipelines by hard-wiring the input-output connections. Nevertheless, programming in any form in C++ is not for the faint-hearted.

Consequently, to escape from the complexities of C++, there is a wrapper mechanism which allows ITK filters to be accessed from simpler, interpreted languages. Currently those supported are Java, Python and Tcl. Image processing applications can be written in any of these languages, using standard ITK filters as the primitive components. However, even with such high-level languages, the text-based programming task still slows down the experimenter who simply wants to investigate the effect of different image processing algorithms on a given image set.

3 A USER'S PERSPECTIVE OF ITKBOARD

ITKBoard is a system that implements a visual dataflow language, providing a simple means of interactively building a dataflow model for two- and three-dimensional image processing tasks. Unlike the underlying ITK system, outlined in Section 2, ITKBoard requires no knowledge of programming. It provides a much less intimidating approach to the image processing tasks that ITK is designed to support. To execute an image processing task simply involves clicking on any image viewer element in the model. By offering a single user interface for both model construction and execution, ITKBoard encourages experimenters to try out different filtering algorithms and different parameter settings.

We summarise the key features that are evident from the screenshot displayed in Figure 1. This shows a small example of an image processing task to illustrate the key concepts behind our design of ITKBoard. The simplicity and ease-of-use of the interface and the intuitive nature of the way in which tasks are modelled should be evident from the figure. The panel on the left shows the palette of available elements, which are typically wrapped versions of ITK process objects. This provides the interface with the library of components that ITK provides—no other exposure of ITK functionality is required in ITKBoard.

The main panel displays a complete filtering application mid-way through execution, as suggested by the execution progress bar. To construct such an application, the user selects elements from the left panel, placing them in the application panel, and connecting outputs to inputs as required. We visually distinguish between image data, the thick dataflow edges, and parametric data, the thin edges. Image inputs are provided on the left of filter elements, and outputs on the right. Parameter inputs are via pins at the top of filters, outputs at the bottom. ITKBoard uses colour coding of image dataflows to indicate when image data is up-to-date. In the colour version of Figure 1, red image dataflows indicate that the data is out-of-date, yellow flows indicate that data is being updated, and green ones indicate up-to-date data.

Viewer elements act as probes, and provide the mechanism for execution. The user may request any particular view to be updated. This demand is propagated upstream, just as in ITK, until the cached image data is found to be up-to-date. Viewers cache their image data. This allows experimenters to attach multiple viewers to a given filter output, and update them independently. This is an easy-to-use device for simultaneously observing images produced before and after changes to the upstream model which is particularly helpful for experimenting with filtering models. Figure 2 illustrates the display for a viewer, in this case one that compares two-dimensional cross sections of three-dimensional images.

Parameters of a filter are interactively accessible via property tables, which can be displayed by clicking on the filter icon. Input parameters can be set in one of three ways: at initialisation, with some default value, by explicit user interaction, or by being configured as being dependent on some other parameter value. For example, in Figure 1, inputs of the *Binary Threshold* depend on output parameters of the upstream *Statistics* filter. We will discuss how parameter data propagation works in Section 4.

Our example illustrates a simple *if-then-else* control construct. To update its output image, it chooses between its two image data inputs, according to the current value of its selection condition, which is simply some expression computed from its parameters. One key design consideration is apparent here: in order to prevent image update requests being propagated upstream via all inputs, the condition is evaluated based on current parameter settings. Otherwise, propagating the request on all inputs would typically involve expensive image processing computations, many of which may simply be discarded. This will be discussed further in the following sec-

