Towards a Visual Programming Environment Based on ITK for Medical Image Analysis

Hoang D. K. Le, Rongxin Li and Sébastien Ourselin
BioMedIA Lab, Autonomous System Laboratory, CSIRO ICT Centre
Locked Bag 17, North Ryde, NSW 1670, Australia.
http://www.ict.csiro.au/biomedia
{hoang.le, ron.li, sebastien.ourselin}@csiro.au

Abstract

Medical image analysis experiments usually require a certain level of programming skills and are often time-consuming. Visual programming can address these issues by providing a high-level user interface to visually access and combine underlying medical image analysis functionalities. This paper describes the ITKBoard, an extension of the Insight Toolkit for Registration and Segmentation (ITK). Unlike traditional approaches, ITKBoard, a visual tool for experimentation and prototyping, allows rapid visual construction of filter pipelines, via a graphical user interface. An overview of the main functionalities and the system design is presented in this paper. The approaches that we have developed to resolving critical issues such as pipeline handling, automatic object wrapping and plugin mechanism are discussed in detail. This is a first step towards a visual programming environment designed for medical image analysis, in which the specific needs in terms of visual data-flow and control-flow programming will be progressively addressed. It is envisaged that when the software is released in the near future it will be significantly beneficial to the medical image analysis community.

1. Introduction

The Insight Toolkit for Registration and Segmentation (ITK) [5] is an extensive open-source library of image analysis algorithms. Image analysis experiments (even using ITK) are often time-consuming, especially when the program needs to be written from scratch. ITKBoard, a visual programming tool for performing rapid experimentation and prototyping, provides a bridge for users to access ITK functionalities. It also works as a visual debugger for algorithm developers. In this section, visual programming and ITK will be discussed in more detail.

1.1. Visual Programming

Visual Programming (VP) uses a graphical representation to display and create computer programs [9]. VP is normally associated with visual programming languages to support logical constructs [2]. Using visual representations in the programming process, VP can bring computing facilities to people who do not have intensive computer training [12]. In this paper, VP is applied to the image processing domain. Related previous efforts include Khoros [7] and SCIRun [11]. A filter can be represented by an icon that is associated with a label as the filter's name. One icon can be joined with another icon to form a connection which is illustrated by a connector. Notice that, each icon may have one or more ports which stand for input or output ports of the associated filter. Many icons can be joined together to form flow-charts which represent data-flows or conceptual models.

1.2. ITK

ITK is implemented in the C++ language with the generic programming approach, i.e. it uses templated code. ITK is cross-platform and built by using the CMake ¹ build environment to manage the compilation process in a platform-independent fashion. In addition, an automatic wrapping process known as CableSwig ² generates interfaces for ITK between C++ and other programming languages such as Tcl, Java and Python. This allows developers to create software on top of ITK using different programming languages.

¹CMake is an extensible, open-source system that manages the build process in an operating system and compiler independent manner [3].
²CableSwig is a software tool that is used to create interfaces (i.e. "wrappers") to interpreted languages such as Tcl and Python. It was created to generate wrappers of C++ structures for ITK.
1.3. Outline

This paper describes the integration of the ITK toolkit into ITKBoard. Firstly, an overview of ITKBoard is provided in Section 2. In Section 3, three critical issues will be discussed in detail in relation to pipeline handling, automatic wrapping and plugin mechanisms. System design will be given in Section 4, followed by some experiments and results in Section 5. Finally, some discussion and conclusions are presented in Section 6.

2. Overview of ITKBoard

2.1. Rationale

ITK provides a state-of-the-art software suite for performing segmentation and registration. However, ITK is a library rather than an end-user application. This makes it difficult to access ITK functionalities for those who are less experienced in programming (e.g., radiologists). Although ITK supports pipeline designs for image processing, it is time-consuming to code and difficult to visualise the pipeline. ITKBoard was introduced to address these issues. ITKBoard offers intuitive and efficient prototyping of filter pipelines.

