SimITK: Model Driven Engineering for Medical Imaging

by

Melissa Trezise

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen’s University
Kingston, Ontario, Canada
August 2013

Copyright © Melissa Trezise, 2013
Abstract

The Insight Segmentation and Registration Toolkit (ITK) is a highly utilized open source medical imaging library. Written in C++, ITK provides chiefly the functionality to register, segment, and filter medical images. Although extremely powerful, ITK has a very steep learning curve for users with little or no background in programming. It was for this reason that SimITK was developed. SimITK wraps ITK into the model driven engineering environment Simulink, a part of the Matlab development suite. The first released version of SimITK was a proof of concept, and demonstrated that ITK could be wrapped successfully in Simulink. Very few segmentation and registration functions were available and the system was based on ITK version 3 with a semi-automatic wrapping procedure.

In this thesis a new version of SimITK is presented that includes thirty-seven image filter, twelve optimizer, and nineteen transform classes from ITK version 4 which are successfully wrapped and tested. These classes were chosen to represent a broad range of usability (in the case of the filters) and to allow for greater flexibility when creating registration pipelines by having more options for optimizers, transforms, and metrics. Many usability improvements were also implemented for the registration pipeline, including providing the user with the metric value while executing a registration model and allowing the output image size to be specified for certain filters. In order
for SimITK to transition to a usable research tool, several usability improvements were needed. These included transitioning from wrapping ITK version 3 to ITK version 4, fully automating the wrapping process, and usability modifications to the registration pipeline including a metric value output.

These implementations of an automated wrapping procedure for ITK version 4, and improved usability of the registration pipeline have propelled SimITK on a path towards a usable research tool. The author will be creating a release of these changes, updating installation documentation, and updating tutorials which are available at www.SimITKVTK.com
Statement of Co-Authorship

The work presented in this thesis was accomplished under the supervision of Dr. Parvin Mousavi, Dr. Purang Abolmaesumi, Dr. James Cordy, and Dr. David Gobbi who provided guidelines and development recommendations as well as corrections to the manuscript.

SimITK existed previously as a proof-of-concept tool as presented by Andrew W. L. Dickinson in his Master’s Thesis titled “SimITK: Visual Programming of the ITK Image Processing Library within Simulink” [8]. Dr. Gobbi contributed the code for automatically creating simplified XML files for the SimITK templating process. Otherwise, the material presented in this thesis is the original work of the author.
Acknowledgments

First of all I would like to thank my supervisors: Dr. Parvin Mousavi, Dr. Purang Abolmaesumi, Dr. James Cordy, and Dr. David Gobbi. Each of you provided me and this project with something unique and it was the combination of these factors that allowed me to reach the success I did. Special thanks to David for your guidance throughout the development process and for helping me to overcome the learning curve that was SimITK.

To the members of the Med-i Lab: Layan, Farhad, Amir, and most especially to Simrin: when I started this project I never expected to leave with a best friend. I am so thankful that we were able to go on this journey together.

To all the friends who have supported me along the way, thank you for the many laughs and great memories (you know who you are).

Lastly, and most importantly, I dedicate this work to my parents: Bill and Margie. Without your constant support and encouragement I would not have accomplished what I did during my Master’s or have so far in life. Thank you and love you.

Melissa Trezise, July 2013

“Happiness can be found, even in the darkest of times, if one only remembers to turn on the light.” -Albus Dumbledore
Glossary

**2D** 2 Dimensions/Dimensional.

**3D** 3 Dimensions/Dimensional.

**CT** Computed Tomography.

**GUI** Graphical User Interface.

**ITK** The Insight Segmentation and Registration Toolkit.

**MRI** Magnetic Resonance Imaging.

**Registration** A process for determining the correspondence of features between images.

**Segmentation** The division of an image into its main regions or objects.

**Simulink** A block diagram environment for model driven engineering in Matlab.

**Templating** The process whereby code is automatically generated by expanding keywords in a set of base files.

**XML** Extensible Markup Language.
# Contents

Abstract i  
Statement of Co-Authorship iii  
Acknowledgments iv  
Glossary v  
Contents vi  
List of Figures x  
List of Code Listings xiv  
List of Tables xv  

## Chapter 1: Introduction  
1.1 Motivation ............................................. 1  
1.2 Thesis Objectives and Contributions ..................... 6  
1.3 Thesis Outline ....................................... 7  

## Chapter 2: Background  
2.1 Model Driven Engineering .............................. 9  
2.2 Medical Image Processing ................................ 10  
  2.2.1 Registration ....................................... 11  
  2.2.2 Segmentation ..................................... 14  
2.3 The Insight Segmentation and Registration Toolkit (ITK) .... 14  
2.4 Simulink .............................................. 16  
2.5 SimVTK ................................................. 17  
2.6 SimITK ................................................ 17  
2.7 ITK-Based Image Processing Systems ...................... 19  
  2.7.1 SimpleITK ......................................... 19
4.5.3 SimITK Code Base Improvements ........................................ 67
4.6 Testing ............................................................................. 68
4.7 Summary ........................................................................... 69

Chapter 5: Results .................................................................... 70
5.1 Available Simulink Block Libraries ........................................ 70
5.2 Model Creation .................................................................... 71
  5.2.1 Creating a Simple Filter Model: FlipImageFilter ............... 71
  5.2.2 A More Complex Model: Image Registration ................. 76
5.3 ITK Version 4 Block Creation ............................................... 79
  5.3.1 ITK Version 4 Registration Models ............................... 80
    5.3.1.1 Registration Expansion 1: Optimizer Blocks ............. 80
    5.3.1.2 Registration Substitution 2: Transform Blocks ......... 82
    5.3.1.3 Registration Substitution 3: Metric ...................... 83
    5.3.1.4 A Note about 2D Registration ............................. 84
  5.3.2 ITK Version 4 Image Filtering Models ......................... 84
    5.3.2.1 Smoothing .......................................................... 87
    5.3.2.2 Thresholding ....................................................... 87
    5.3.2.3 Geometry ........................................................... 89
    5.3.2.4 Edge Detection .................................................... 89
    5.3.2.5 Region ............................................................... 90
5.4 Usability Improvements ..................................................... 91
  5.4.1 Metric Block Output ..................................................... 91
  5.4.2 Resample Image Filter Modified Mask Parameters ........... 93
5.5 Summary ........................................................................... 95

Chapter 6: Conclusions and Future Work .................................. 97
6.1 Summary of Conclusions ................................................... 97
6.2 Future Work ...................................................................... 98

Bibliography ........................................................................ 100

Appendix A: SimITK File Examples and Templates ..................... 106
  A.1 ITK Process Examples ..................................................... 106
  A.2 ITK Version 4 Conversion Examples ............................... 109

Appendix B: Transitioning to ITK Version 4:Models and Testing .... 115
  B.1 Creating Models in Simulink with SimITK ....................... 115
  B.2 2D Registration Testing ................................................. 118
  B.3 Filter Testing ............................................................... 121
    B.3.1 Smoothing ............................................................. 121
B.3.2 Thresholding ............................................... 122
B.3.3 Edge Detection ............................................... 123
## List of Figures

1.1 SimITK image reader Simulink block ................................. 2
1.2 SimITK 3D registration model created by the author ............ 4
1.3 The SimITK development process .................................... 5

3.1 The Simulink application windows .................................. 29
3.2 The SimITK development process .................................... 29
3.3 An annotated excerpt from the ITK header file for FlipImageFilter . 31
3.4 An annotated excerpt from the XML file for FlipImageFilter ...... 32
3.5 Simulink function blocks ............................................. 35
3.6 SimITK interface with process model ............................... 36

4.1 SimITK development process overview .............................. 39
4.2 SimITK templating overview ......................................... 40
4.3 SimITK templating process expanded ............................... 42
4.4 ThresholdImageFilter mask: un-modified and modified ......... 50

5.1 Matlab toolbar with Simulink library icon outlined in red ....... 71
5.2 Simulink UC2D block library ....................................... 72
5.3 Result of adding a block to an untitled model .................... 73
5.4 Input, filter, and writer blocks after being added to an untitled model 73
5.5 Input, filter, and writer blocks with connections. 74
5.6 Mask for the input block to set the input file name. 74
5.7 FlipImageFilter mask where flip image filter properties are set. 75
5.8 The final model with play button to run the simulation circled in red. 75
5.9 Simple model example input and output: (a) the input; (b) the output. 76
5.10 SimITK 3D registration model. 78
5.11 3D registration: (a) fixed image (CT) for 3D image registration (visualized in red); (b) moving image (MRI) in registered position; (c) overlay of CT and MRI to show registration result. 79
5.12 The SimITK optimizer library. 81
5.13 SimITK registration model with Powell optimizer outlined. 81
5.14 The SimITK 3D transform library. 82
5.15 SimITK registration model with affine transform outlined. 83
5.16 SimITK registration model: normalized correlation metric. 84
5.17 Images used for testing ITK version 4 filters; (a) brain proton density slice (UC2D); (b) CT slice of a head (FL2D). 86
5.18 Effect of smoothing filters on brain proton density slice (Figure 5.17(a));
(a) effect of MeanImageFilter; (b) effect of DiscreteGaussianImageFilter. 87
5.19 Binary version of brain proton density slice (Figure 5.19). 88
5.20 Effect of thresholding filters on brain proton density slice; (a) effect of BinaryContourImageFilter on Figure 5.19; (b) effect of OtsuMultipleThresholdsImageFilter on Figure 5.17(a). 89
5.21 Effect of edge detection filters on CT head slice (Figure 5.17(b)); (a) effect of CannyEdgeDetectionImageFilter; (b) effect of DerivativeImageFilter. ........................................ 90

5.22 Effect of region of interest filters on brain proton density slice (Figure 5.17(a); (a) effect of CropImageFilter; (b) effect of ConstantPadImageFilter; (c) effect of MirrorPadImageFilter. ......... 91

5.23 Old SimITK registration model showing available outputs from [9]. . 92

5.24 Registration model showing metric value output and scope. .......... 93

5.25 Registration model showing the ResampleImageFilter with size, scaling, and origin values set. ................................................. 94

5.26 Output image from 3D registration with image information window. . 95

B.1 Base model for 2D image registration. ................................. 118

B.2 2D image registration with amoeba optimizer outlined. ............ 119

B.3 2D image registration with centered similarity 2D transform outlined. 119

B.4 2D image registration with gradient difference metric outlined. ... 120

B.5 Smoothing filters applied to brain proton density slice (Figure 5.17(a));
(a) effect of DiscreteGaussianImageFilter; (b) effect of MeanImageFilter; (c) effect of MedianImageFilter; (d) effect of BilateralImageFilter; (e) effect of BinomialBlueImageFilter; (f) effect of RecursiveGaussianImageFilter. .................. 121
B.6 Thresholding filters applied to applied to Figure 5.19 ((a) and (b)) and applied to Figure 5.17(a) ((c) through (f)); (a) effect of BinaryContourImageFilter; (b) effect of LabelContourImageFilter; (c) effect of OtsuThresholdImageFilter; (d) effect of OtsuMultipleThresholdsImageFilter; (e) effect of ThresholdImageFilter; (f) effect of BinaryThresholdImageFilter.