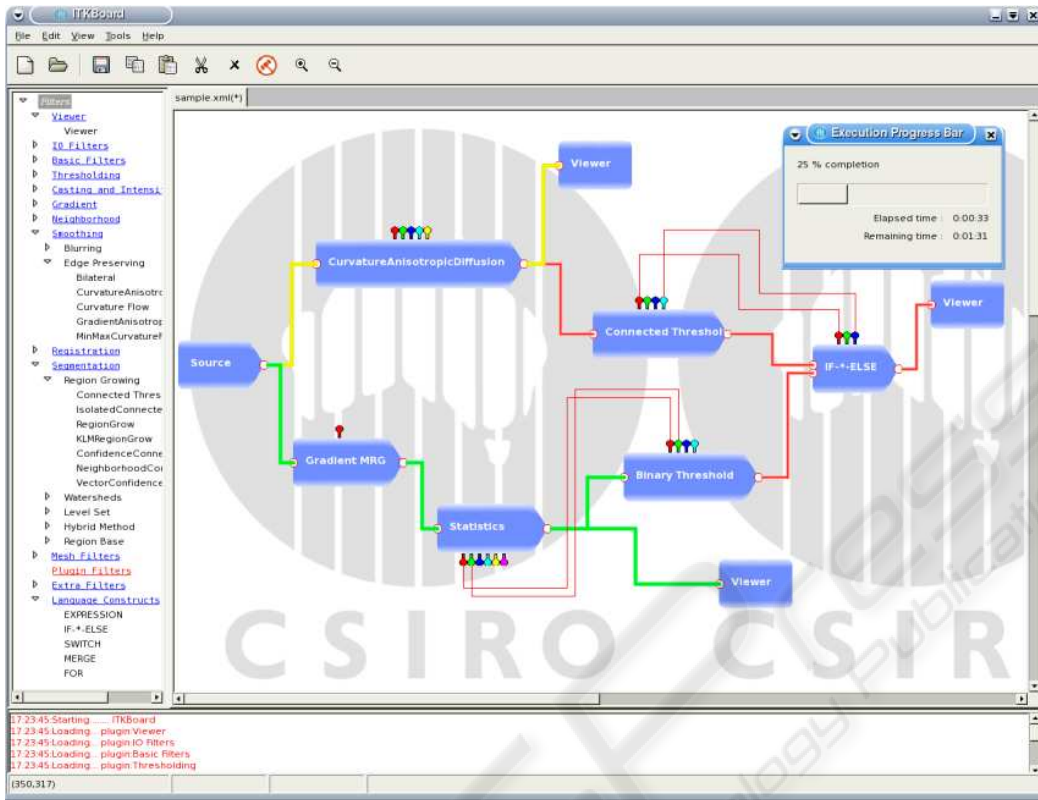

Figure 1: Screenshot of ITKBoard.

tions. The bottom panel of Figure 1 simply displays a trace of underlying configuration actions for debugging purposes during development of ITKBoard.

Figure 2: Comparing images with a Viewer.

4 THE HYBRID DATAFLOW MODEL

In ITK there is a clear distinction between image data which participates in demand-driven dataflow computations, and parametric data used to configure filters. Image data is large and must be managed with care, whereas parameters are usually simple scalars or vectors that consume negligible memory. Sometimes in image processing tasks, parameters do need to be logically shared between different filters. With ITK, any such sharing must be configured explicitly by the user when they set the parameters for individual filters. For ITKBoard we introduced a design requirement that sharing of such data between filters must be made explicit in the model that a user builds.

A couple of design alternatives presented themselves to us. First, instead of treating parameters as local to individual filters, we could simply take them to be global variables for the whole model. This solution is certainly implementable, but requires a global namespace for the parameters, and binding of the global parameter names to the local names used within each of the individual filters. On the downside, this approach requires the user to invent a global name and keep track of its local variants, in order to understand what parameters are shared where. With the ability to form composite models, we must also deal with the added complexity of nested namespaces. Furthermore, we want parameters of some filters to be able to determine those of other filters. This would require the user to distinguish between those parameters which had to be set externally by the user, and those which would be the responsibility of a filter to set.

Instead of using global variables for handling

parametric data, we have opted for a dataflow solution. In our ITKBoard model, we deem this appropriate because we are already using a dataflow model for image data, and because it avoids some of the problems mentioned above in treating parameters as global variables. First, with a visual dataflow language, there is no need to provide global names for the dataflow parameters; second, the dataflow connections make apparent any dependencies between parameters, and who is responsible for setting the data values, by distinguishing between input and output parameters—more on this soon; third, with a composite model, it is a relatively simple matter to choose which internal parameters to expose at the composite level, with external inputs mapping to internal inputs, and internal outputs mapping to external outputs.

Perhaps the most interesting design choice in ITKBoard has been to opt for a *data-driven propagation scheme for data parameters*. Given that ITK already supports a demand-driven model of computation for image data, why not instead just go with the flow, and make the update of parameters demand-driven as well? There are three main reasons for making parameter updates data-driven.