2.2. Goals

The overall goal is to provide a visual tool to help in the development, deployment, and refinement of segmentation and registration algorithms to benefit the image processing community. In particular, the tool will enable the construction of filter pipelines visually via a graphical user interface (GUI) and evaluation of the output through the designed pipeline. It also provides the ability for functionality extension such as the mechanism to import user-defined algorithms without requiring any GUI programming skills. It also enables access to and combination of ITK functionalities with less effort for enhancing the illustration of conceptual models and visualisation of volume data.

2.3. Features

Currently ITKBoard is an in-house development for research purposes. It provides a number of features to support data-flow image processing and data visualisation:

- Design or construction of multiple filter pipelines via GUI;
- Integration of ITK filters into modules for performing image processing;
- Output visualisation on the fly from each single filter on the pipeline;
- Setting of filter parameters via GUI;
- Code Generation to build stand-alone applications for each filter design;
- All system functionalities with handy shortcuts are supported as a powerful editor for filter pipeline design;
- Multi-threading support;
- Execution progress visualisation including overall progress status and filter pipeline state;
- Automatic generation of wrappers for ITK-style filters;

3 Critical Issues

3.1 Filter Pipeline

Developing a visual tool for designing filter pipelines on top of the ITK library is a challenging task. Some of the issues and challenges are as follows:

- Once the filter template is instantiated, its methods can only accept the declared argument types [4]. For example, a filter initiated with unsigned short input image can only accept an unsigned short image via the SetInput method. A filter initiated with float output image can not directly connect with the filter that has been initiated unsigned short input image.
- From the software engineering perspective, the Source filter, the Output filter and normal filters cannot share the same function prototype. A source filter only requires the filename of the image source and does not require the SetInput method while Output filter requires a filename for writing its data and the SetInput method has to be available. In contrast, normal filters need both the SetInput and GetOutput API methods.
- The system has to convert human design that simulates a conceptual model into a logical connection in which each connection is interpreted equivalent to C++ code.
- Unlike having a programmed internal pipeline for a particular experiment, the output type of the upstream filter and input type of the downstream filter in the pipeline design are not known in advance. In designing the filter pipeline, the input and output for each connection have to be compatible. If there is any potential conflict, the compiler may not compile the source code or the system may crash at run-time.

The techniques that have been implemented in ITKBoard to solve these issues can be summarised as follows:
• Implementing a top level non-template class that works as an interface for all objects in the pipeline. This provides many advantages such as sharing the most common APIs and provides a means to easily manage objects from a global scope.

• Each filter instance is associated with a unique key that is also the filter’s name. The filter and its name form a pair that is stored in a hashtable that is located in the system’s library.

• Each filter is required to implement a Create member method to clone itself. During the pipeline design, there are many requests for a new filter instance associated with a key. To deal with this, the system’s library makes a query on the hashtable using the key. The filter matching that key in the hashtable will use its Create method to return its clone copy.

• Classifying filters into different lower level groups so that each group has its own interface. Rather than handling individual filters, the system can deal with the group instead. Each group can communicate with each other by using their interfaces.

• Setting the common image data type that flows across the pipeline to a float image data type and using an Adaptor to resolve any conflicts in the pipeline.

• Using internal Adaptors to handle conflicts among different image data types in the pipeline. As the library is subdivided into small subgroups, the adaptation for resolving conflict is needed in where one group uses a different interface to communicate with other groups. For example, in a filter pipeline, if there are both vector and scalar image filters, it always requires an adaptation of vector images to scalar images or vice versa.

These techniques have a great impact on the whole design architecture as shown in Section 4.

3.2 Automatic Wrapper

Wrapping is the process of encapsulating another component and provides means to communicate with that component. The Automatic Wrapper (AW) is a mechanism that we have devised for automatically wrapping pure ITK style filters for use with ITKBoard. Four distinguishing advantages of the AW are described as follows.

The AW is composed of three steps including parsing, data interpretation and wrapper output file generation as shown in Figure 3. The header file normally provides an API function prototype of a filter class in the ITK format syntactically. During the parsing step, the AW traverses through the header and picks out parameters for each filter. The details of how the parser identifies the encapsulation of the parameter property from the header file is described below.

Picking property parameters out of the header is critical to the AW as it has an impact on the internal functions of the filter class that need to be wrapped. AW incorporates all of the following three methods to recognize the property parameters from the header file:

• Identify the property parameter using defined macros from ITK.