B.7 Edge Detection filters applied to CT head slice (Figure 5.17(b)); (a) effect of DerivativeImageFilter; (b) effect of LaplacianImageFilter; (c) effect of LaplacianSharpeningImageFilter; (d) effect of SobelEdgeDetectionImageFilter; (e) effect of CannyEdgeDetectionImageFilter.
List of Code Listings

1.1 ITK C++ code to create an image reader. .................. 2
1.2 Excerpt of ITK C++ code for registration taken from [8]. ....... 3

4.5 Example XML tags from ITK version 3 XML files. ............... 52
4.6 Example XML tags from ITK version 4 XML files. ............... 53

A.1 Simulink mask file for ThresholdImageFilter. ............ 106
A.2 Exception dump file for ResampleImageFilter class. ....... 109
A.3 Version 3 XML File for CenteredTransformInitializer. ... 110
A.4 Version 4 XML File for CenteredTransformInitializer. ... 113

B.1 FlipImageFilter C++ code from [18]. ....................... 115
List of Tables

4.1 Templating process: responsible Perl script and resulting file extension. 41
4.2 Class type to context name mapping. .......................... 55

5.1 Smoothing classes included in SimITK. ......................... 85
5.2 Thresholding classes included in SimITK. ....................... 85
5.3 Geometry classes included in SimITK. ......................... 85
5.4 Edge detection classes included in SimITK. .................... 85
5.5 Region of interest classes included in SimITK. ............... 86
Chapter 1

Introduction

1.1 Motivation

ITK is a comprehensive image segmentation, registration, and filtering toolkit that is well regarded and widely used in the area of medical image processing. At the same time, many potential users do not attempt to take advantage of this toolkit due to their lack of programming background and the steep learning curve that is associated with ITK.

It is for this reason that SimITK was developed and first presented in 2008 [13]. SimITK is a domain specific language for ITK that takes advantage of the model driven engineering (MDE) capabilities of Simulink. Simulink is a block diagram environment that was created to be used within Matlab’s (Mathworks Inc., Natick, MA) development environment. With the use of Simulink, in SimITK, all written programming code is abstracted away and replaced with a visual model. For each ITK class, there is a specific Simulink block representation. A simple block example is an image reader. As shown in Listing 1.1 at least five lines of code are required to create a reader in ITK. In SimITK, a simple block, as shown in Figure 1.1, can be
1.1. MOTIVATION

Listing 1.1: ITK C++ code to create an image reader.

```cpp
// Define and create CT data reader
typedef short InputPixelType_CT;
typedef itk::Image< InputPixelType_CT, 3 > InputImageType_CT;
typedef itk::ImageFileReader< InputImageType_CT > ReaderType_CT;
ReaderType_CT::Pointer readerCT = ReaderType_CT::New();
readerCT->SetFileName( inputFilenameCT );
```

Figure 1.1: SimITK image reader Simulink block.

used instead to provide the same functionality without the need for coding. Several blocks can be selected and dropped onto a Simulink canvas, and combined to create a model to represent different medical image processing workflows or algorithms.

An example of the process is creating a registration pipeline. Registration is the process of finding the correspondence between two images based on their features. In ITK, code that must be written to do a registration of a CT (Computed Tomography) and an MRI (Magnetic Resonance Imaging), for example, would require extensive programming expertise. Every component of the registration must be initialized correctly in order for the process to succeed. An excerpt of the code to perform a CT registration taken from Dickinson et al. [8] can be seen in Listing 1.2. The equivalent model in SimITK is exclusively done by selecting appropriate blocks and connecting them to create and run a model. There is no need to interact with the code and the workflow is created using model driven engineering. This model can be seen in Figure 1.2.
Listing 1.2: Excerpt of ITK C++ code for registration taken from [8].

```cpp
// Define and create CT data reader
typedef short InputPixelType_CT;
typedef itk::Image< InputPixelType_CT, 3 > InputImageType_CT;
typedef itk::ImageFileReader< InputImageType_CT > ReaderType_CT;
ReaderType_CT::Pointer readerCT = ReaderType_CT::New();
readerCT->SetFileName( inputFilenameCT );

const unsigned int Dimension = 3;
typedef short PixelType;
typedef itk::Image< PixelType, Dimension > FixedImageType;
typedef itk::Image< PixelType, Dimension > MovingImageType;
typedef itk::CenteredEuler3DTransform< double > TransformType;
typedef itk::RegularStepGradientDescentOptimizer OptimizerType;
typedef itk::MattesMutualInformationImageToImageMetric< FixedImageType,
MovingImageType > MetricType;
typedef itk::NearestNeighborInterpolateImageFunction< MovingImageType,
double > InterpolatorType;
typedef itk::ImageRegistrationMethod< FixedImageType, MovingImageType >
RegistrationType;

MetricType::Pointer metric = MetricType::New();
TransformType::Pointer transform = TransformType::New();
OptimizerType::Pointer optimizer = OptimizerType::New();
InterpolatorType::Pointer interpolator = InterpolatorType::New();
RegistrationType::Pointer registration = RegistrationType::New();

metric->SetNumberOfSpatialSamples( 1250000 );
metric->SetNumberOfHistogramBins( 32 );
metric->SetUseAllPixels( 0 );

// Setup registration
registration->SetMetric( metric );
registration->SetOptimizer( optimizer );
registration->SetTransform( transform );
registration->SetInterpolator( interpolator );
registration->SetFixedImage( readerCT->GetOutput() );
registration->SetMovingImage( readerMR->GetOutput() );

// Setup transform initializer

// Set up Optimizer

// run the registration
registration->StartRegistration();
```
Figure 1.2: SimITK 3D registration model created by the author.
In SimITK, Simulink was chosen as the visual programming environment for its MDE capabilities but also because of its affiliation with Matlab. Matlab is widely used for teaching and research and most graduate students, undergraduate students, and researchers are familiar with this tool. Additionally, Simulink is very easy to learn and is seamlessly integrated with Matlab functions. There are many functionalities in Matlab that would complement those in ITK, and therefore, SimITK can enable the integration of ITK and Matlab.

The process to create SimITK is a complex templating procedure. An overview of this process can be seen in Figure 1.3. Each of the blocks represent an artifact that is created to support the process to develop SimITK. The arrows represent source code transformations that must occur to generate these artifacts. The process as shown here is fully automated with a complex templating procedure that occurs to transform the XML files into the glue files that are required to create and use the Simulink function blocks.

In the previous version of SimITK, this process used hand-written XML files that had to be created individually [9]. This made the process of adding new classes or upgrading to a newer version of ITK very difficult. The version of ITK that these XML files were originally created for was version 3 which became a concern when ITK moved to version 4 in 2011. The previous version of SimITK also had many usability concerns including not being able to view the metric value in a registration model.
and not being able to set the output image size for certain filters. Although limited versions of registration and segmentation pipelines were made available, along with several filters, this release was really only a proof-of-concept.

The remainder of this thesis will present usability improvements that were made to SimITK with the goal of propelling it to the forefront of model driven engineering for medical imaging.

1.2 Thesis Objectives and Contributions

The objective of this work was to begin to transition SimITK from a limited proof-of-concept to a usable research tool and to introduce features to improve its usability for registration, segmentation, and filtering; three of the most important aspects of medical imaging. These usability improvements included adding a metric value output which had a different format from the outputs that were previously supported. Also, this transition needed to maintain the currency of SimITK in relation to the ITK project itself while implementing an automatic generation process.

In this thesis, I have:

- Successfully transitioned SimITK from supporting ITK version 3 to ITK version 4, allowing the system to remain current and up-to-date with the ITK project.
- Fully automated the procedure to wrap ITK in Simulink through the integration of new XML files that were developed by Gobbi.
- Successfully added the ability for the user to monitor the metric value in a registration model during registration.
• Solved the complex problem of having a different output image size than the input image for the filters that required this functionality.

• Tested the registration pipeline with all wrapped transform, metric, and optimizer blocks, therefore further improving its usability by generating a test suite from a base registration model.

• Tested all filters that were to be included in SimITK to ensure their functionality and usability by manually creating a test model for each filter.

1.3 Thesis Outline

This thesis is organized as follows:

Chapter 2, Background: provides a brief description of model driven engineering, medical image processing, the components of SimITK, and similar ITK-based systems.

Chapter 3, Overview: provides an overview of the process needed to create SimITK.

Chapter 4, Methods: expands upon the process to create SimITK and its components, and details the changes made to improve the usability of SimITK, including an automatic wrapping procedure, transitioning to ITK version 4, and modifications to the registration pipeline.

