J Digit Imaging (2014) 27:220–230
DOI 10.1007/s10278-013-9667-7

SimITK: Visual Programming of the ITK Image-Processing Library within Simulink

Andrew W. L. Dickinson · Purang Abolmaesumi · David G. Gobbi · Parvin Mousavi

Published online: 9 January 2014

Abstract The Insight Segmentation and Registration Toolkit (ITK) is a software library used for image analysis, visualization, and image-guided surgery applications. ITK is a collection of C++ classes that poses the challenge of a steep learning curve should the user not have appropriate C++ programming experience. To remove the programming complexities and facilitate rapid prototyping, an implementation of ITK within a higher-level visual programming environment is presented: SimITK. ITK functionalities are automatically wrapped into "blocks" within Simulink, the visual programming environment of MATLAB, where these blocks can be connected to form workflows: visual schematics that closely represent the structure of a C++ program. The heavily templated C++ nature of ITK does not facilitate direct interaction between Simulink and ITK; an intermediary is required to convert respective data types and allow intercommunication. As such, a SimITK "Virtual Block" has been developed that serves as a wrapper around an ITK class which is capable of resolving the ITK data types to native Simulink data types. Part of the challenge surrounding this implementation involves automatically capturing and storing the pertinent class information that need to be refined from an initial state prior to being reflected within the final block representation. The primary result from the SimITK wrapping procedure is multiple Simulink block libraries. From these libraries, blocks are selected and interconnected to demonstrate two examples: a 3D segmentation workflow and a 3D multimodal registration workflow. Compared to their pure-code equivalents, the workflows highlight ITK usability through an alternative visual interpretation of the code that abstracts away potentially confusing technicalities.

Keywords Software design · Image processing · Image segmentation · Image registration · Visual programming

A. W. L. Dickinson (✉) · P. Mousavi
School of Computing, Queen's University, Goodwin Hall, 25 Union Street, Kingston, ON K7L 3N6, Canada
e-mail: andrew@cs.queensu.ca

P. Abolmaesumi
Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, Canada

D. G. Gobbi
Department of Radiology, University of Calgary, Calgary, Alberta, Canada

Introduction

Medical image analysis is typically performed with the aid of software designed to manipulate and augment medical images. Such manipulation enables deriving surgical plans, registering information between different modalities, and making regions of interest clearly visible. The Insight Segmentation and Registration ToolKit (ITK) [1] is an open-source, cross-platform collection of C++ image-processing classes and libraries commonly used in developing software for research in clinical applications of medical image analysis.

Classes in ITK have individual functionalities and are broken down into many different types, such as file readers/writers and image filters that perform image-processing techniques. As of March 2013, ITK consisted of 12,985 files composed of 2,445,718 lines of code. Of these files, approximately one-third are classes with specific functionalities; the remaining files aid in the building and installing of ITK or provide documentation.

Medical image analysis software is typically built using a dataflow paradigm or "pipeline," where one selects and connects desired classes to one another to form a program where, upon execution, the data will flow and be manipulated by each specified subsequent class until completion; a graphical representation of a typical ITK pipeline can be seen in Fig. 1a.

Constructing an ITK program requires familiarity with advanced C++ concepts such as generic programming and is further complicated by a proliferation of ITK-specific data types. These potentially heavy programming requirements are a major setback, keeping libraries like ITK from being commonplace within a biomedical research laboratory or a clinical environment. A solution that abstracts away these programming complexities would greatly reduce the required learning curve and increase accessibility as the user could learn how to build and manipulate programs without needing to first learn how to program. At the same time, a means to rapidly prototype ITK programs that can showcase and take advantage of the powers of these libraries would be of great benefit to both present and future researchers.

As detailed in "Background," several solutions have been proposed before to alleviate the issues above and facilitate ITK programming, including ITKBoard [2], MeVisLab [3, 4], and MITK [5, 6]. However, these solutions rely on a proprietary graphical user interface (GUI) unique to each implementation. In such cases, this convenience substitutes deep ITK knowledge for deep environment knowledge putting the user at risk of becoming proficient in the particular environment and not the underlying library. This either forces the user to remain within the familiar environment or begin anew and learn how to perform the same tasks within a new environment.

Alternatively, since we observed that the pipeline-based nature of ITK and the visual programming approach employed by Simulink, an environment within the MATLAB (Mathworks, Natick, MA) scientific programming suite, are both based on dataflow programming, combining the two environments was a natural conclusion. Simulink is an interactive graphical environment within MATLAB created for model-based design of dynamic systems, complete with a customizable set of block libraries that facilitate the design, simulation, implementation, and testing of a variety of time-varying systems, including, but not limited to, image processing. From a given block library (or combination thereof), desired blocks are selected and added to a blank Simulink canvas before being interconnected to create a model file. Subsequently, the model file is executed to perform a given task. Figure 1b is an example of a Simulink signal-processing model.

In this article, we present SimITK, an implementation of ITK within the Simulink environment that joins the power of image processing with the simplicity of visual programming. A benefit of graphical environments like Simulink, particularly when the user is not an experienced programmer, is that details of a written programming code are abstracted away and replaced by an equivalent visual representation, allowing the user to focus on solving the problem at hand. As shown in Fig. 1, similarities can be observed between pipeline-based programming and visual programming models: blocks represent data or performed actions and arrows represent information passed between blocks.

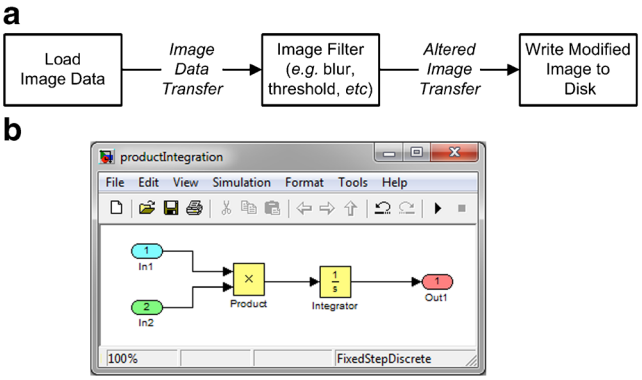

Fig. 1 A comparison of pipeline-based ITK programming and graphical programming within Simulink. If blocks are incorporated into Simulink from an image-processing library such as ITK, then Simulink can be used as a visual programming environment for image processing. **a** A flowchart representing the pipeline-based nature of ITK programming where classes are connected to one another to create a virtual information "processing chain". **b** In a Simulink model, the inputs, outputs, and the processing blocks are displayed graphically in the Simulink window. For example, the model above computes a product and then an integral (highlighted in *yellow*) of two different inputs (in *cyan* and *green*) and produces the result as an output (shown in *red*)

Background

The dataflow programming paradigm was initially envisioned in the late 1960s (Adams [7]) and early 1970s (Dennis [8]) out of an increased desire to design a method of programming based on the connections, or flow, between program elements as opposed to focusing on the data changes occurring as program execution progressed. Typically, this programming style is modeled similar to a graph diagram representation, as shown in Fig. 1.