The first reason is implementation biased—it is simpler to extend ITK with a data-driven propagation scheme for parameter updates. Whenever a parameter is updated, we simply ensure that any downstream dependants are immediately notified of the change, and correspondingly updated. This mechanism works for output parameters of a filter as well. Normally when an ITK filter completes its (demand-driven) computation of its output image, the downstream filters are notified of completion, so they may proceed with any pending computation. In ITKBoard, our implementation simply catches this notification event, and updates downstream parameters with the new output parameter values which can be gotten from the underlying ITK filter, before proceeding with the interrupted demand-driven computation downstream. If, alternatively, we implemented demand-driven parameter update, we would have to track when output parameters are out-of-date, and this would presumably be delegated to the ITK mechanism which is based on checking whether the output images are out-of-date; whatever the implementation trick used, the demand-driven approach implies that requests for parameter updates will trigger (potentially expensive) image processing computations. With the data-driven approach, (cheap) parameter updates are propagated immediately, and never trigger image processing without a separate request for an up-to-date image.

The second reason for adopting the data-driven scheme for parameter propagation is more user focused. Our scheme implies that all image processing computations are driven by requests for up-to-date images, and nothing else. These requests are propagated upstream, and the state of computation is solely determined by the current validity of image data and the current thread of incomplete requests. In particular, a dependency of an input parameter on an upstream filter's output parameter will not trigger computation for the upstream filter, even if that filter is not up-to-date.

The third reason for not applying the demand-driven scheme to parametric data is to allow a richer class of models, while still avoiding recursion caused by cycles between image data and parameter dependencies. In a standard demand-driven model, downstream output parameters cannot connect to upstream filters without causing a feedback loop. By separating the image demands from the data-driven parameter updating mechanism, ITKBoard can handle such dependencies. So, ITKBoard allows cyclic dependencies that involve at least one image and one parameter data dependency. ITKBoard does not allow cycles of image dataflow, or of parameter dataflow; only mixed cycles are allowed. ITKBoard provides built-in support for iterative filters, so image feedback is not needed for expressing repetitive processing.

Because the user interface gives visual cues about which data is not up-to-date, and where parameter dependencies lie, it is a simple matter for a user to explicitly request that an output parameter for a particular filter be brought up-to-date by requesting an updated view of the output image for that filter. The rationale for this is to give the user better control over which parts of the model are actually re-computed after changes are made, thereby avoiding potentially expensive but unnecessary image processing. For biomedical experimenters, we decided this finer granularity of control over where image processing occurs was a worthwhile feature.

Input parameters can be set in a number of ways. Unconnected parameters have a default value. Input parameter pins are positioned along the top of the filter icon, and outputs at the bottom. Connections can be made from either input or output pins to other input pins. When a connection is made between two input parameters, the downstream input parameter is overridden by the value of the upstream input. For all parameter connections, whenever the upstream value is updated, the connected downstream value is correspondingly updated. Users may directly override the current value of an input parameter associated with a filter; in this case, any downstream dependants will be updated, but upstream parameters will retain their old value—again, this gives experimenters some ex-

tra flexibility. They have the opportunity to test local parameter changes in a model, without restructuring the dependencies in the model. This is similar to the way some program debuggers allow the values of variables to be set externally. Note that the value of a user-modified input parameter will be retained until there is some other upstream variation causing the parameter to be modified via some inward parameter dataflow, if such a connection exists.

Our design choice in opting for a hybrid dataflow model does have an impact on the behaviour of control constructs as described later in Section 5.

5 MORE DETAILS OF ITKBOARD

We detail some of the more useful features of ITKBoard: auto-wrappers for ITK filters and a plug-in mechanism, ITKBoard's take on composite filters, support for expressing parameter-based computations, and finally control constructs.

5.1 Wrappers and Plug-ins for ITKBoard

The main goal of ITKBoard has been to present an easy-to-use interface for experimenters who want to use ITK without worrying about programming details. To this end it is critical that ITKBoard can easily access all of the ITK infrastructure and that new pre-compiled components can be easily added to ITKBoard.

The first mechanism is an auto-wrapper for ITK filters so that ITKBoard is easily able to leverage all of the filtering functionality provided by ITK, or by any other C++ code implementing similar interfaces. Our auto-wrapper parses the C++ code for a filter, and generates C++ code that wraps the filter so that it can be used within ITKBoard. In particular, the auto-wrapping process is able to extract input and output parameters for an ITK filter, assuming the convention that input parameters are those defined with get and set methods (or macros), and output parameters those just with get methods. The auto-wrapper allows a user to intervene to adapt the auto-wrapper's translation as required.