Chapter 5, Results: examples of the usage of SimITK are provided with a focus on the new filters that have been implemented with ITK version 4 and the improved registration pipeline.

Chapter 6, Conclusions and Future Work: presents a summary of the key
ideas that have been discussed in this document and suggests future improvements to SimITK.
Chapter 2

Background

The chapter outlines the fundamental knowledge that is required to understand and use SimITK, including its development style, basis, and relevant technology.

In this chapter, model driven engineering including domain specific languages is introduced and discussed, medical image processing terminology with respect to SimITK applications is introduced, ITK and Simulink —the components of SimITK—are presented and their main properties are outlined, a survey of similar ITK systems that wrap ITK in some way to increase its usability is presented and SimITK is discussed within this content including its differences with other systems.

2.1 Model Driven Engineering

Model Driven Engineering (MDE) technologies were developed to help alleviate the complexity of programming platforms and express domain specific concepts more adequately [11] [22] [32]. In the past models were mainly used to document software systems and the software cycle was thought of as a chain of objects. However, thinking of the software cycle as a chain of models is not that different from thinking of it as a chain of objects, thus MDE was introduced [1]. MDE combines both domain-specific
modeling languages and transformation engines [32].

Domain specific languages (DSLs) are languages that are designed specifically to express concepts of a certain domain, such as medical image processing. They are used instead of combining other general purpose language constructs to express these domain specific concepts, for example in a programming language. These domain specific languages have evolved from this definition and can now include libraries or frameworks that are built on top of general purpose languages to support a specific domain [33]. Domain specific modeling languages describe a domain using metamodels, which describe the relationships between concepts in a domain along with the language and processes to create the models [32].

Transformation engines are used to analyze parts of a model and then generate different artifacts that can include simulation inputs.

SimITK can be thought of as a domain specific language for expressing image processing workflows. SimITK automatically generates code to execute image processing models in Matlab’s Simulink environment.

2.2 Medical Image Processing

ITK is a comprehensive library of C++ classes that was designed for image processing, registration and segmentation [17]. SimITK has focused on three specific areas of ITK: segmentation, registration, and filtering. Segmentation and registration will be briefly introduced in the next sections. The reader is encouraged to refer to [15] and [25] for detailed descriptions of these topics.
2.2.1 Registration

Registration “is a process for determining the correspondence of features between images” [7]. Generally these images have been taken at different time points or using different imaging modalities; as such, a correspondence between the images needs to be determined [7]. Registration has wide spread applications in the medical imaging field, from allowing structural and functional images to be viewed at the same time, to allowing the growth of a tumor to be tracked in time [7]. There are two main types of registration: rigid and non-rigid. Rigid registration allows images to be rotated and translated to find a correspondence between them. Non-rigid registration allows for deformations of images in addition to rotation and translation [7].

The general approach for registration involves the input images; one image is considered the fixed image and the other is the moving image. The registration output is a transformation that represents the optimal alignment of the moving image to match the fixed image [34]. A transform initializer is often used to find an initial position to begin the alignment. The transform parameters are iteratively modified from an initial position, through an optimization process. Once a satisfactory transformation has been found, the moving image may be resampled to match the size, spacing, and origin of the fixed image.

Centered Transform Initializer

As described previously, the transform initializer begins the process of registration. Its objective is to simplify the transformation process. In SimITK, a centered transform initializer is used. This transform initializer computes the centers of the fixed and moving images. The center of the fixed image is used as the rotational center of the transform and the vector from the fixed image center to the moving
image center is used as the first translation. This simplifies the process of overlaying the images and determining the final transform [17].

**Transform**

In the registration process, the transform defines how one image is deformed, translated, and rotated to match the other [7].

**Optimizer**

The optimizer is responsible for adjusting the transform to achieve the greatest image similarity during registration. A successful optimizer is able to do this quickly and reliably, without ending up in a local minima. A local minima can occur when an optimizer believes it has chosen the best set of parameters, but it is only a good match and not the best one. This is a particular problem with non-rigid registration because the more degrees of freedom the registration has the more parameters that are required to describe it [7]. For each degree of freedom that is added to the registration process, an additional parameter must be optimized. In the case of 3D rigid registration there are six degrees of freedom: three for rotation and three for translation in the x, y, and z axes and therefore, six parameters are required.
2.2. MEDICAL IMAGE PROCESSING

Metric

The metric or similarity measure is a value that describes how well two images match [7]. This value provides information on the success of the registration. A popular subset of similarity measures deals exclusively with the intensity-based similarity between images. This group includes those based on mean squared error, gradient difference, and mutual information. The mutual information metric is particularly useful since it can be used for mono-modality and dual-modality registration. The mutual information metric assumes a probabilistic relationship between intensities which will be maximized when the images are optimally registered [7], allowing images from different modalities to be successfully registered.

Sampling

Sampling occurs at two places in image registration. In ITK’s image registration pipeline, sampling occurs as part of the comparison of the two input images (performed by the metric), and at the end of the registration. The first phase, when sampling occurs in the metric, involves samples being taken from each input image and then compared using interpolation as necessary. Interpolation involves taking a weighted average of the surrounding points of the location being sampled. This interpolation allows the metric to compare the two images.

The second sampling phase occurs at the end of registration when the whole moving image must be resampled to the pixel spacing of the fixed image.
2.2.2 Segmentation

Image segmentation is a process that divides an image into its main regions or objects [15]. There are two main types of image segmentation: low-level and model-based. Low-level segmentation uses image features to classify voxel types or identify regions in an image. These include region growing, thresholding, and classifiers. Model based segmentation attempts to fit a certain model to the image data. These include level sets, deformable models, and atlases [25]. Hybrid techniques also exist [25].

2.3 The Insight Segmentation and Registration Toolkit (ITK)

ITK is a large open source library of classes [17] that was initially developed in 1999 through funding from the US National Library of Medicine and the National Institutes of Health. The goal of ITK was to develop a toolkit to support image processing, registration, and segmentation. The main language of choice was C++, allowing the use of object oriented methodologies and templating. Today, ITK is comprised of approximately 25 groups of modules\(^1\), and over 2.2 million lines of code\(^2\).

ITK is organized into several groups of modules, filters, and objects. These groups contain classes that each have a specific functionality. The groups of interest to the SimITK project at this time are core, numerics, registration, segmentation, and filtering. A description of these groups and their main functionalities are outlined below:

- Core contains base classes that are used by most of the other groups, including

---

\(^1\)The structural breakdown of ITK can be found here: http://www.itk.org/Doxygen/html-/modules.html

\(^2\)Statistics for ITK are available here: http://itk.org/ITK/project/statistics.html
2.3. THE INSIGHT SEGMENTATION AND REGISTRATION TOOLKIT (ITK)

the transform and interpolator classes.

- **Numerics** contains classes that are basic numerical tools and algorithms, including the optimizer classes.

- **Registration** contains the main registration classes along with the metrics and transform initializers.

- **Segmentation** contains the classes for performing image segmentation.

- **Filtering** contains filters that modify images in an ITK pipeline including the image feature, image grid, image label, smoothing, and thresholding filter classes.

Working with these classes in ITK follows a general structure of selecting a particular class, attaching input and output images to that class, and running the program. Most workflows in ITK follow a pipeline procedure where one class feeds information into the next. In a simple example, an input image would be fed into a filter which would perform some action on the input image and feed it into an output image. This pipeline gets more complicated when working with registration for example, since it requires multiple classes to be initialized and fed to each other in the correct order. This is one of the qualities of ITK that makes it particularly difficult to work with.

Another difficulty with ITK is its widespread use of typedefs. These typedefs are specific ITK types that are derived from C++ primitives. The user must be aware of the nature of these types (of which there are 1000s) in order to successfully work with ITK. Along with these specific typedefs, ITK is also a templated toolkit which means that each class can take a variety of types and dimensionalities. This is useful feature for code reuse but makes ITK extremely hard to work with for someone who does not have a strong programming background.
The complexity of ITK has led to the requirement for simpler systems to be developed. These systems generally try to abstract away some of the more complex aspects of ITK including the need to initialize all classes in a specific way, understand the ITK `typedefs, and deal with templates. The pipeline nature of ITK lends itself well to integration with Simulink as was discussed in Chapter 1.

2.4 Simulink

Simulink\(^3\) is a model driven engineering environment that was developed by MathWorks and is integrated within Matlab. Simulink diagrams are pipelines in nature and are very easy to create. Blocks representing different functions can be dragged from the block library to a blank canvas and then connected together to perform a specific function. There are a large number of blocks that are included with Simulink including block libraries for signal processing, mathematical functions, and neural networks. Custom made blocks and block libraries can also be created and included. Once a pipeline has been created, the model can be simulated, and the user can use display blocks to view information about the model. These information blocks can include sinks that output values, scopes that graph a value output, or output blocks to write the result to a file or the Matlab workspace [21].

Simulink, which easily allows MDE to occur, is a great choice for modeling a complex system because any written code is abstracted away and replaced by graphical elements in the form of blocks. The flow of information is modeled by the arrow connections between the blocks allowing the user to connect the parts of a system in a logical order.

\(^3\)http://www.mathworks.com/products/simulink/
2.5 SimVTK

SimVTK\(^4\) is an open-source block library that wraps the Visualization Toolkit\(^5\) (VTK) into Simulink. VTK is a toolkit for image processing, and visualization developed by Kitware Inc. SimVTK is considered a sister project to SimITK; they were both developed at the same time and use similar principles \([13]\). SimVTK does have some significant differences from SimITK however. All VTK classes are wrapped in SimVTK as opposed to only the subset that is available in SimITK \([14]\). This is because of two fundamental differences between the two toolkits. As discussed in Section 2.3, ITK is templated over datatypes and dimensionalities; whereas, VTK is not. This has made the wrapping procedure for VTK much easier, allowing the entire toolkit to be captured in SimVTK. Also ITK tends to use special \texttt{typedefs} to represent C++ primitives as was also discussed in Section 2.3. VTK, in comparison, uses fundamental types such as \texttt{float}, \texttt{int}, and arrays making its architecture much less complex \([13]\). SimVTK also has the additional feature of being able to view documentation about a class in its parameter dialog box \([14]\), a useful feature. SimVTK is a complete and usable representation of VTK. It can also be used in conjunction with the SimITK blocks to allow for visualization in the SimITK pipelines.