Another paradigm, visual programming, aims to substitute the written aspect of programming with functionally equivalent visual representations. All programming elements, such as input data and executed commands, are represented by graphics that interconnect to create a program. As an example of an early visual programming implementation, in pictorial transformations (PT) [9], the user performs "algorithm animation." A programmer specifies an algorithm in PT by altering and interacting with visual representation objects. By the nature of the programming, as the "animation" is visually constructed so is the algorithm. This is only one of many of the concepts introduced in PT that parallel the Simulink environment. However, since ITK exists as a C++ library without any standard visual programming front-end, the technique of "language wrapping" can be employed in order to incorporate ITK into the visual programming paradigm used in Simulink.

Wrapping is a computer language translation technique that aims to translate just the interface specification of a programming library from one computer language to another, while leaving the implementation in the original programming language. This is commonly done to facilitate software library interaction written in the former language in terms of the latter environment. The Simplified Wrapper and Interface Generator (SWIG) [10] is an example of a tool that can be used to take libraries and/or programs written and implemented in C or C++-like programming languages and integrate them within higher-level programming environments. WrapITK [11] is a front-end for SWIG designed specifically for use with the ITK image-processing library, which uses a powerful tool called GCCXML to create a symbolic representation of the ITK interface specification. As such, WrapITK represents a crucial component necessary to encapsulate ITK within Simulink to create SimITK. The implemented method (detailed in "Methods") is a complex wrapping process that integrates the C++ ITK library into the Simulink visual programming environment facilitates the rapid development of ITK programs in the form of SimITK workflows.

There are several image-processing environments that aim to provide the user with enough flexibility to construct custom pipelines, without requiring knowledge of programming. Implementations include, but are not limited to, SCIRun [12], ANALYZE [13], MATITK [14], 3D Slicer (Slicer) [15, 16], and XIP [17, 18].

In most of these implementations, the convenience of ITK integration carries a potentially heavy setback: deep ITK knowledge is substituted for deep environment knowledge. The danger of this is that the user becomes competent in the environment and not the implemented library; moving to a different environment may require relearning leading to prolonged development. Furthermore, when collaborating or presenting results, having the environment installed becomes a necessity. As such, with ITK becoming increasingly ubiquitous, it may be of greater interest to take advantage of the functionalities ITK offers by being familiar with its inner workings rather than a given implementation environment.

As with SimITK, several other visual programming environments have integrated ITK's image-processing abilities through the visual programming paradigm. Examples include, but are not limited to, ITKBoard [2], MeVisLab [3, 4], MITK [5, 6], and VolView [19]. In the case of ITKBoard, this includes the convenience of automatically wrapping ITK. However, they all differ sharply as a proprietary GUI has been developed exclusively for each implementation. SimITK differs from the rest in that it uses WrapITK, which provides a comprehensive, automatic translation of the ITK interface instead of only translating a hand-picked subset. Furthermore, the use of Simulink is another difference which couples the power of ITK with the scientific computing abilities of MATLAB, allowing for the creation of more complex workflows.

All these previously detailed concepts will be fused together to create the SimITK visual programming implementation of the ITK image-processing library. As outlined, the ITK image-processing library uses dataflow programming concepts to build programs composed of connected classes. These ITK class interconnections can be naturally represented by the visual programming paradigm found at the core of Simulink. Therefore, to successfully integrate ITK within Simulink, a language-wrapping procedure is used to combine the image-processing power of ITK with the visual ease of Simulink to create SimITK. The next section extensively outlines the SimITK wrapping process including the steps taken, challenges faced, and solutions implemented.

Methods

A graphical overview of the wrapping method, the different required files, and the order in which they are created can be seen in Fig. 2. As a convention, the files at the top of the figure are lowest on the dependency tree, meaning they must be generated before the files lower in the figure.

The implemented method for wrapping ITK classes as SimITK blocks requires a simple and legible representation of each ITK class at its core. This transparent representation is critical because the information stored therein will be retrieved frequently for substitution into several different, custom-written template files—one for each filetype needed—to successfully integrate ITK within Simulink. Within the representation, pertinent class information needs to be stored such as the name, type (Image Filter, Optimizer, Transform, etc.), parameters, as well as all inputs, outputs, methods, method arguments, acceptable dimensionalities, and permitted data types.

The Extensible Markup Language (XML) was selected as the language best suited for creating a schema, an organizational structuring of the contained data, which would allow for storage and retrieval of the ITK class information. Another reason for selecting XML stemmed from the desire to take advantage of WrapITK, an optional part of ITK that generates a complete, yet complex, XML representation of an ITK class. However, this complex document requires significant consolidation to make it useful within the SimITK wrapping procedure.

After an XML representation of each ITK class is generated using WrapITK and refined into a usable state, the ITK information therein is used to generate all SimITK source files by expansion of keywords within specialized SimITK-template files: special keywords within the templates serve as anchor points to be replaced by ITK information and/or