• Encapsulating C/C++-language style for method prototype.

• Combining defined macro and C/C++ language style.

In order to identify a property, the parser has to be able to recognize the Get and Set methods that point to the same variable name. With the first method, the Set and Get macro need to refer to the same property. For example, itkGetMacro(OutsideValue, OutputPixelType) and itkSetMacro(OutsideValue, OutputPixelType) produce a OutsideValue property in OutputPixelType type which is a user-defined datatype. By following the first method, a parameter can be picked out efficiently. Many other ITK macros are also managed in the same fashion as long as they conform to the above principle. Notice that, in the first step, classification on ITK macro can be used to get some other information such as filter name, filter type and superclass name.

On the other hand, the second method relies on having method prototypes for Set and Get to the same property. In the same example for the first method, the coexistence of void SetOutsideValue(OutputPixelType) and OutputPixelType GetOutsideValue() results in a property parameter OutsideValue in OutputPixelType type. With the last method, a property parameter can be recognized if Set or Get defined in ITK macros corresponds to Get or Set methods in C/C++ language style.

The following code snippet illustrates the outcome of the above methods. Given the following statements:

\[
\begin{align*}
&\text{virtual void SetLowerThreshold(const InputPixelType threshold);} \\
&\text{virtual InputPixelType GetLowerThreshold() const;} \\
&\text{typedef typename Superclass::TimeStepType TimeStepType;} \\
&\text{itkSetMacro(TimeStep, TimeStepType);} \\
&\text{itkGetMacro(TimeStep, TimeStepType);} \\
\end{align*}
\]

The output header code generated by the AW is:

\[
\begin{align*}
&\text{void SetLowerThreshold(const InputPixelType);} \\
&\text{const InputPixelType GetLowerThreshold()} \\
&\{ \\
&\quad \text{return m_filter— > GetLowerThreshold();} \\
&\};
\end{align*}
\]
void SetTimeStep(TimeStepType);
TimeStepType GetTimeStep()
{
    return m_filter—>GetTimeStep();
};

During the first step, all data from the header file is
then stored for the later steps. When the AW module com-
pltes the parsing of the header file, it proceeds to the next
step which processes the stored information and interprets
the stored data. The AW provides an iterative process for
finding whether the datatype of each parameter is scalar or
vector. Each header may have a number of definition data
types known as typedefs. The process to find these data
types’ dimension is also similar to a limited iterative depth
first search. Figure 1 shows the initial screenshot of the AW
to initiate the first step. Once the header is loaded, prop-
erty parameters are displayed as in Figure 2 before the AW
moves on to Step 2. The flowchart in Figure 3 depicts the
interaction of the first step with the last two steps to produce
wrapped output files. Part of the output header file can be
seen in the code snippet above.

Extensible markup language (XML) is the most com-
mon choice in such a situation as XML provides standards
for describing a certain problem. For example, it is used in
SCIRun. However, this would require knowledge of XML
in addition to predefined rules that the users need to follow
in order to transform their filter class into a required for-
m. As the AW only requires the input header file, AW
makes the automatic wrapping process much simpler and
more flexible than traditional methods. There is no prereq-
usite of knowledge of XML syntax used for the AW. All the
informative data about filter is included inside the header
class itself. The effectiveness and productivity of the AW
relies on the qualitative code style of the header file. The
AW needs to follow the ITK coding convention so that the

Figure 1. Screenshot of Automatic Wrapper at the
first step

Figure 2. Screenshot of Automatic Wrapper at the
second step after parsing

Figure 3. Automatic Wrapper module for ITK style
filter
AW can recognize efficiently the patterns and data from the header.

In addition, as the AW provides an automatic mechanism to utilize ITK coding style filters, to improve the productivity and code reusability, once the AW determines a filter class’ inheritance hierarchy, the AW reuses some predefined highly abstract level wrapper filters. Based on the parsed data, the AW can select an appropriately corresponding top hierarchy for each filter.