2.6 SimITK

SimITK\(^6\) is an open-source block library that wraps ITK into Simulink. As was mentioned in Section 2.5 it is the sister project to SimVTK which began development at the same time \([13]\). When originally presented, SimITK did not support options

\(^4\)http://www.simitkvtk.com/
\(^5\)http://www.vtk.org/
\(^6\)http://www.simitkvtk.com/
for the metrics, transforms, and optimizers and instead only offered a registration
helper class block [13]. Since then, significant advances were made in the process of
creating the SimITK blocks but there were still many limitations [9].

Dickinson et al. presented a semi-automatic wrapping procedure to create Sim-
ITK [9]. Much of the core of SimITK was established at this point and that structure
has not changed since then. One main drawback of that procedure however, was
the use of handwritten XML files to represent each class. Instead of being able to
automatically generate XML files (e.g. similar to SimVTK), these files had to be
individually created [9].

This version of SimITK wrapped ITK version 3.18 [9]. Since then, ITK version 4
has been released and is widely adopted; as such, SimITK needed to be upgraded to
maintain currency with ITK.

Several examples of workflows that could be created in SimITK were presented
by Dickinson et al. [9]. These highlighted the ability to view the optimizer value and
the transform parameters but the user was unable to view the metric value during
registration. The metric is an important part of the registration pipeline and its
monitoring is a useful feature.

SimITK as presented by Dickinson et al. [9] was a prototype of how ITK could
be wrapped into Simulink. It consisted of a small useful subset of classes but did not
provide a full range of functionality for these classes and was not created by a fully
automated process.
2.7 ITK-Based Image Processing Systems

Many ITK-based systems exist that have attempted to alleviate the learning curve associated with ITK by abstracting away some of its more complex aspects. A survey of some of these systems will be presented and a comparison of these systems to SimITK will be provided.

2.7.1 SimpleITK

SimpleITK\(^7\) is created and maintained by Kitware Inc. It is a library of simplified ITK classes that can be used to facilitate rapid ITK development. It is available in C++ as well as several other languages (Python, Java) and was designed to expose the classes that are available in ITK while minimizing the learning curve. In these available programming languages (C++, Python, Java, Ruby) the use of ITK has been simplified. Templates and types are abstracted away making it easier to work with SimpleITK than with native ITK [3]. As of September 2012 there were 255 ITK image filters available in SimpleITK [19]. This group of filters does not contain any classes for registration as the focus has been on wrapping the image filters themselves and not the registration classes [20].

SimITK in comparison to SimpleITK incorporates a MDE visual environment through the use of Simulink. This focus on MDE allows all programming code to be abstracted away in SimITK and replaced with a visual representation. With SimpleITK the user still has to program and there is no visual programming option available.

\(^7\)http://www.simpleitk.org/
2.7. ITK-BASED IMAGE PROCESSING SYSTEMS

2.7.2 MatITK

MatITK\(^8\) is a set of mex files that provide an interface between ITK and Matlab [6]. These mex files can be accessed and run within the Matlab command line. One of the main functionalities of this interface is the translation of image data between ITK and Matlab in memory to allow for greater efficiency. MatITK also provides some error handling [6]. Currently, MatITK supports ITK version 2.4 [4] and includes approximately forty ITK classes with only two being from the registration group [5]. MatITK also only supports 2D double image types. MatITK has built in functionality for a small subset of ITK classes, however, there is the ability for more advanced users to implement their own ITK methods as outlined in [6].

Although both MatITK and SimITK make use of the Matlab environment, MatITK only uses the text command interface and does not make use of the graphical environment Simulink. MatITK also only supports a small subset of ITK version 2 classes with limited registration classes available for 2D double image types. SimITK supports a wide range of classes with a large number from registration from ITK version 4, along with supporting four datatypes, and both 2D and 3D images.

2.7.3 The Image-Guided Surgery Toolkit (IGSTK)

IGSTK\(^9\) which has been developed by Kitware Inc. and Georgetown University is an open-source C++ framework for application development. It is a toolkit that provides high level components that can be used to create image-guided surgery applications. There is also an application front end that can be used if the user does not want to create their own. Its focus is on allowing safety critical (able to prevent patient

\(^8\)http://matitk.cs.sfu.ca/
\(^9\)http://www.igstk.org/
harm) and robust applications to be developed using IGSTK as the backbone [10].
To support the development of safety critical applications, the underlying structure of
IGSTK follows component-based and state-machine architectures. Each component
has a certain set of functions that it is allowed to perform and these are regulated
by state machines. The user does not have to know about this architecture since
they are only presented with a high level component view. These components are
implemented in C++ and many are based on ITK and VTK. As with many high level
applications using ITK, the user does not see the underlying ITK classes. This allows
researchers to focus on creating image-guided applications to solve specific problems
instead of having to learn a new toolkit along with ensuring that their systems are
safe and robust [10].

In comparison to SimITK, users of IGSTK must either create their own software
application to use the toolkit or must learn the provided front-end. SimITK, on the
other hand, makes use of the Matlab environment which is ubiquitous among research
institutions and therefore would not be a new system to learn or to develop.

2.7.4 ITK-SNAP

ITK-SNAP\textsuperscript{10} is a standalone software application that was developed specifically for
segmentation and is now considered part of the ITK development initiative. The
decision to focus on segmentation was done for several reasons: to avoid a steep
learning curve by minimizing the number of features that a user would have to learn,
and to provide an alternative to manual segmentation [37]. ITK-SNAP provides
many benefits for users including automatic segmentation, manual outlining, quality
control measures, and an intuitive GUI [36]. Tools include being able to view and

\textsuperscript{10}http://www.itksnap.org/pmwiki/pmwiki.php
navigate 3D images, manually label regions, combine multiple segmentation results, and perform postprocessing in 2D or 3D. ITK-SNAP, as the name suggests, is built on top of ITK and takes advantage of the ITK image formats as well as many of its classes \cite{37}. It also continues to be an active software project with version 2.4 having been released in November 2012.

With its focus on segmentation, ITK-SNAP offers limited functionality in comparison to the large number of classes that are available in ITK. ITK-SNAP is not a MDE environment and therefore, has very limited flexibility in terms of the ways that workflows can be created. SimITK, in comparison, tries to provide a broad range of useful classes to allow the user to perform different tasks in the same flexible software platform.

### 2.7.5 VolView

VolView\textsuperscript{11} is an end-user software application developed by Kitware Inc. for volume visualization. It is designed for the exploration of 3D volumes and does not support 2D images \cite{2}. At the time of publication, thirty-seven ITK and VTK filters were available in a graphical manner through menus or GUI panels in the VolView application \cite{2}. Processing in VolView is done interactively through the GUI interface. After the output from one filter is calculated, the next filter can be selected and applied until the final output has been achieved \cite{2}. Along with the base functionality that is available, VolView also includes a "plug-in" architecture. This allows users to create plug-ins for added functionality. Popa et al. present the use of the plug-in architecture where they created algorithms for volume measurement and comparison \cite{28}. These algorithms were then used with the included ITK functionalities \cite{28}. VolView is

\textsuperscript{11}http://www.kitware.com/opensource/volview.html
most useful for creating non-repetitive end-user workflows that are focused on volume applications [2].

Although VolView offers very good functionality for volume visualization and for allowing the creation of plug-ins, it has limited its usability by only supporting 3D images. SimITK supports 2D and 3D images, allowing the user a greater range of possible uses. SimITK is also incorporated in the ubiquitous Matlab environment whereas VolView requires the user to learn a new application.

2.7.6 MeVisLab

MeVisLab\textsuperscript{12} is a standalone software application for visual programming that combines ITK and VTK with the native MeVis library [31]. MeVisLab combines ITK and VTK with a visual programming environment that allows for rapid prototype development and algorithm testing. Modules are automatically created from ITK code through the use of XML descriptions for each class. These modules are named in the same way as the ITK class they represent to distinguish them from the MeVis library classes [30]. These modules can then be combined in the visual programming environment to create workflows to perform specific tasks. Modules can also be created by the user [30] and scripting in JavaScript or Python can be included in the workflow as well [16]. The addition of a scripting code in a workflow can allow the workflow to modify its parameters based on user actions, or to save and calculate specified values [16].

The standalone platform MeVisLab provides a wide range of functionality in the area of medical image analysis. By integrating ITK and VTK with the MeVis visualization library users can create and execute workflows to perform medical imaging

\textsuperscript{12}http://www.mevislab.de/
tasks. MeVisLab also allows the user to create their own modules and add extra functionality to their workflows through scripting. The development of MeVisLab is active. Most recently, version 2.4 was released in February 2013 and included over 300 modules based on ITK.

Although MeVisLab is a wide reaching, well developed, software application, it does have two potential drawbacks. Firstly, since it is a standalone application the user must learn and interact with an interface they have never used before. As was mentioned before, SimITK uses the ubiquitous Matlab environment to provide the user a familiar environment to work in. Secondly, MeVisLab has expanded greatly to include many functionalities beyond ITK and VTK. This again poses the problem that a user has to learn and become familiar with the functionality of a new tool. SimITK has chosen to remain small and focused by wrapping the most useful ITK classes in Matlab’s Simulink environment.