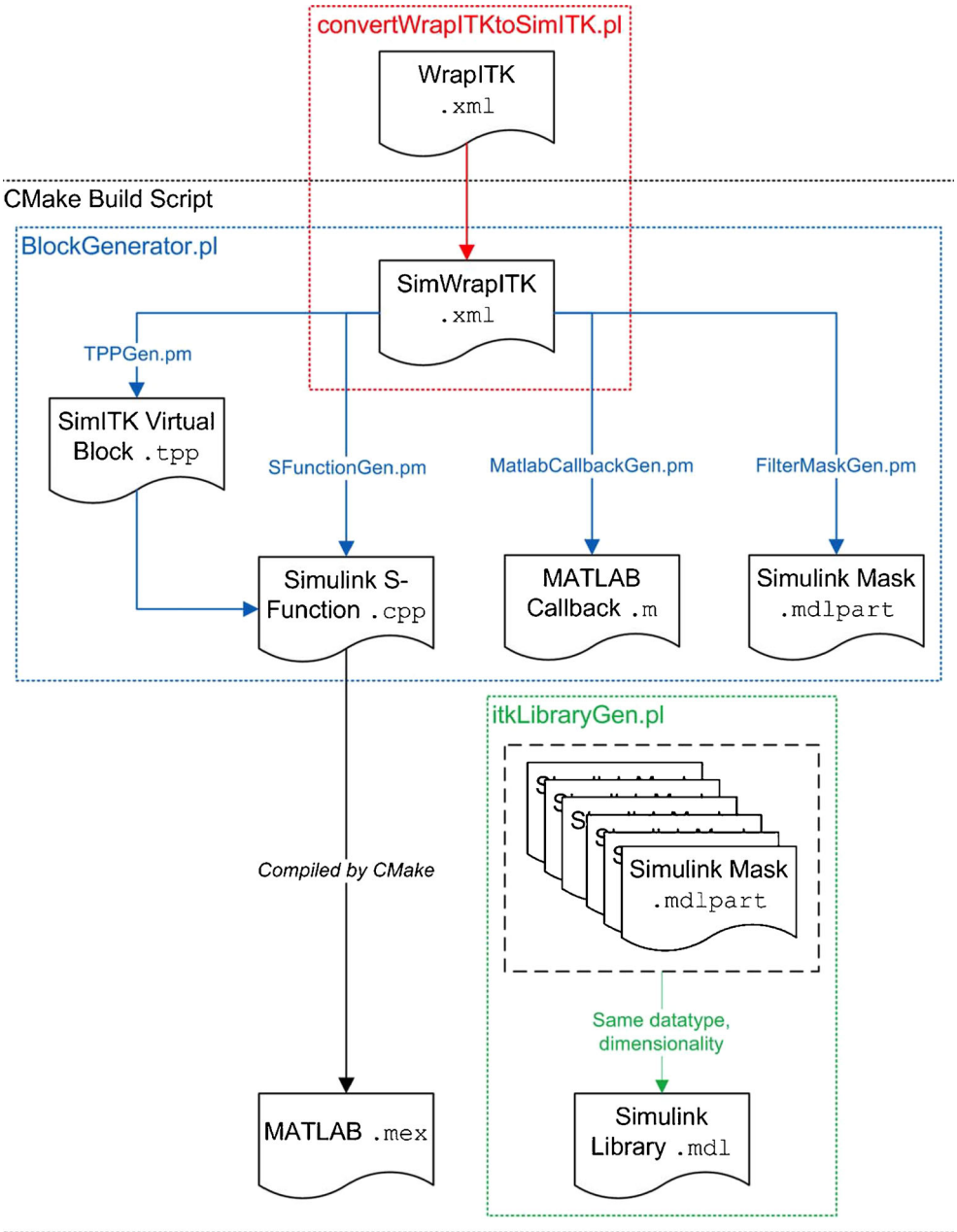

Fig. 2 A graphical overview of the SimITK wrapping procedure. The dependency tree is constructed such that a parent file is a required dependency of a child file (e.g., the .cpp depends on both the .tpp and SimWrapITK .xml files being generated before it can be generated itself)

integration code. One series of substitutions exists for each template with one template for each filetype. Several perl scripts and modules were written to perform the substitution series for each template in order to generate class-specific files.

As illustrated, after the XML class information has been retrieved, it is substituted into a template for the Virtual Block .tpp file: a data converter that exists between the Simulink and ITK workspaces. The subsequent files require the Virtual Block to be created prior to subsequent blocks being generated.

Following .tpp generation, several Simulink/MATLAB source-code files need to be generated. One is a Simulink "S-Function" .cpp file that contains the code that will be executed by Simulink at runtime. Two accompanying files provide the block representation within Simulink: a MATLAB Callback .m file and a Simulink Mask .mdlpart file. These files detail information such as the number of ports on a given block as well as establish the options the user can modify within the "block mask," a dialog box that appears when the block is double-clicked that facilitates block configuration (e.g., providing the filename for image data to be loaded). These files are generated like the .tpp file by taking their respective custom filetype template and applying a series of substitutions on custom keywords therein.

Next, the .cpp files are compiled into MATLAB-specific .mex files that contain the code executed by Simulink at runtime.

Finally, since an image-processing pipeline depends strongly on the data type used to store the image (16-bit integer, 32-bit floating-point, etc.) and the type of image being manipulated (2D image or 3D volume), this procedure is repeated to generate corresponding Simulink .mdl library files composed of all similar .mdlpart files for each combination of data type and dimensionality.

XML Creation

Representing the ITK classes in a simple, standardized, and legible format was absolutely critical for ensuring an effective wrapping procedure—a required consideration as the XML information needed to be later substituted into the various file templates. "WrapITK" details the optional WrapITK package and the challenges therein, the solutions to these problems, and the final structure of the SimITK XML schema.

WrapITK

WrapITK [11] is an optional part of an ITK installation that aids in wrapping the library into other non-C/C++ languages and/or environments. GCCXML, a program used by WrapITK, generates a complex XML representation of each ITK class according to specified configuration options. This output representation contains all class information including name, parameters, data type, dimensionality, methods, method arguments, inputs, and outputs as well as any required ITK types and associated superclasses (and respective set of pertinent class information—the same aforementioned information list) for all related superclasses in the ITK-hierarchy. An example of this XML can be seen in Fig. 3a. To give the reader a more concrete understanding for the desired output from the XML conversion, the final representation for this same method can be seen in Fig. 3b.

a

```
<Method id="_97" name="SetUpperThreshold"
  returns="_24" virtual="1" overrides="" context="_8"
  access="public" mangled="_ZN3itk29CannyEdge
——→DetectionImageFilterINS_5ImageIfLj2EEES2_E17
——→SetUpperThresholdEf"
  demangled="itk::CannyEdgeDetectionImageFilter&lt;
——→itk::Image&lt; float ,_2u&gt;,
——→itk::Image&lt; float ,_2u&gt;&gt;
——→::SetUpperThreshold(float)"
  location="f1:198" file="f1" line="198" extern="1"
  inline="1">
    <Argument name="_arg" type="_54c" location="f1:198"
      file="f1" line="198"/>
</Method>
```

b

```
<Parameter>
  <Parameter_Name>UpperThreshold</Parameter_Name>
  <Parameter_Type>ImageType</Parameter_Type>
  <Parameter_Size>1,1</Parameter_Size>
</Parameter>
```

Fig. 3 A comparison of the XML files generated by WrapITK and what is ideal for SimITK. **a** Sample WrapITK XML. **b** Final SimWrapITK XML

It can be observed from Fig. 3a that while some of the information can be easily extracted by reading the XML (the method name from the "name" element, for example), there is much information to be desired from this code that is not directly apparent: class-specific information like dimensionality, data type, and whether data are related. Though data relations can be determined through comparing ID and context values associated with each entry, any possible relations are not immediately apparent and difficult to understand.