Additional efficiency of the AW comes from the use of hash-tables. That is, the AW builds hash-tables to store all collected information during its parsing and processing data steps. At the end of the process, the AW can quickly retrieve informative data in order to produce wrapper files in ITK-Wrapping style. More importantly, the generated wrapper files can be compiled into a plugin module to be loaded as a plugin. The plugin module will be described in more detail in Section 3.3.

3.3 The Plugin Mechanism

Plugins are commonly used in modern software programs [8]. The main goal of a plugin architecture is to incorporate additional modules into the existing software without the need to recompile the software source, in order to minimize the risk to the main system. Under the plugin module, the input plugin file in shared library format(.so) will be imported into the system. The .so file may be a single filter or a group of filters. Once the filter is loaded, it is added into the filter library, which is a pre-compiled set of filter objects.

Figure 4 shows the basic steps for writing a filter plugin.

![Figure 4. Basic steps in writing a plugin using the public API with C++ illustration](image)

Most of the time, a filter plugin may involve GetParameters and SetParameters to handle normal parameters which are scalar or vector types. GetExtraParam and SetExtraParam API methods are also available if there is a need to support different types of parameters. The plugin filter code differs from the Automatic Wrapper as it requires one to implement the create method that includes code to initialize the filter object itself.

4. System Design

In this section, the description focuses on different design architectures for GUI and library by introducing the High Level and Detailed Design.

4.1 High Level Design

The High Level Design shows different abstract components in which each component becomes a subsystem. Each subsystem may in turn contain more subsystems itself. The High Level Design UML diagram given in Figure 5 illustrates the top-down viewpoint of the design principles.

![Figure 5. High Level design shows the relationship and interaction among different classes and modules at an abstract level. Viewer, Source, Output and Filter are all inherited from the ImageFilter abstract class.](image)

4.2 Detailed Design

In the High Level Design, details of some modules have been hidden. These are described as follows.

**ImageFilter**  The ImageFilter in the design is a highly abstract class which defines the most common attributes and operations. Apart from shared features, all the sub-classes of this class have fairly different characteristics of their own. Therefore, they have been separated into different branches of the hierarchy.

**Viewer**  This component provides the facilities to examine the output from a filter. The filter’s output is the input for the viewer. The use of a connector with an adapter makes...
the viewer adaptable to different inputs. The viewer enables users to examine the output from the list of connected filters.

**UI Component** The **UI Component** is used to provide the user interface. Figure 6 shows different classes which constitute the UI Component. As well as providing a GUI for

**Figure 6. The UI Component UML class diagram.**

ITKBoard, the UI Component plays an important role to facilitate users of the system. There are three main components, which are ITKBoardUI, ITKBoardChild and Design. Figure 6 provides the details of these classes as well as the relationship among them. All of them extend from their corresponding super classes in the wxWidgets library [13]. From Figure 6, it is clear that the core functionality will be implemented in ITKBoardUI and Design. The Design is the class which allows the user to draw filters and create the pipeline filter.

**ITK Filter Wrapper** The **ImageSource** component defines a high level wrapper of the ITK image reader filters. Each of its implemented child classes is the start point of the filter pipeline as it provides the data source. It creates an Image object from a file and holds the image data for subsequent filters. The **Output** component holds the final output of the pipeline. The image from the Source is processed through the pipeline and is eventually transformed into a potential output and stored in the Output filter. This output can be written into a file with a given name. On the other hand, the **Filter** provides a high level wrapper of ITK scalar image filters. Figure 7 shows a sample hierarchy of wrapper filters in the ITK filter library. To some extent, the filter wrappers in ITKBoard have a basic filter hierarchy identical to the ITK library. Its design is also based on generic programming approach (i.e. using templated code) [1]. Similar to the Filter hierarchy shown in Figure 7, some hidden components such as ImageToVector and VectorToImage are also included.

**ITKBoardController Component** The **ITKBoardController** handles most user events. It is not responsible for displaying the user interface but it is involved in callback functions from the UI component. It is considered as a core component to manage the available filter objects for each of the Design object and serves a numbers of callback functions such as code generation, finding filter objects and viewing the output image.