The second mechanism provides support for plug-ins. This allows us to include newly developed filters into the ITKBoard system without recompiling the source code for the system. This is important for effective sharing and distribution of extensions. We rely on a shared library format (.so or .dll) to achieve this. Every shared library can provide a collection of filters, and must provide a creation routine for instantiating the filters implemented by the library.

Details of both the auto-wrapper and plug-in mechanisms for ITKBoard have been presented elsewhere (Le et al., 2005).

5.2 Composite Filters

Although ITK itself can support the construction of composite filters made up of other filters, we provide a separate mechanism in ITKBoard. The reason for doing this is that we wish to distinguish between primitive filters (typically ITK filters), and those which can be composed of sub-filters.

So we simply provide an ITKBoard implementation of the *Composite* design pattern (Gamma et al., 1995). The *Component* interface is the standard ITKBoard abstraction for representing a filter. *Leaf* components are ITK filters with their standard ITKBoard wrapper. What perhaps is interesting here, is that the composite structure is not hard-coded, as it is in ITK. The definition of the composite is simply an XML-based description of the structure of the composite filter which describes the individual components of the composite, together with their data dependencies (the "wiring"). When a composite filter is instantiated by the user, the actual internal structure of the composite is dynamically configured by interpreting the XML description.

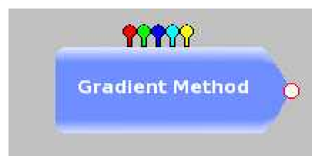

Figure 3: A Composite Filter: collapsed form.

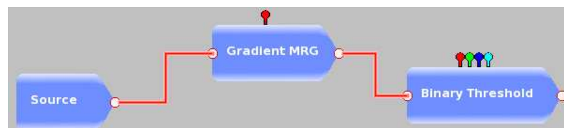

Figure 4: A Composite Filter: expanded form.

At the user interface level, composites can either be collapsed, appearing as a single filter icon, or expanded, showing the internal structure of the composite. For example Figure 3 displays a composite filter in collapsed form, which can be used just like any other filter. The expanded form of the same composite filter, Figure 4 shows both the internal data dependencies and the distribution of parameters amongst the internal components.

5.3 Parameter Expressions

Most filter parameters are simple scalar or vector values used to customise filters, such as the threshold level for a simple filter. In some cases we may wish to combine parameters in simple ways. To that end we define a simple expression interpreter for defining arithmetic and comparison operations on data values. These expressions can be entered into the property tables that define the parameters for a component. Although a more elaborate mechanism could be built in, we think that a simple expression interpreter is likely to suffice for most kinds of applications; anything more complex can easily be implemented

5.4 Control Constructs

Structured control flow has three aspects: sequencing, choice and repetition.The dataflow model naturally supports sequencing of computations through its directed graph of data dependencies. Given that filters may have multiple inputs, it is apparent that we can encode choice and repetition within the implementation of a filter. In fact, some of the standard ITK components have optimising filters which may already implement repeated filtering behaviour until some criterion is met. However, our goal with ITK-Board is to provide a visual language to allow users to specify selective and repetitive behaviour.

Selection. Conditional selection of inputs is achieved with the *If-*-Else* construct, as depicted in Figure 5. Each image data input is guarded by a

Figure 5: Conditional construct.

boolean expression. The first of these to be true, starting from the top, is selected for input. Given inputs In_i for $i = 0 \ldots n$, with boolean guards $Cond_i$ for $i = 0 \ldots n-1$, we can define the output Out by:

$$\begin{aligned} Out &= In_0, && \text{if} Cond_0 \\ &= \ldots \\ &= In_{n-1}, && \text{if} Cond_{n-1} \\ &= In_n, && \text{otherwise} \end{aligned}$$

The guard expressions are defined in terms of the input parameters, as properties of the *If-*-Else* construct. With the data-driven policy for parameters, it is clear that the guards do not directly depend on the input image data. So, with demand-driven computation, we find that each request for output is propagated to just one input, at most. In other words, the selection of the input dataflow In_i only depends on i, which, in turn only depends on the input parameters.