As previously mentioned, all acceptable data types and dimensionalities are included within the initial XML document generated by WrapITK. Since an ITK class is capable of handling image data stored in different dimensionalities and data types, the XML representation describes a "class template instance" for each combination. For example, if a given class can handle data in either two or three dimensions, of either a float or a short data type, four template instances will exist: 2D-float, 3D-float, 2D-short, and 3D-short. These multiple instances of the same class template cause problems when attempting to resolve if/when variables corresponding to the data type or dimensionality were used within the class instance. These "template variables" act as placeholders—the exact value is not important, but knowing that this variable corresponds to information that depends on the class instance is critical. Discovering when these variables have been used is extremely difficult as GCCXML resolves these variables to the instance information within its output XML. As such, template variables must be reverse-engineered from the class instances in order to re-establish when such variables were used. This requirement, as well as all other pertinent class information, can then be reflected within the final SimWrapITK XML to be used throughout the wrapping procedure.

The solution used to integrate the ITK class information into the SimITK wrapping procedure was to process and refine the original WrapITK XML into a simple and transparent class representation. This was accomplished through the creation of convertWrapITKtoSimITK.pl, a perl script that used the XML::DOM perl module to analyze the WrapITK XML file for a given class, construct a hierarchy of XML nodes based on the relations between the entries, and produce a final XML document containing the extracted class information in an easily retrievable format.

After this process had been performed for each ITK class file, the data could then be integrated into the various template files using the "custom-keyword substitution" technique outlined next.

XML Substitution

The technique employed in substituting the various templates with the appropriate information seeks out special keywords

within the templates and replaces them with appropriate XML information and/or extra code. To ensure unique keywords, the variables were pre-pended and appended with an ampersand or "at symbol" (@). The various file templates and their role in the wrapping procedure will be outlined next in the following sections.

File Generation

As outlined in Fig. 2, the "Virtual Block" .tpp is the first file template substituted with the class XML data and subsequently compiled. The .tpp file serves as a communication layer between the Simulink and ITK workspaces. The Virtual Block is aptly named as it exists transparently to the user yet serves the important task of converting special ITK data types into appropriate Simulink data types and vice versa. It is also responsible for instantiating the ITK class that is created at simulation runtime as well as ensuring that all the arguments specified within the workflow are established as specified by the user. The .tpp file for a given class is required as a dependency in the creation and generation of all subsequent files in the wrapping procedure.

With the .tpp file generated, the next file to be keyword-substituted is the Simulink S-Function .cpp file template that contains the C and C++ source code for the Simulink block. It is divided into multiple methods that are executed at different steps when executing a Simulink workflow. Once substituted with the XML information, the .cpp file is ready to be compiled into its final .mex file—the file executed at Simulink runtime.

Several additional files required generation using the same method as the .cpp and .tpp files. These included the MATLAB Callback .m file (contains the code necessary to alter the GUI elements within the block "mask": the dialog box the user controls to customize a given ITK class' method arguments); the .mdlpart file (containing the actual code representation of the GUI elements); one .mdl library file for each combination of Image Filter data type and dimensionality, Transform dimensionality, and one library for all Optimizer classes.

Build Automation

Creation of the various files is handled individually by the previously mentioned collection of perl scripts and modules. In order to automate the process, a build script was written in CMake [20], the same build tool used when configuring ITK, which takes all the input XML and processes them into each template by executing the perl scripts/modules in sequence, appends the .mdlpart files together into libraries, and finally compiles each Simulink S-Function .cpp file into its final .mex MATLAB-executable library.

A series of perl scripts and modules were written to substitute the keywords within the templates with the appropriate XML information. One main perl script was responsible for managing the substitutions while the creation of the final .mdl files was handled by a second script.

Results and Discussion

The results from the SimITK wrapping procedure are organized into several sections to reflect the various steps of Simulink block generation, evaluation of SimITK runtime, and the website created to promote and support the project.

SimWrapITK XML Generation from WrapITK

As previously described in "XML Creation," the convertWrapITKtoSimITK.pl script is executed to generate SimWrapITK class representations of previously generated WrapITK XML documents. From these generated SimWrapITK files, the information therein can be successfully substituted into the templates for the several required files, and compiled when necessary, to complete the SimITK wrapping procedure. Following this step, construction of SimITK workflows is possible within MATLAB/Simulink.

Simulink Block Libraries

The primary result from the SimITK wrapping procedure is the generation of multiple Simulink Block libraries. Eight Image Filter libraries are created: one for each combination of data type (float, short, unsigned short, unsigned char) and dimensionality (2D or 3D); two Transform libraries: one for each dimensionality; and one library for Optimizers used in registration frameworks. To date, 129 Image Filters, 6 Transforms, and 7 Optimizers have been wrapped in entirety. A screenshot of an example Image Filter library can be seen in Fig. 4.

Workflow Creation

With the libraries generated, it is possible to create SimITK "workflows," Simulink models composed of SimITK blocks that perform a given task. The first required step is to select and place desired blocks as objects within a new Simulink model file. This is accomplished through clicking a given block and dragging it from the library onto the Simulink model workspace, or "canvas," as shown in Fig. 4.