When a Design object requests a new filter object, the Controller object will forward the call to the Library object by the Find operation and get the filter object from the Library. As each object in the Design is assigned a unique identification key, the Controller will assign the returned filter a unique key. The Controller handles Viewers, ImageSource, Filter and Output objects. These objects are created dynamically according to requests from user interaction. For example, when the user creates a new Filter in the pipeline, an instance of that Filter is created dynamically and stored in the memory. In ITKBoardController, all references of created Viewers, Sources and Outputs during the runtime are stored in the hashtables with unique keys so that the time to process callback functions can be reduced significantly.

**Other Modules** The **XML Model** component provides an interface for the Controller to enable XML capabilities in the ITKBoard application. Therefore the main functions are reading and writing an XML file in order to save or restore filter pipeline designs. The XML parser can be written using the Xerces-C++ library\(^1\). The **Code Generator** compo-

\(^1\)Xerces-C++ is a validating XML parser written in a portable subset of C++ [14].
Figure 8. A pipeline using the Region Growing algorithm to segment the brain ventricle.

ITKBoard provides an API for generating code based on a certain template. The template is similar to the style of the code to build ITKBoard and has been modified to generate C++ .h and .cxx files.

5 Experiments and Results

ITKBoard provides a friendly user interface with embedded functions for performing filter pipeline design and experimentation. Every node on the filter tree corresponds to a unique filter, and can be dragged onto the design panel. Users can easily expand the tree to find their desired filters for the pipeline. Figure 8 shows the application’s user interface.

Multiple designs can be created at the same time by using separate tabs. By dragging desired filters from the filter tree, users can design a filter pipeline that represents a conceptual model. An example pipeline design is shown in Figure 8. Once the filter pipeline is constructed, each filter only has the default parameters. The user may need to customize the filter in the pipeline by setting suitable values for parameters. The effectiveness of each conceptual model for solving an image processing problem depends on the appropriate choice of parameters. ITKBoard provides direct interaction with each filter in the pipeline with a simple double-click or right-click on a filter to access its property parameters.

The pipeline displayed in Figure 8 aims to segment the brain ventricle for further investigation. The comparison of the original 3D image and output 3D image of the filter pipeline is shown in Figure 9.

Figure 9. Comparison of input and output for brain ventricle segmentation.
Discussion and Conclusion

In this paper, we have presented a visual tool for rapid prototyping of, and experimentation with, medical image analysis filters. We have also discussed the design methodology and methods for resolving some critical issues. Some advantages and further work are discussed below.

Advantages

- Rapid pipeline design. The filter pipeline can be designed quickly using drag-and-drop from the filter tree as well as extra shortcuts as mentioned in Section 5;
- Simple parameter tuning. It provides a flexible GUI mechanism for parameters setting of different filters on the fly. It also helps to overcome the disadvantage of parameter setting in image analysis;
- Examinable output. ITKBoard allows one to examine pipeline output on the fly;
- Multi-threaded environment. The system utilises multiple threads so that the GUI will be continuously responsive even during the execution of a computationally intensive task;
- Conceptual model verification. The system allows visualization at the conceptual level of the image analysis process;
- Extensibility. A plugin interface allows ITKBoard to be extended for third-party and closed-source algorithms;

Further Work

Further research will focus on providing modules to support the visual representation of high level language constructs [6]. Such modules permit more sophisticated processing [10]. For example, the system can be configured so that a certain part of the pipeline can be executed multiple times or only under certain circumstances. In addition, more error handler routines should be implemented to properly handle unexpected exceptions which may crash the system.

Although ITKBoard will undergo further improvement, the current system already provides a framework meeting the goals defined above. The methodology and techniques discussed in this paper will be applied to future research and development.

It is envisaged that when the software is released in the near future it will be significantly beneficial to both the scientists and clinicians in the medical image analysis community.

Acknowledgements

The data used for segmentation example presented in this paper was provided by A/Prof. Simon Warfield from the Surgical Planning Laboratory, Department of Radiology, Harvard Medical School, Brigham and Women’s Hospital, Boston, MA, USA. The work benefited from the use of the ITK as an initiative from the U.S National Library of Medicine.

The authors are grateful towards A/Prof. John Potter of the School of Computer Science and Engineering, the University of New South Wales and Mr. David Hellier of BioMedIA Lab, CSIRO for their critical reviews of this paper.

References