2.7.7 The Medical Imaging Interaction Toolkit (MITK)

MITK\textsuperscript{13} was originally presented as a toolkit to act as an extension of ITK and VTK. The goal of this toolkit was to combine ITK and VTK with additional features that are required for interactive medical image analysis \cite{35}. These added features include multiple consistent views of the same data, an undo function, and interactions that create and modify data. MITK was developed very similarly to ITK, as it reused the pipeline architecture, adopted the ITK software process (CMake, doxygen, etc), and was written in C++ \cite{35}. The use of ITK attributes in combination with the additional benefits presented by MITK provided a way to develop interactive medical imaging applications \cite{35}.

\textsuperscript{13}http://www.mitk.org/
Since 2003, when MITK was first created [35], MITK has been used as a toolkit and has been integrated into various frameworks for different applications. One such framework was developed for segmentation, and was integrated into an application called InteractiveSegmentation that is used for image segmentation [23]. MITK developers have also released their own application built on MITK called the MITK Workbench which allows the user to customize their own layout, views, and screens [24].

As with IGSTK, users of MITK must either create their own application that uses this toolkit or learn the provided application to access the underlying functionality. Once again SimITK strives to lessen this learning curve by being integrated with the widely used Matlab environment, allowing the user to work in a familiar environment.

2.7.8 Slicer

Slicer\textsuperscript{14} is a software package for medical image computing and visualization. It is built on ITK and VTK amongst other software components with an added application layer that makes these components usable for non programmers [26]. Slicer is built primarily on VTK but has increasingly moved to using ITK for implementing new algorithms [27]. Although Slicer’s foundation is built on VTK and ITK, it has expanded to support a wide range of clinical applications because of its open-source, community-development model [26]. These applications include image guided surgery, brain mapping, robotics, and virtual colonoscopy amongst others.

SimITK when compared to Slicer is a much smaller application that does not have widespread application. SimITK does offer a focus on making ITK classes more usable instead of building a system on top of them. If the user is looking to use native ITK classes in a quick and easy way then SimITK is a great choice.

\textsuperscript{14}http://www.slicer.org/
2.8. SUMMARY

2.7.9 Summary of ITK-Based Image Processing Systems

All surveyed toolkits and applications accomplish many of the goals that have been proposed for SimITK. Each offers a simplified environment for ITK programming. Some use a wrapped version of ITK to allow easier C++ or other language programming in the form of a toolkit such as SimpleITK, MatITK, IGSTK, and MITK. Others offer a software application with the ability to program visually through pipelines or GUI commands such as ITK-SNAP, VolView, Slicer, and MeVisLab. Many offer a broad range of ITK functionality (amongst other functionalities) while others focus on specific aspects such as segmentation in the case of ITK-SNAP.

The main difference between all of these systems and SimITK is their choice of platform. With the exception of MatITK, each of these requires the user to learn a new application or toolkit, none makes use of a popular environment that is widely used for teaching and research. For a novice user, learning a new toolkit can often be time consuming and frustrating since they will not know where to start or how to do complex procedures. It is the integration with Simulink within Matlab that sets SimITK apart. Users of Matlab will very easily be able to use and access Simulink. SimITK also offers a wide range of functions without overwhelming the user with too many added functionalities as some of the larger systems would.

2.8 Summary

In this chapter the key concepts related to SimITK were presented. Model driven engineering was explained, useful terms related to medical image processing were outlined, and the background on SimITK was presented. A review of current ITK-based image processing systems was also included, along with information about how
these systems differ from the proposed solution. In the next chapter an overview of how SimITK works will be provided.
Chapter 3

Overview

The previous chapter introduced the concepts related to SimITK, including Model Driven Engineering (MDE), medical image processing terminology, and the background of the SimITK system. This chapter provides an overview of SimITK. It sets the foundation for discussing the details of SimITK and its components for the following chapters.

3.1 Development of SimITK

As mentioned before, SimITK is a model-driven approach for creating medical image processing pipelines in Simulink. The Simulink interface can be seen in Figure 3.1. This figure shows the Matlab application with the Simulink block library and a SimITK process model.

The process to create SimITK is outlined in Figure 3.2 with the arrows representing source code transformations which transform an input artifact into an output artifact. A description of each step will be given in the sections that follow.
3.1. DEVELOPMENT OF SIMITK

Figure 3.1: The Simulink application windows.

Figure 3.2: The SimITK development process.
3.1.1 ITK Library Header Files to XML Files

The first step to create SimITK is to capture the ITK header files in XML files to create an easy-to-parse representation of each ITK class. These XML files include information about the inputs, outputs, properties, and types of the class that is gathered from the header file. An example of an ITK header file is shown in Listing 3.3 and the XML file for the same class is shown in Listing 3.4 with the code sections representing inputs, outputs, properties, and types highlighted.
Figure 3.3: An annotated excerpt from the ITK header file for FlipImageFilter.
Figure 3.4: An annotated excerpt from the XML file for FlipImageFilter.
3.1. DEVELOPMENT OF SIMITK

3.1.2 XML Files to Matlab Glue Files

The next transformation that occurs is to create a set of Matlab glue files from the XML files. These glue files are necessary to create the interface between Matlab (in which Simulink is implemented) and ITK. These files include class header files, Simulink system function files, block property files, and parameter dialog files. The class header files and Simulink system function files are needed to create the back end of the block that will communicate between ITK and Simulink. The block property files and parameter dialog files are required to create and change the appearance of the block. Excerpts from these files can be seen in Listings 3.1, 3.2, 3.3, and 3.4.

Listing 3.1: An excerpt from the class header file for FlipImageFilter.

```c
// mutators
void SetFlipAxes(double value) {
  m_FlipAxes = value;
  FlipAxesIsSet = 1;
}

void SetFlipAboutOrigin(bool value) {
  m_FlipAboutOrigin = value;
  FlipAboutOriginIsSet = 1;
}
```

Listing 3.2: An excerpt from the Simulink system function file for FlipImageFilter.

```c
// Gets the value from the parameters.
double FlipAxes=static_cast<double>(mxGetPr(ssGetSFcnParam(S,2*0 + 1))[0]);
bool FlipAboutOrigin=static_cast<bool>(mxGetPr(ssGetSFcnParam(S,2*1 + 1))[0]);
```
### 3.1. Development of SimiTK

#### Listing 3.3: An excerpt from the block property file for FlipImageFilter.

```matlab
function FlipAxesInputIndicatorCallback(block)

vals = get_param(block,'MaskValues');
vis = get_param(block,'MaskVisibilities');
if strcmp(vals{1},'As Parameter'),
    vis(2) = {'on'};
    set_param(gcb,'MaskVisibilities',vis),
else
    vis(2) = {'off'};
    set_param(gcb,'MaskVisibilities',vis),
end
%%to update the blocks number of ports
set_param(block, 'MaskValues', vals);
SetPortLabels(block)

function FlipAxesCallback(block)
```

#### Listing 3.4: An excerpt from the parameter dialog file for FlipImageFilter.

```plaintext
Block {
    BlockType     "S-Function"
    Name          "itkFlipImageFilterFL2D"
    Ports         [2, 2]
    BackgroundColor    "[1, 1, 1]"
    ...
}
```

### 3.1.3 Matlab Glue Files to Simulink Function Blocks

The glue files that were shown in the previous section are then used to create Simulink function blocks. Each of these blocks represents a different ITK class and can have inputs, outputs, and properties that can be modified. These blocks are organized in groups called block libraries based on their function. Block examples can be seen in Figure 3.5.
3.1. DEVELOPMENT OF SIMITK

3.1.4 Simulink Function Blocks to SimITK

Once the function blocks are available, SimITK process models can be created. Blocks can be joined together in the SimITK workspace within Simulink to create pipelines to perform medical imaging tasks. An example pipeline can be seen in Figure 3.6.
3.2 Summary

In this chapter an overview of the development process for SimITK was presented. The process begins with a class header file that is captured in an XML file. This XML is used to create glue files which in turn create Simulink blocks. Finally, these blocks can be used to create SimITK process models. The next chapter discusses these transformations in further detail and presents a break-down of the components of SimITK. Also presented will be the modifications that have been made to transition SimITK to a usable research tool.
Chapter 4

Methods

In the previous chapter, a broad overview of the development process for SimITK was presented. This process includes XML parsing, template creation, and keyword expansion. In this chapter these components will be expanded upon.

The improvements that were made to begin to transition SimITK to a usable research tool from a proof-of-concept will also be shown. The main focus of these improvements was to enhance the usability of SimITK in two main ways. The first enhancement was to transition from supporting ITK version 3 to supporting ITK version 4, the most recent version of ITK. This involved moving to a new XML format and a large number of modifications to the templating process. This transition also fully automated the SimITK generation process. Secondly, the image registration pipeline was enhanced by allowing iterative, real-time monitoring of the metric value during registration and incorporating the option to set the output image size for certain filters.
4.1 SimITK Development Process

In order to understand the changes that have been made to transition SimITK to a usable research tool, the process of creating SimITK must be explained further.

The process to create SimITK is fully automated. At the highest level, SimITK is composed of three groups of inputs and three transitions. As shown in Figure 4.1 and outlined in Chapter 3, the process starts with the header files of classes in ITK version 4. Each class’ header file is translated into a simplified XML file through the use of a custom designed, automated wrapper created by Gobbi [12] [14]. Once the simplified XMLs are generated, a complex templating phase occurs (Section 4.2). This templating phase generates a set of block libraries based on the type of file (filter, optimizer, transform), the datatype, and the dimensionality (Section 4.3.5). These block libraries are then used in Simulink to create models such as the one shown as the final output of Figure 4.1.