With the desired blocks placed on the canvas, interblock connections can be established that will pass information (such as image data) from block to block in order to process the data as directed by the workflow. This is achieved by

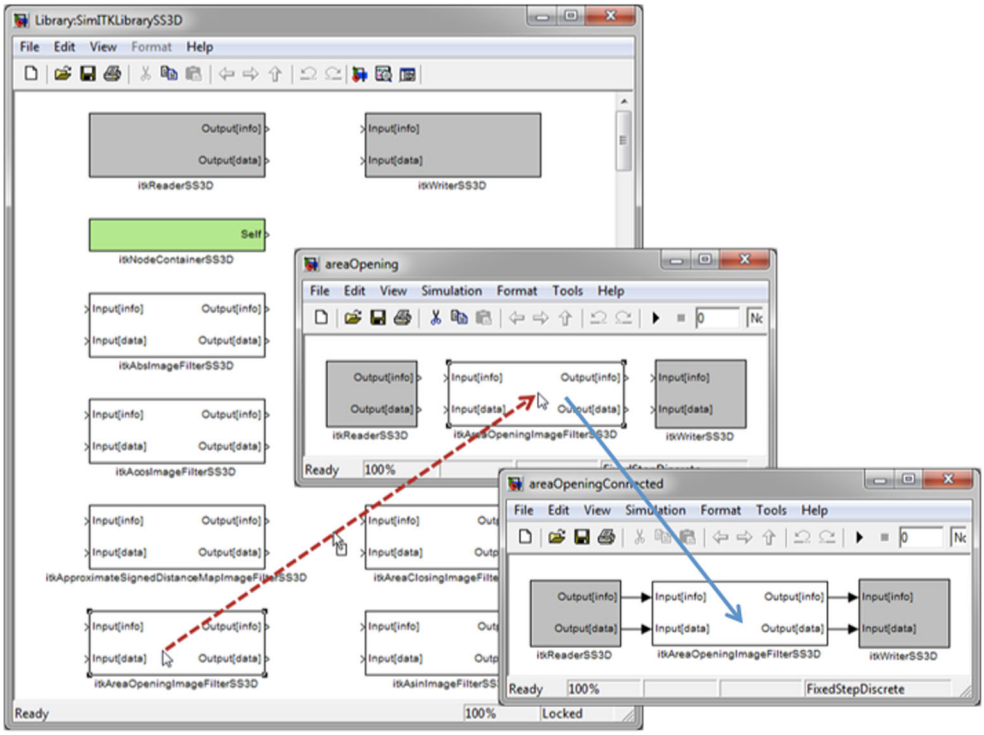

Fig. 4 A graphical representation of adding a block to a workflow canvas and the final connected workflow. An example SimITK Library can be seen in the background with a populated canvas inlay

simple clicking and dragging to connect the output ports of a block to the desired input ports of a consecutive block (Fig. 4).

The final step in workflow creation, prior to execution, is to set the desired method arguments for each class. This is accomplished through double-clicking a given block to reveal its Simulink "mask dialog," the control interface for the block (Fig. 5a). In this dialog box, desired methods can be enabled for use within the workflow. When enabled, a text-input field is revealed where desired method arguments can be entered. In cases like the Transform and Optimizer blocks, the mask contains checkboxes that can enable/disable the input or output of additional information. Figure 5a is an example of an itkCenteredEuler3DTransform3D block mask where multiple inputs and an output can be specified, if desired. As a further example, Fig. 5b–d shows the same Transform block in several configurations depending on the enabled checkboxes within the mask.

Once a given workflow has been built of properly connected blocks, and corresponding mask dialogs adjusted as needed, the Simulink timer, or clock, needs to be set to the desired number of full workflow executions. With the clock set, Simulink can be instructed to start the simulation (i.e., execute the workflow and perform the ITK image-processing task).

Two in-depth examples are presented as case studies for SimITK. One demonstrates segmentation using pixel-intensity thresholding and another demonstrates MRI-to-CT registration using a Mattes Mutual Information metric, Gradient Descent optimization, and Nearest Neighbour interpolation. With respect to the data used in these examples, the segmentation cranial data (Fig. 6, lower left) can be found within the Examples directory of a base ITK installation, and the registration cranial CT (Fig. 8a) and MRI data (Fig. 8b) can be acquired from the American National Institute of Health Visible Human Project.

Case Study 1: 3D Segmentation

An example SimITK workflow that segments the skull from cranial CT data is shown in Fig. 6. The first block is a file reader used to load the 3D image data and pass the information to a threshold filter that replaces the pixels below the user-set pixel-intensity threshold level with black, while highlighting the bony areas above the threshold in white. This modified volume is then written to disk for future use.

As a point for comparison, the three-block SimITK workflow for this example was also implemented as a C++ ITK application, which required 86 lines of code.

Case Study 2: 3D MRI to CT Registration

Another workflow example performs a registration of MRI and CT data of the same cranium is shown in Fig. 7 (corresponding data and output can be seen in Fig. 8). Similar to the previous example, one file reader is used to load each of the MRI and CT data as separate inputs to a registration method using a Centered 3D Euler transform, Nearest Neighbour interpolation, Mattes Mutual Information metric, and Gradient Descent registration optimization. Following

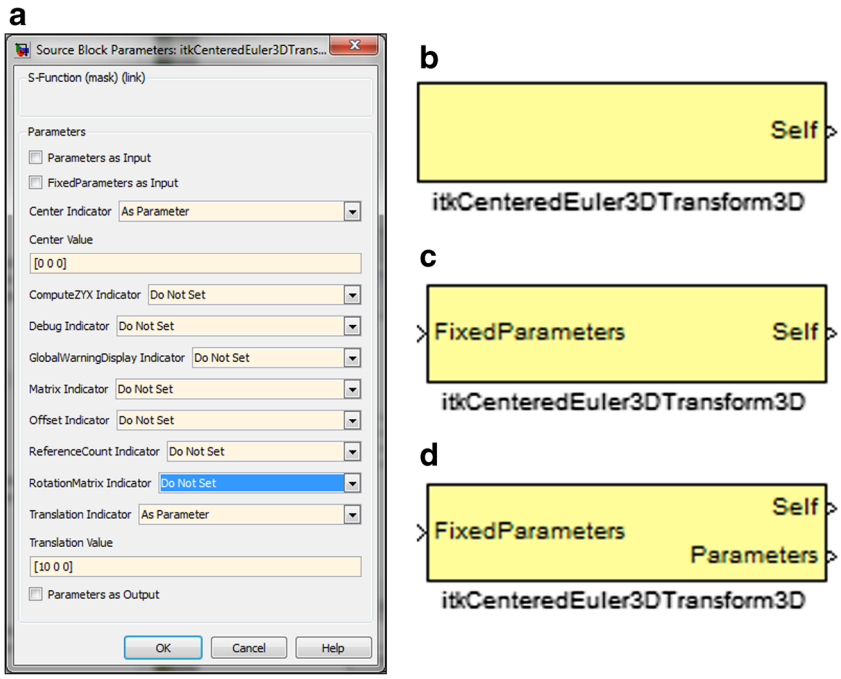

Fig. 5 Block modifications. **a** An example mask dialog. **b** The default state for the itkCenteredEuler3DTransform3D block. **c** The same block with the *FixedParameters as Input* checkbox enabled. **d** The block once again with both the *FixedParameters as Input* and *Parameters as Output* checkboxes enabled