This deterministic behaviour for conditional input selection, which may seem somewhat unorthodox on first sight, is in fact, one of the main reasons for opting for the hybrid dataflow model, as already discussed in Section 4. If instead, we had opted for a demand-driven model for input parameters, then this conditional expression could generate multiple upstream requests until a true condition was found. In Figure 1, for example, the parameters for the conditional block are only dependent on the input parameters of the preceding *Connected Threshold* filter. So, while those parameters remain unaltered, the *If-*-Else* will always select the same input when the right-most viewer is updated—the selection does not depend on the input image data. If we want downstream choices to depend on upstream images, then we must make it explicit in the model, by arranging for some filter to process the upstream image, generating output parameters which can then be used as inputs to an *If-*-Else*. When such a dependency is modelled, the actual behaviour for the conditional selection can depend on where the user makes update requests. If the user *guides* the computation downstream, by ensuring that all upstream image data is up-to-date, then any conditional selections made downstream will be up-to-date. However, if the user only requests updates from the downstream side, it is possible that, even with repeated requests, that some of the upstream data remains out-of-date.

Although it may seem undesirable to allow different results to be computed, depending on the order of update requests, we claim this to be a benefit of our hybrid model. As mentioned in Section 4, the user is always given visual cues to indicate which image data is not up-to-date. This gives the experimenter control over which parts of the model are brought up-to-date, and which parts may be ignored while attention is focused on another part of the model.

Repetition. In some dataflow languages that focus on stream-based dataflow, iteration is often presented in terms of a feedback loop with a delay element. Although our implementation is indeed based on this idea, we have chosen to keep this as implementation detail which need not be exposed to the user. Instead we provide a simple way of wrapping an existing filter in a loop construct, with explicit loop control managed as a property of the loop construct.

In Figure 6 we illustrate the manner in which a *while loop* can be expressed. Observe how the ini-

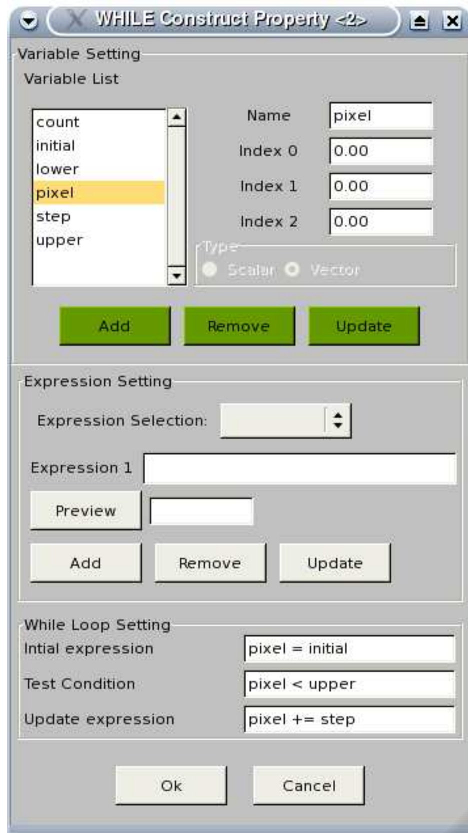

Figure 6: While Loop construct.

tial, test and update expressions are written in terms of properties of the loop construct. Although logically redundant, we also provide a counting *for loop*.

6 RELATED WORK

A recent survey (Johnston et al., 2004) provides a good account of dataflow programming, including dataflow visual programming languages (DFVPLs). The current state of play for DFVPLs with iteration constructs is reviewed by (M. Mosconi, 2000).

Prograph (Cox et al., 1989) is a general purpose DFVPL that uses iconic symbols to represent actions to be taken on data. Its general-purpose features are not ideal for supporting computationally intensive domains such as signal processing and image analysis. However, control flow in Prograph is implicitly defined, forcing users to implement their own control constructs rather than directly use available constructs.

The Laboratory Virtual Instrumentation Engineering Workbench(LabVIEW) (LabVIEW,) is a platform and development environment for a VPL from National Instruments. LabVIEW aims to provide design tests, measurement and control systems for research and industry. It is not geared towards experimental image processing like ITKBoard is. Two iterative forms are supported in LabVIEW namely the for-loop and the while-loop, with a square-shaped block to represent the loop's body.