4.2 SimITK Templating Process

The bulk of the work to create SimITK involves a templating process where source code is automatically generated through macro expansion of a set of base files. The base files are templates that represent the source files to be created. This is the transition that occurs from XML to glue files that was first presented in Section 3.1.2. This procedure takes the simplified XML files and creates the necessary glue files to generate the block libraries that appear in Simulink. This translation is done using perl scripts to automatically populate file templates that have been created for the various output files. An overview of how the main components of SimITK are created using this templating process can be seen in Figure 4.2, a detailed explanation of
Figure 4.1: SimITK development process overview.
4.2. SIMITK TEMPLATING PROCESS

Figure 4.2: SimITK templating overview.
Table 4.1: Templating process: responsible Perl script and resulting file extension.

<table>
<thead>
<tr>
<th>Templating Process</th>
<th>Responsible Perl Script</th>
<th>Generated File Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class Header Generation</td>
<td>TPPGen.pm</td>
<td>.tpp</td>
</tr>
<tr>
<td>Simulink System Function Generation</td>
<td>SFunctionGen.pm</td>
<td>.cpp</td>
</tr>
<tr>
<td>Matlab Block Property Generation</td>
<td>MatlabCallbackGen.pm</td>
<td>.m</td>
</tr>
<tr>
<td>Simulink Parameter Dialog Generation</td>
<td>FilterMaskGen.pm</td>
<td>.mdlpart</td>
</tr>
<tr>
<td>Simulink Library Generation</td>
<td>itkLibraryGen.pl</td>
<td>.mdl</td>
</tr>
</tbody>
</table>

the components can be found in Section 4.3. In Figure 4.2 it can be seen that there are three main inputs to the templating procedure, a class header template file, a Simulink system function template file, and the Simplified XML. These three components, along with other supplemental template files, are used to generate the Matlab executable interface file, Matlab block properties file, and the Simulink library which are used to create the block library in Simulink.

In order to understand exactly how the Matlab executable interface file, Matlab block properties file, and Simulink library files are generated, an expanded diagram of the process can be found in Figure 4.3. Further information on the names of the files that are responsible for performing these generations and the file extension of the resulting file can be found in Table 4.1. The Simplified XML files are parsed using an XML parsing script. This script generates a hash that contains all of the XML derived information that is used to create the final output files, which in turn are the glue of the SimITK development process: the Matlab executable interface file, block properties file, and Simulink library file. Each file that is output throughout
4.2. SimITK TEMPLATING PROCESS

Figure 4.3: SimITK templating process expanded.
the development process is generated through macro expansion of template files along with compilation of previously generated files or the use of CMake (in the case of the Simulink system function file). CMake\(^1\) is a cross-platform, open-source build system that is used to compile SimITK.

The Matlab executable interface file (MEX file) provides the interface between ITK C++ and Matlab, and has the most complicated generation process as can be seen in Figure 4.3. In order to create these MEX files, the class header file must first be generated by populating the class header template file from the simplified XML hash. The class header file is a C++ header file for the block. The Simulink system function (S-function) template file and the simplified XML hash are then used to generate the Simulink S-function file. The class header file is then used directly with the Simulink S-function file to create the MEX files through CMake compilation.

The process to generate the Simulink library file is the second most complicated templating process. It requires the creation of the Simulink parameter dialog files using the simplified XML hash and the parameter dialog template files. These parameter dialog files are then combined to generate the Simulink library files. The final output of this process is the Matlab block properties file which is generated by populating the block properties template files.

4.3 System Components

SimITK is composed of many different components that work together to create, update, and run the blocks that are found in the block libraries with Simulink. All of

\(^{1}\)http://www.cmake.org/
these files are automatically generated through macro expansion during the templating process that was described in Section 4.2.

4.3.1 Class Header File (Virtual Block)

The class header file which is known as the virtual block on the development side is a representation in code of the block that will be displayed in Simulink for a particular class. The class header file is in charge of setting up inputs and outputs, initializing the properties of the block, and setting up the various functions that will be used throughout the block execution.

For the inputs and outputs, the class header file sets up the appropriate number of regular inputs, regular outputs, special inputs, and special outputs. Special inputs are inputs that are created for a specific purpose in the code. One such specific purpose is setting up inputs that are controlled by a checkbox. Special outputs are similar in function, and can be controlled by a checkbox or set up specifically in the code. The class header also initializes the data information of any input images (size, spacing, origin) and converts Matlab matrices to ITK images that can be passed to ITK for image processing.

Along with setting up the inputs and outputs, the class header sets up all of the required information for the properties that will be displayed in the mask (see Section 4.3.4). Each of the properties is declared with the appropriate datatype in the private section of the class header file. These are then initialized to an appropriate value in the constructor of the class header. Mutators and accessors are generated for each property as well to allow them to be accessed and modified by the block. Finally the parameters are set using their original ITK methods with conversion to
ITK types if necessary.

The class header file is the backbone of SimITK as it does the majority of the setup that is required for the block to function accurately, along with conversions to allow information to pass from Matlab to ITK and vice versa.

## 4.3.2 Simulink System Function (S-function)

S-functions are custom blocks in Simulink that can be written in a variety of different programming languages. In SimITK S-functions are written in C++ since ITK is a C++ library. The S-function contains the code that will be executed when a model is run in Simulink. Since different types of blocks require different information to run, there are several different types of S-function template files depending on the type of the ITK class. These templates are automatically keyword-substituted by an S-function generator script.

The S-function file is split into several functions with the three most important being `mdlInitializeSizes`, `mdlStart`, and `mdlOutputs`. The responsibilities of each of these functions are outlined below:

- `mdlInitializeSizes` is in charge of the block setup, including configuring input and output ports, setting up parameters, and declaring special datatypes as needed
- `mdlStart` sets up parameters and declares user data
- `mdlOutputs` gets information from the input ports and the callback that is based on the model set up by the user

Both `mdlInitializeSizes` and `mdlStart` run once right at the beginning of the
block execution. `mdlOutputs` runs each iteration of the block and is an integral part of the S-function. For example, `mdlOutputs` is behind the setup of the value output port in the metric and optimizer in cases where it has been selected by the user in the mask (see Section 4.3.4). The S-function also manages the maintenance of the virtual block which it initializes, updates, and deletes when it is no longer needed.

### 4.3.3 Matlab Block Property File (Callback file)

The Matlab block property file or Callback file as it is known in Simulink is responsible for modifications based on changes made in the mask of the block (see Section 4.3.4) to allow for customization of that particular class. Each parameter within the mask has a function in the property file that handles updating that parameter if its attributes have changed. The property file can also contain code to update the inputs and outputs of the block in cases such as metrics and transforms where there are special inputs and outputs controlled by checkboxes in the mask.

### 4.3.4 Parameter Dialog (Simulink Mask)

The Simulink mask is a dialog box that appears when any Simulink block is double clicked. It contains options that can be set or unset by the user. In SimITK, this consists of setting parameters and enabling or disabling input and output ports. The parameter dialog file contains the information needed by Simulink to generate this mask, including the name of the block, the names of the parameters that can be modified, the options for those parameters (checkbox, drop down list, text box, etc.), and the default setting (do not set, off, etc.).
4.3. SYSTEM COMPONENTS

4.3.5 Simulink Library

Several Simulink libraries are generated during the execution of SimITK. Currently, libraries are separated by datatype, dimensionality, and class type. The datatypes that SimITK supports are \texttt{unsigned char}, \texttt{float}, \texttt{short}, and \texttt{unsigned short} with 2D and 3D images being supported. Class types are divided into filter, optimizer, and transform. Optimizers have one library, transforms are split into 2D and 3D, and filters are divided based on datatype and dimensionality. All of these libraries are made up of the Simulink Mask files (Section 4.3.4) which are combined together to create the larger library files.

4.3.6 System Components Example

To better understand why the many components of SimITK are necessary, a simple example that follows a function from the ITK header file to being displayed in a Simulink block will be presented. This will further expand the translations and artifacts that were discussed in Chapter 3 and earlier in this chapter.

For this example \texttt{ThresholdImageFilter} will be used. This is a simple filter that thresholds an image by setting image values to a specified value (by default ‘black’) if they are below, above, or between simple threshold values. In the header file for \texttt{ThresholdImageFilter} (\texttt{itkThresholdImageFilter.h}) one of the properties is called \texttt{Lower} and is of type \texttt{PixelType}. This property is the lower threshold for the filter. The process to display this property as an option in the Simulink mask (Section 4.3.4) starts with an XML representation of the property as seen in Listing 4.1.

As seen in Listing 4.1, the name of the property is \texttt{Lower} and its ITK type is
PixelType. The first step is to create the representation of this property in the class header through automatic macro expansion. Listing 4.2 shows the class header file (Section 4.3.1) code where \texttt{Lower} is declared, accessors and mutators are defined, and the property is initialized.

Two member variables are declared in Listing 4.2 for \texttt{Lower}: one with its type (\texttt{PixelType}) and one to determine if it has been set in the mask (see Section 4.3.4). The variable \texttt{isLowerSet} is used to check if \texttt{Lower} has been set in the mask; if it was not set, it indicates to the system that the default setting in ITK will be used and nothing will be set by SimITK. A mutator is created to set \texttt{Lower} and the flag to say that it is set and an accessor is created to retrieve the value of \texttt{Lower}. \texttt{Lower} is also initialized in the Virtual Block if it has been set. The next file to be created is the S-function (Section 4.3.2). The S-function gets the value from the \texttt{Lower} parameter and sets it as shown in Listing 4.3.