registration, the MRI data are processed with the optimally calculated registration parameters before being written to disk as a new file. Upon completion, the final value for the Metric (in violet) and the Transform parameters (in yellow) used to create the optimal alignment will be printed in their respective labelled blocks in the workflow.

As a point for comparison, the 15-block SimITK workflow for this example was also implemented as a C++ ITK application, which required 296 lines of code.

StepByStepImageRegistrationMethod

To complement the automatically wrapped built-in ITK classes, a custom ITK class, StepByStepImageRegistrationMethod, was created for inclusion with SimITK that allows Simulink to collect information like Metric value(s), Transform parameters, and any other desirable intermediary results during workflow execution. Functionally, the class operates identically to the standard ITK-supplied itkImageRegistrationMethod

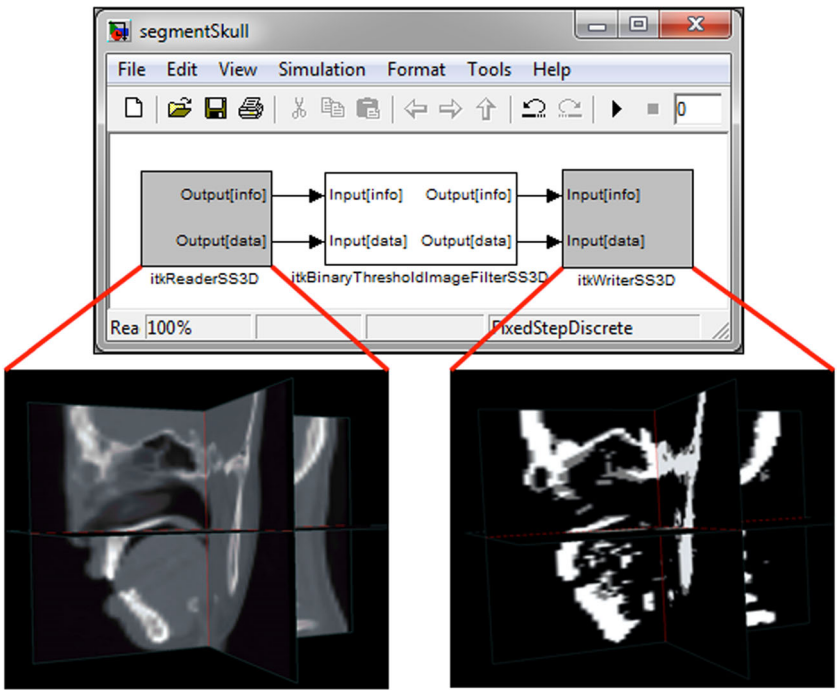

Fig. 6 An example workflow segmenting the skull from 3D Cranial Volume data. This three-block workflow replaces approximately 90 lines of C++ code

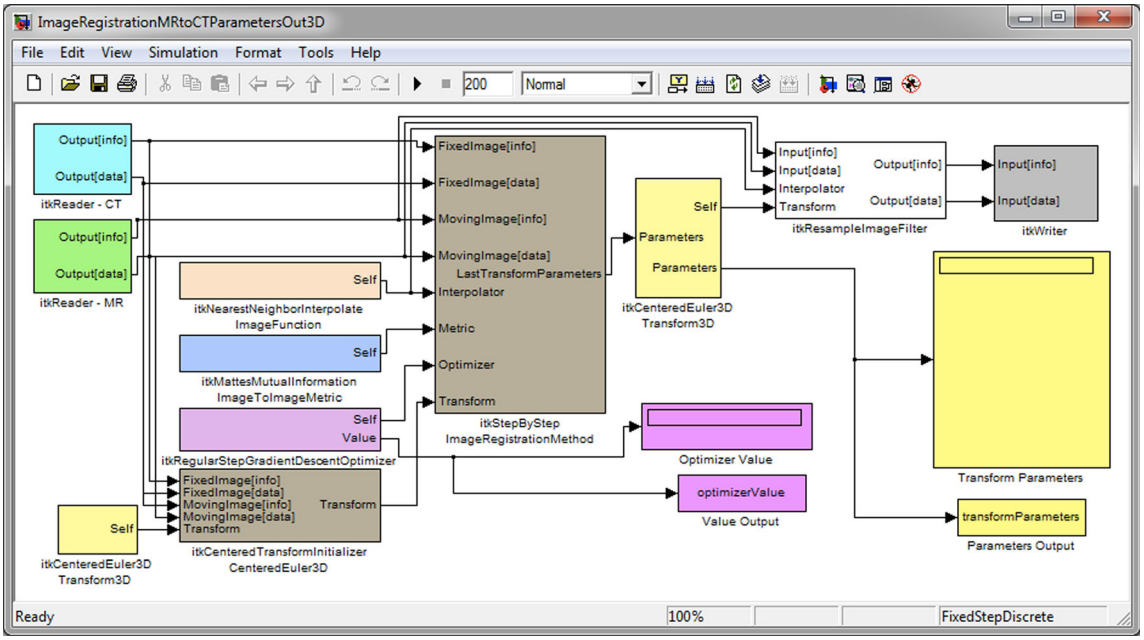

Fig. 7 An example workflow registering 3D MRI cranial data to 3D CT data. The 15 blocks illustrated here replace approximately 300 lines of C++ code

except that the iteration of the ITK optimizer is tied to the iteration of the Simulink clock. Typically, the Simulink clock is responsible for establishing the number of desired full workflow executions; StepByStepImageRegistrationMethod modifies this behaviour such that one Simulink clock iteration is equivalent to one ITK registration iteration. This allows the user to retrieve and store registration parameters from within the MATLAB/Simulink workspace in real time (i.e., with each iteration progressing towards registration completion). Figure 7 exemplifies this through the use of Simulink "To Workspace" blocks for the Value Output and Parameters Output blocks that save the data collected from either respective block at each iteration as a user-defined MATLAB variable.