Cantata (Young et al., 1995) is a language originally designed for image processing within the Khoros system (Konstantinides and Rasure, 1994), a visual programming environment for image processing. It is also useful for signal processing and control systems. In Cantata, icons (called glyphs) represent programs in Khoros system. Unlike ITKBoard, Cantata provides a coarse-grained computation model more suited to distributed computation, where each glyph corresponds to an entire process rather than to a process component. There are two iterative forms provided in Cantata namely the count-loop and the while-loop, similar to our approach. VisiQuest (VisiQuest,) is a commercial scientific data and image analysis application that builds on the foundation laid by Khoros and Cantata.

VIPERS (Bernini and Mosconi, 1994) is a DFVPL based on the standard scripting language Tcl, which is used to define the elementary functional blocks (the nodes of the data-flow graph) which are similar to Khoros programs in Cantata. Interestingly, each block corresponds to a Tcl command so VIPERS is less coarse-grained than Cantata. VIPERS programs are graphs that have data tokens travelling along arcs between nodes (modules) which transform the data tokens themselves. VIPERS has some basic representations to allow node connections, ports to link blocks into a network. However, the use of enabling signals in each block introduces a new boolean data-flow channel on top of the main input-output data-flow. VIPERS relies on enabling signals to support language control constructs. See (M. Mosconi, 2000) for examples and screen-shots.

MeVisLab (Rexilius et al., 2005) is a development environment for medical image processing and visualisation. MeVisLab provides for integration and testing of new algorithms and the development of application prototypes that can be used in clinical environments. Recently, ITK algorithms have been integrated into MeVisLab as an "add-on" module. However, in ITKBoard, ITK provides the core image processing functionality; with its extensive user base, ITK library code reliable and well validated. Moreover, ITKBoard provides a full DFVPL with extra expressiveness in comparison to MeVisLab.

7 CONCLUSION

We have presented ITKBoard, a rich and expressive visual dataflow language tailored towards biomedical image processing experimentation. By building on top of ITK, we have been able to leverage the strengths of ITK's architecture and libraries. At the same time, we provide an interactive development environment in which experimenters can develop new image processing applications with little or no concern for programming issues. Furthermore, in one and the same environment, they can interactively execute all or part of their model, and investigate the effect of changing model structure and parameterisation in a visually intuitive manner.

The hybrid dataflow model appears to be unique amongst dataflow languages, and matches well with the division of data into two kinds in the underlying ITK framework, and with the experimental image processing that ITKBoard has been designed to support. Currently we have a complete working implementation of the features of ITKBoard as discussed in this paper. Further work is needed to develop ITKBoard to support streaming of image data and multithreaded computation that ITK already goes some way to supporting.

REFERENCES

Bernini, M. and Mosconi, M. (1994). Vipers: a data flow visual programming environment based on the tcl language. In *AVI '94: Proceedings of the workshop on Advanced visual interfaces*, pages 243–245, New York, NY, USA. ACM Press.

Cox, P. T., Giles, F. R., and Pietrzykowski, T. (1989). Prograph: A step towards liberating programming from textual conditioning. In *Proceedings of the IEEE Workshop on Visual LanguagesVL'89*, pages 150–156, Rome, Italy.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, MA,USA.

Ibáñez, L., Schroeder, W., Ng, L., Cates, J., and the Insight Software Consortium (2005). *The ITK Software Guide*, 2nd edition. http://www.itk.org.

Johnston, W. M., Hanna, J. R. P., and Millar, R. (2004). Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34.

Konstantinides, K. and Rasure, J. R. (1994). The Khoros software development environment for image and signal processing. *IEEE Transactions on Image Processing*, 3(3):243–252.

LabVIEW. National Instruments Corporation. LabVIEW. User Manual, 2003.

Le, H. D. K., Li, R., and Ourselin, S. (2005). Towards a visual programming environment based on itk for medical image analysis. In *Digital Image Computing: Techniques and Applications (DICTA'05)*, page 80. IEEE Computer Society.

M. Mosconi, M. P. (2000). Iteration constructs in data-flow visual programming languages. *Computer Langages*, 22:67–104.

Rexilius, J., Spindler, W., Jomier, J., Link, F., and Peitgen, H. (2005). Efficient algorithm evaluation and rapid prototyping of clinical applications using itk. In *Proceedings of RSNA2005*, Chicago.

VisiQuest. Accomplish complex image and data analysis tasks with an advanced, VisiQuest Visual Proramming Guide,2006.

Young, M., Argiro, D., and Kubica, S. (1995). Cantata: visual programming environment for the khoros system. *SIGGRAPH Computer Graphics*, 29(2):22–24.