Next, the block property (callback) file (Section 4.3.3) is created and functions are set up to alter the appearance of the mask based on the users actions as shown in Listing 4.4. When the user selects to have \texttt{Lower} ‘As Parameter’ in the mask then the appearance of the mask is changed to allow the user to input a value in a text box. Finally the Simulink Mask (Section 4.3.4) file is created which sets up the initial appearance of the mask. An example for \texttt{ThresholdImageFilter} can be found in Appendix A.1 Listing A.1 which shows that for each property the default settings

Listing 4.1: XML listing lower property.

```
<property name="Lower" access="public" type="PixelType">
    <methods bitfield="GET|SET" access="public" />
</property>
```
Listing 4.2: Class header (virtual block) code for lower property.

```cpp
//Declaration:
typename InputPortType::PixelType m_Lower;
int LowerIsSet;
//Mutator:
void SetLower(typename InputPortType::PixelType value) {
    m_Lower = value;
    LowerIsSet = 1;
}
//Accessor:
typeinfo InputPortType::PixelType GetLower() {
    return m_Lower;
}
//Initialization:
if (LowerIsSet == 1) {
    m_Object->SetLower(m_Lower);
}
```

Listing 4.3: S-function code for lower property.

```cpp
//Get the value from the parameter:
INPUT_IMAGE_PIXELTYPE
    Lower = static_cast<INPUT_IMAGE_PIXELTYPE>(mxGetPr(ssGetSFcnParam(S,2*1 + 1))[0]);
//Set the parameter:
if (static_cast<int>(mxGetScalar(ssGetSFcnParam(S,2))) == 2) {
    vBlock->SetLower(Lower);
}
```

are defined. In the line where MaskVariables are defined (line 18) Lower and
LowerInputIndicator can be seen. The masks displaying the Lower property
from ThresholdImageFilter can be seen in Figure 4.4. The mask on the left hand side is the default mask with Lower not set. The mask on the right is the changed GUI option where the user wants to set Lower in the mask.

As seen by this example, all of the components of SimITK are necessary to set up the ability to modify a property in the mask.
4.3. SYSTEM COMPONENTS

Listing 4.4: Block property (callback) code for lower property.

```matlab
function LowerInputIndicatorCallback(block)

vals = get_param(block,'MaskValues');
vis = get_param(block,'MaskVisibilities');
if strcmp(vals{3},'As Parameter'),
    vis(4) = {'on'};
    set_param(gcb,'MaskVisibilities',vis),
else
    vis(4) = {'off'};
    set_param(gcb,'MaskVisibilities',vis),
end

%%%to update the blocks number of ports
set_param(block, 'MaskValues', vals);
SetPortLabels(block)

function LowerCallback(block)
```

Figure 4.4: ThresholdImageFilter mask: un-modified and modified.
4.4 Transitioning to ITK Version 4

One of the most important modifications required to increase the usability of SimITK was to update the ITK version that was supported to version 4. ITK version 4 implemented many changes including support for 64 bit operating systems and implementing a new modularized file structure.

For ITK version 3, SimITK had handwritten XML files for each of the classes that was wrapped. For ITK version 4, the process of creating SimITK is automated by using the generated simplified XML files from Gobbi [12] [14] and shown in Section 4.1. Simplified XML files are automatically generated from the ITK version 4 class header files.

These automatically generated ITK version 4 XML files differed greatly from the handwritten version 3 XML files in terms of structure, as many XML tags were renamed or removed. An example of the version 3 XML structure can be seen in Listing 4.5 with the corresponding version 4 XML file in Listing 4.6. In order to use these version 4 XML files to automatically generate the components needed for SimITK, changes had to be made to the XML generation script. This is the script that parses the XML files to create the hash that is used to populate the template files through macro expansion as explained in Section 4.2. These changes involved rewriting much of this script to determine the file type, inputs, outputs, template parameters, properties, datatypes, and dimensionalities of the ITK version 4 classes from their XML files. These parsing changes will be explored in detail in this section.
Listing 4.5: Example XML tags from ITK version 3 XML files.

```xml
<Filter>
  <Name>ResampleImageFilter</Name>
  <Template_Parameters>
    <Template_Parameter>TInputImage</Template_Parameter>
    <Template_Parameter>TOutputImage</Template_Parameter>
  </Template_Parameters>
  <Allowed_Datatypes>
    <Datatype>float</Datatype>
  </Allowed_Datatypes>
  <Allowed_Dimensionalities>
    <Dimensionality>3</Dimensionality>
  </Allowed_Dimensionalities>
  <Inputs>
    <Input>
      <Input_Name>Input</Input_Name>
      <Input_Type>ImageType</Input_Type>
      <Input_Dimension>ImageDimension</Input_Dimension>
    </Input>
  </Inputs>
  <Parameters>
    <Parameter>
      <Parameter_Name>OutputOrigin</Parameter_Name>
      <Parameter_Type>double</Parameter_Type>
      <Parameter_Size>1,1</Parameter_Size>
    </Parameter>
  </Parameters>
  <Outputs>
    <Output>
      <Output_Name>Output</Output_Name>
      <Output_Type>ImageType</Output_Type>
      <Output_Dimension>ImageDimension</Output_Dimension>
    </Output>
  </Outputs>
</Filter>
```
4.4. TRANSITIONING TO ITK VERSION 4

Listing 4.6: Example XML tags from ITK version 4 XML files.

```
<file name="itkResampleImageFilter.h">
  <namespace name="itk">
    <class name="ResampleImageFilter" template="1">
      <tparam name="TInputImage" type="typename" />
      <tparam name="TOutputImage" type="typename" />
      <tparam name="TInterpolatorPrecisionType" type="typename"
        value="double" />
      <base name="ImageToImageFilter<TInputImage, TOutputImage>"
        access="public" />
      <inheritance>
        <context name="ImageToImageFilter<TInputImage,
          TOutputImage>" access="public" />
        <context name="ImageSource<TOutputImage>" access="public" />
        <context name="ProcessObject" access="public" />
        <context name="Object" access="public" />
        <context name="LightObject" access="public" />
      </inheritance>
      <property name="OutputOrigin" access="public" type="double"
        pointer="*">
        <methods bitfield="SET" access="public" />
      </property>
      <property name="Input" context="ImageToImageFilter<TInputImage,
          TOutputImage>" access="public" type="InputImageType"
        pointer="*">
        <methods bitfield="GET|SET" access="public" />
      </property>
      <property name="Output" context="ImageSource<TOutputImage>"
        access="public" type="OutputImageType" pointer="*">
        <methods bitfield="GET" access="public" />
      </property>
    </namespace>
  </file>
```

4.4.1 File Types and Classes

Determining the type of the class whose XML is being examined is very important throughout the block generation process, since different types of classes are processed individually, have various exclusions, and are put in distinct block libraries. With the version 4 XML, determining these types was made more difficult because the type
tag did not exist. Another difficulty that was encountered with the version 4 XML files was the presence of additional class tags in certain XML files.

In the version 3 XML files, the file type was explicitly stated in the form of an XML tag at the beginning of the XML file. Files can be classified as either filter, metric, optimizer, transform, or TransformInitializer. In the version 4 XML files, this tag did not exist. Instead, the file type had to be deduced from the XML itself. The inheritance section was used to determine this information. The inheritance section outlines the parent classes of the class that is being wrapped and can be seen in Listing 4.7 (Note: In this listing several \texttt{&lt;} and \texttt{&gt;} special characters can be seen. These are the XML escape codes for the \textless{} and \textgreater{} symbols).

For all but the TransformInitializer classes, this inheritance section could be used to determine the type based on the context name attributes that were found. For example, in Listing 4.7, it can be seen that the DiscreteGaussianImageFilter class has five parent classes. The one that was used to determine that DiscreteGaussianImageFilter is a filter type was the ProcessObject parent class. The other corresponding context names to types are shown in Table 4.2. The only file type that could not be determined from the inheritance section was the TransformInitializer. Instead the class name was

Listing 4.7: Inheritance section from DiscreteGaussianImageFilter.xml.

85 \begin{verbatim}
86    <inheritance>
87    <context name="ImageToImageFilter\lt;TInputImage, TOutputImage\gt;" access="public" />
88    <context name="ImageSource\lt;TOutputImage\gt;" access="public" />
89    <context name="ProcessObject" access="public" />
90    <context name="Object" access="public" />
91    <context name="LightObject" access="public" />
92  </inheritance>
\end{verbatim}
### 4.4. TRANSITIONING TO ITK VERSION 4

Table 4.2: Class type to context name mapping.

<table>
<thead>
<tr>
<th>Class Type</th>
<th>Context Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filter</td>
<td>ProcessObject</td>
</tr>
<tr>
<td>Interpolator</td>
<td>InterpolateImageFunction</td>
</tr>
<tr>
<td>Metric</td>
<td>ImageToImageMetric</td>
</tr>
<tr>
<td>Optimizer</td>
<td>Optimizer</td>
</tr>
<tr>
<td>Transform</td>
<td>TransformBase</td>
</tr>
</tbody>
</table>

Listing 4.8: Two classes in XML from CannyEdgeDetectionImageFilter.xml.

```xml
<class name="ListNode" template="1">
  <tparam name="TValueType" type="typename" />
</class>

<class name="CannyEdgeDetectionImageFilter" template="1">
  ...
</class>
```

...checked before looking at the inheritance section to make sure that the current file was not a TransformInitializer type.

In certain XML files, one or more extra class tags were found as can be seen in Listing 4.8. The block generator script would pick up file names, template parameters, and properties from these extra classes since they could occur before the desired class in the XML file. As shown in Listing 4.8, the CannyEdgeDetectionImageFilter version 4 XML file has two classes: ListNode and CannyEdgeDetectionImageFilter. Changes had to be made throughout the XML parsing code to ensure that the appropriate class was used in the case where there were more than one. This code compared the current class to the file name to ensure that the correct one was used.
4.4.2 Template Parameters

Template parameters are used in ITK to identify, for example, the numerical datatype (short, unsigned char, float, unsigned short), and dimensionality (2D, 3D) that the class is templated over. In the version 3 XML, as shown in Listing 4.9, these template parameters were explicitly defined with specific names that would easily map in the TPP Generator and CPP Generator files. In the version 4 XML, as shown in Listing 4.10, these template parameters were still laid out explicitly and in the majority of cases (including the version 4 XML example), the names of the template parameters did not change.