Run-Time Performance

All results were computed using an Intel Core 2 Quad CPU Q9400 @ 2.66 GHz with 4 GB of RAM, Windows XP Service Pack 3, ITK 3.18, MATLAB/Simulink R2008b, CMake 2.8, and Visual Studio 2008. All benchmarking times were generated using the same C++ timestamp-generating

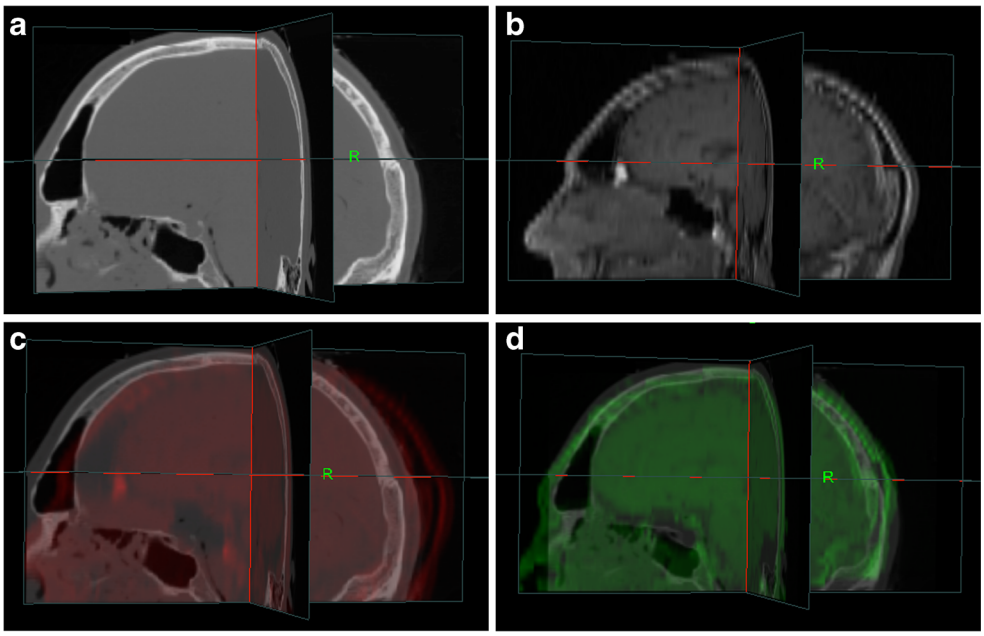

Fig. 8 Reference images of the initial CT and MRI data used in the Registration example (Fig. 7) as well as a comparison of the CT and MRI datasets pre- and post-execution of the registration implementations. **a** The CT data were processed into a .vtk volume file of 129 slices (at 1-mm thickness) of 245-by-255 pixel images. One pixel is equivalent to 0.898 mm. **b** The T1 MRI data were processed into a .vtk volume file of 34 slices (at 4-mm thickness) of 128-by-128 pixel images. One pixel equivalent to 1.016 mm. **c** The CT data and unregistered MRI data overlay in *red*. **d** The CT data and registered MRI data overlay in *green*

code injected at the same execution point within each respective code to ensure comparability.

For the segmentation workflow (Fig. 6), the runtimes are considered equivalent in both pure, written C++ ITK code and SimITK flowchart. The difference in runtime was found to be on the order of milliseconds. As such, implementing code to calculate a sub-millisecond time difference would have introduced time overhead well above this requirement without merit.

The registration example (Fig. 7) results for both the standard itkImageRegistrationMethod and the custom StepByStepImageRegistrationMethod are shown in Fig. 9, which also compares SimITK workflow runtimes to their pure-C++ equivalent. All implementations were executed for the same number of iterations and yielded the same final registration parameters.

It is worth noting that the decrease in runtime observed in Fig. 9 for SimITK workflows was not anticipated. Since the runtime comparison code behaved identically in each case (as the timestamp code executed was verbatim at equivalent points in each respective code), it is hypothesized that any discrepancy is due to MATLAB pre-allocating memory for all SimITK image storage, while ITK must make system calls to initialize, allocate, and manage memory.

Longer registration times are expected for the StepByStepImageRegistrationMethod compared to the standard itkImageRegistrationMethod. The former runs the registration for only a single optimizer iteration before returning control to Simulink in order to facilitate information retrieval prior to resuming registration. This results in an overall slowdown of program execution. The latter is coded to run the registration until completion without interruption.

The runtime results shown in Fig. 9 were presented using a specific set of case studies running a specific hardware configuration. While the results suggest that SimITK decreases time overhead, the runtime performance has not been further analyzed to ensure that similar results would be expected on other systems with configurations that differ from the development environment. For further validation, it is recommended that the created C++ programs and equivalent SimITK workflows are processed using a code profiler. This would provide a more complete insight with respect to the time discrepancies to give a more complete understanding as to the true nature of the runtime performance comparison between pure C++ ITK and SimITK.

	itkImage-Registration-Method	StepByStep-Image-Registration-Method
Pure C++ ITK Program	170 ± 1 seconds	600 ± 1 seconds
SimITK Workflow	158 ± 1 seconds	450 ± 1 seconds
Time Difference	12 seconds (8.8%)	150 seconds (25%)

Fig. 9 Comparison of MRI to CT Registration code runtimes (100 executions, each consisting of 100 iterations, timed until completion)

Conclusions and Future Work

In this work, SimITK was presented: a collection of block libraries that can be connected within the Simulink visual programming environment to create workflows that parallel their equivalent pure C++ ITK code. Constructing the workflow visually, instead of building a written pipeline of code, allows the user to focus on solving the image-processing problem with simple, intuitive graphical representations while not having to be concerned with potentially difficult programming nuances.

SimITK allows for the image-processing capabilities of ITK to be made readily available to users that would normally have to invest substantial time and effort into learning the details of ITK, and potentially high-level C++. This makes SimITK an ideal environment for educators hoping to teach the concepts of image processing within a classroom setting. Medical research groups can also use SimITK to test hypotheses by rapidly developing workflows as a starting point for experiments.

While this work is proof-of-concept that a subset of approximately 130 classes of ITK can be successfully wrapped and integrated into SimITK, there are areas for further development that would increase the impact and overall abilities of SimITK. A large, required development will be the expansion of the Virtual Block .tpp files as they are presently capable of handling only a subset of all the defined data structures and types specific to ITK. Many more types still require appropriate conversion rules before they can be properly integrated into SimITK. As such, this has limited the total number of classes within SimITK.

Simulink was the targeted visual programming environment because of the ubiquity of MATLAB, its parent application, within research and development environments. This ubiquity would allow for a large and established user-base to take advantage of ITK's image-processing capabilities within a familiar environment. Furthermore, since MATLAB can execute Simulink codes (and vice versa), any Simulink model file with a SimITK block can interact with any previously developed MATLAB code, increasing the variety and power of the toolset available to the user. Utilizing a previously developed, mature, and well-maintained graphical environment also eliminated the need for creating such an environment specifically for this project.

At present, each different data type and dimensionality yields its own Simulink Block Library. As such, a workflow created to segment, for example, three-dimensional float data cannot be used to segment short data, a duplicate workflow composed of blocks of the latter data type must be created. It is recommended that instead of resolving the template variables into their specific data types and dimensionalities at compilation time, they should be left as is (i.e., unresolved). Workflows could then be coupled with an initial block (not presently developed) that would establish the desired data type for workflow execution. This development would greatly increase the reusability of a created workflow allowing it to work for each of the wrapped data types by changing only one block instead of recreating entire workflows for each data type, when needed.

A website was created to promote SimITK, as well as its sister project SimVTK, and provides user and developer-level support. Present and past releases can be found on the website as well as extensive user and developer documentation. The documentation includes installation and configuration instructions for SimITK and prerequisite software, as well as tutorials and examples to guide a user through workflow creation. All examples are also supplied as pre-built Simulink files for immediate use. Demonstration videos, previous publications, and contact information are also included.

Acknowledgments We would like to acknowledge the Natural Sciences and Engineering Research Council (NSERC), Canadian Institute for Health Research (CIHR), and the NSERC Collaborative Research and Training Experience Program (NSERC-CREATE) for their assistance and support; Saba El-Hilo for her contributions towards the tutorials found on the project website; and Karen Li and Jing Xiang for their initial development groundwork on the project.

The intent to publish this work was on the parts of the authors; the study sponsors had no direct involvement.

References

1. Ibanez L, Schroeder W, Ng L, Cates J: ITK Software Guide: The Insight Segmentation and Registration Toolkit (Version 2.4). Kitware Inc, Clifton Park, New York, 2005
2. Le HDK, Li R, Ourselin S: Towards a visual programming environment based on ITK for medical image analysis. In Digital Image Computing: Techniques and Applications, p. 80. 2005
3. Koenig M, Spindler W, Rexilius J, Jomier J, Link F, Peitgen HO: Embedding VTK and ITK into a visual programming and rapid prototyping platform. SPIE Medical Imaging 2006: Visualization, Image-Guided Procedures, and Display 6141(1), 2006
4. Rexilius J, Spindler W, Jomier J, König M, Hahn HK, Link F, Peitgen HO: A framework for algorithm evaluation and clinical application prototyping using ITK. The Insight Journal - 2005 MICCAI Open-Source Workshop 2005
5. Wolf I, Nolden M, Böttger T, Wegner I, Schöbinger M, Hastenteufel M, Heimann T, Meinzer HP, Vetter M: The MITK approach. The Insight Journal - 2005 MICCAI Open-Source Workshop 2005
6. Zhao M, Tian J, Zhu X, Xue J, Cheng Z, Zhao H: The design and implementation of a C++ toolkit for integrated medical image processing and analyzing. SPIE Medical Imaging 2004: Visualization, Image-Guided Procedures, and Display 5367(1):39–47, 2004
7. Adams DA: A computation model with data flow sequencing. Ph.D. thesis, Stanford, CA, USA, 1969
8. Dennis J: First version of a data flow procedure language. In Proceedings, Programming Symposium, volume 19 of LNCS, pp. 362–376. 1974
9. Hsia YT, Ambler AL: Programming through pictorial transformations. In Proceedings of the International Conference on Computer Languages, pp. 10–16. 1988
10. Beazley DM: SWIG: an easy to use tool for integrating scripting languages with C and C++. In Proceedings of the 4th Conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4. USENIX Association, Berkeley, CA, USA, 1996
11. Lehmann G, Pincus Z, Regrain B: WrapITK: enhanced languages support for the Insight Toolkit. The Insight Journal 2006
12. Parker SG, Weinstein DW, Johnson CR: The SCIRun Computational Steering Software System. Birkhauser Boston Inc., Cambridge, MA, USA, 1997
13. Robb RA, Hanson DP: ANALYZE: a software system for biomedical image analysis. Proceedings of the First Conference on Visualization in Biomedical Computing pp. 507–518, 1990
14. Chu V, Hamarneh G: MATLAB-ITK interface for medical image filtering, segmentation, and registration. SPIE Medical Imaging 2006: Image Processing 6144(1):61443T, 2006
15. Gering DT, Nabavi A, Kikinis R, Hata N, O'Donnell LJ, Grimson WEL, Jolesz FA, Black PM, Wells WM: An integrated visualization system for surgical planning and guidance using image fusion and an open MR. J Magn Reson Imaging 13(6):967–975, 2001
16. Gering DT, Nabavi A, Kikinis R, Grimson WEL, Hata N, Everett P, Jolesz F, Wells WM: An integrated visualization system for surgical planning and guidance using image fusion and interventional imaging. In Medical Image Computing and Computer-Assisted Intervention. LNCS 1679:809–819, 1999
17. Prior FW, Erickson BJ, Tarbox L: Open source software projects of the caBIG™ in vivo imaging workspace software special interest group. Journal of Digital Imaging pp. 94–100, 2007
18. Paladini G, Azar FS: An extensible imaging platform for optical imaging applications. SPIE Medical Imaging 2009: Multimodal Biomedical Imaging IV 7171(1), 2009
19. Kitware Inc: VolView 3.2 User Manual. Clifton Park, New York, 2009
20. Martin K, Hoffman B: Mastering CMake. Kitware, Inc., USA, 2003