There were, however, a few additions that had to be dealt with. In Listing 4.10, there is a new tparam attribute named TInterpolatorPrecisionType that did not exist in the version 3 XML in Listing 4.9. This new attribute represented a new template parameter that was added in ITK version 4 to the class header file and therefore it was picked up in the version 4 XML. However, because this template parameter had a default value (as can be seen in Listing 4.10) it could be ignored. There are a small number of cases where these template parameters with a value must be included, but those are dealt with individually in the code, since the majority of classes fall into the former set.

Listing 4.9: Version 3 template parameter example from ResampleImageFilter.xml.

```xml
<Template_Parameters>
    <Template_Parameter>TInputImage</Template_Parameter>
    <Template_Parameter>TOutputImage</Template_Parameter>
</Template_Parameters>
```
Listing 4.10: Version 4 template parameter example from ResampleImageFilter.xml.

```xml
<tparam name="TInputImage" type="typename" />  
<tparam name="TOutputImage" type="typename" />  
<tparam name="TInterpolatorPrecisionType" type="typename"  
  value="double" />  
```

4.4.3 Datatypes and Dimensionalities

SimITK templates its classes over four possible datatypes: unsigned char, unsigned short, float, and short, and two possible dimensionalities: 2D, and 3D. Not all classes, however, can support the full set of datatypes and dimensionalities. In the version 3 XML, the allowed datatypes and dimensionalities were explicitly listed in the XML as shown in Listing 4.11. The version 4 XML files did not contain any information about supported datatypes and dimensionalities except in the case where a dimension was part of a class name (e.g. Euler3DTransform, Rigid3DPerspectiveTransform, HoughTransform2DLinesImageFilter).

In order to know what datatypes and dimensionalities are supported by a particular class, a plain text file was created called AllowedDatatypesAndDimensionalities.txt that lists files that only support certain datatypes and dimensionalities

Listing 4.11: Version 3 datatype and dimensionality XML.

```xml
<Allowed_Datatypes>  
  <Datatype>float</Datatype>  
  <Datatype>short</Datatype>  
  <Datatype>unsigned char</Datatype>  
  <Datatype>unsigned short</Datatype>  
</Allowed_Datatypes>  
<Allowed_Dimensionalities>  
  <Dimensionality>2</Dimensionality>  
  <Dimensionality>3</Dimensionality>  
</Allowed_Dimensionalities>
```
4.4. TRANSITIONING TO ITK VERSION 4

Listing 4.12: AllowedDatatypesAndDimensionalities file excerpt.

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>116</td>
<td>CannyEdgeDetectionImageFilter;float;;</td>
</tr>
<tr>
<td>117</td>
<td>BinaryThinningImageFilter;short, unsigned char, unsigned short;2;</td>
</tr>
<tr>
<td>118</td>
<td>BinaryPruningImageFilter;float, unsigned char, unsigned short;2;</td>
</tr>
<tr>
<td>119</td>
<td>BinaryMinMaxCurvatureFlowImageFilter;float;;</td>
</tr>
<tr>
<td>120</td>
<td>AtanImageFilter;float;;</td>
</tr>
<tr>
<td>121</td>
<td>ApproximateSignedDistanceMapImageFilter;float, short, signed char;;</td>
</tr>
<tr>
<td>122</td>
<td>AntiAliasBinaryImageFilter;float;;</td>
</tr>
<tr>
<td>123</td>
<td>AnisotropicFourthOrderLevelSetImageFilter;float;;</td>
</tr>
<tr>
<td>124</td>
<td>Similarity3DTransform;3;</td>
</tr>
<tr>
<td>125</td>
<td>Similarity2DTransform;2;</td>
</tr>
</tbody>
</table>

in the format shown in Listing 4.12. This format was chosen since it is easily parseable by the perl script responsible for parsing the XML. If a particular class does not appear in this list then it is assumed that it supports all datatypes and dimensionalities.

4.4.4 Typedefs and ITK Types

There are many types available in ITK; some of these are C++ primitive types, such as double, int, and char. There are also many different ITK types, which are typedefs of the C++ primitives or for ITK class types such as `itk::Array<double>`. For example `PixelType` is just the type assigned to the pixels of an image which is a character string such as float or unsigned char, and is therefore a C++ primitive. `BoundValueType`, on the other hand, resolves to an ITK class type of `itk::Array<double>`. For the properties of these special types to be usable in Simulink, they had to be resolved to their primitive or ITK class types. In the version 3 XML, in many cases, these types were already resolved or were defined in a specific way so they would be easily mapped to their regular type in the XML parsing script. With the version 4 XML, some of these types had their original names and so they had to be resolved. One example of this was `BoundValueType`, which was
4.4. TRANSITIONING TO ITK VERSION 4


```xml
<Inputs>
  <Input>
    <Input_Name>Input</Input_Name>
    <Input_Type>ImageType</Input_Type>
    <Input_Dimension>ImageDimension</Input_Dimension>
  </Input>
</Inputs>

<Outputs>
  <Output>
    <Output_Name>Output</Output_Name>
    <Output_Type>ImageType</Output_Type>
    <Output_Dimension>ImageDimension</Output_Dimension>
  </Output>
</Outputs>
```

not previously recognized, but is an array of doubles so fairly easy to implement in Simulink. In that case, code was added to resolve and include properties with this type.

4.4.5 Inputs and Outputs

Inputs and outputs are another property of the SimITK blocks that were changed with the version 3 XML. Previously, there were input and output tags in the XML, as shown in Listing 4.13, that outlined the name, type, and dimension of the inputs or outputs. In the version 4 XML, inputs and outputs had to be determined from the properties that were listed in the XML. Since at this point images are the only inputs and outputs that are automatically generated from the XML, this was not a difficult task. An example input and output can be seen in Listing 4.14.

A list of possible input and output types was created and each property’s type was compared to this to determine if it was an input or an output. These properties then had to be excluded when generating the parameters for the block (that process
4.4. TRANSITIONING TO ITK VERSION 4


```xml
<property name="Input" context="ImageToImageFilter&lt;TInputImage,
TOutputImage&"> access="public" type="InputImageType" pointer="*">
  <methods bitfield="GET|SET" access="public" />
</property>

<property name="Output" context="ImageSource&lt;TOutputImage&gt;" access="public" type="OutputImageType" pointer="*">
  <methods bitfield="GET" access="public" />
</property>
```

will be outlined in Section 4.4.6). Along with checking the type of the property, a test was added to ensure that an input had \texttt{SET} or \texttt{GET} as its bitfield attribute value, and that outputs only had \texttt{GET} as their bitfield attribute value. Special accommodations also were made for certain classes where inputs or outputs were added to the XML that were actually from the parent class of the class.

4.4.6 Parameters/Properties

The XML property with the most changes when moving to ITK version 4 was the block’s parameters. In Listing 4.15, it can be seen that parameters were previously defined with a parameter tag that encompassed the tags for the parameter’s name, type, and size. This format allowed a single parameter to be defined with two different types: an ITK typedef and a primitive type. In the version 4 XML, these parameters were defined with a property tag that contained attributes relating to the property.

In Listing 4.16, it can be seen that the parameter called \texttt{OutputOrigin} in version 3 is now a property with the same name that appears twice in the version 4 XML. It appears twice because there are two \texttt{SetOutputOrigin()} methods in the header file. One with \texttt{OriginPointType} as the type which is a typedef for the ITK class \texttt{itk::Point}, and once with a \texttt{double*} type which is a pointer to
4.4. TRANSITIONING TO ITK VERSION 4

Listing 4.15: Version 3 parameter XML from ResampleImageFilter.xml.

```xml
<Parameter>
  <Parameter_Name>OutputOrigin</Parameter_Name>
  <Parameter_Type>double</Parameter_Type>
  <ITK_Parameter_Type>PointType</ITK_Parameter_Type>
  <Parameter_Size>1, ImageDimension</Parameter_Size>
</Parameter>
```

Listing 4.16: Version 4 property XML from ResampleImageFilter.xml.

```xml
<property name="OutputOrigin" access="public" type="OriginPointType">
  <methods bitfield="GET|SET" access="public" />
</property>

<property name="OutputOrigin" access="public" type="double" pointer="*">
  <methods bitfield="SET" access="public" />
</property>
```


A C++ primitive type. Also of note is the fact that the size is no longer included in
the property's information but that there are some additional useful attributes that
are included. The purpose of these new attributes will be discussed later on in this
section. With these large differences between parameters in version 3 and properties
in version 4, several changes were required in the XML parsing script.

One of the most important attributes of a property is its type, therefore, de-
termining this in the version 3 XML was the first priority. In the version 3 XML
(Listing 4.15), the type(s) of the property were laid out explicitly and it was obvious
to the parsing script whether it was dealing with a primitive type parameter or an
ITK typedef parameter. In the version 4 XML (Listing 4.16), the type was still
easily accessible, but no information existed about whether the type was a primitive
type or an ITK typedef. There was also the added complexity where some proper-
ties were defined multiple times in the version 4 XML with different types. In order to
determine whether a type was a primitive type or an ITK typedef, a list of primitive
type parameters was compiled based on what was tagged with Parameter_Type in the version 3 XML. Each type encountered was compared to this list to determine whether it was a primitive type or an ITK typedef. This allowed the property to be routed through the correct section of code depending on its type.

The corresponding issue of multiple definitions of the same property with different types still needed to be addressed. In order to deal with this issue, rules were added to ignore properties that were primitive type parameters with a pointer attribute which can be seen in the second OutputOrigin property in Listing 4.16. Code was then added in the ITK typedef parameter section to set a primitive type parameter for ITK parameters that required both. These mappings were determined based on the version 3 XML and updated as new classes were added.

Another issue that arose in the version 4 XML that was briefly mentioned at the beginning of this section was the lack of a Parameter_Size attribute in the new property definitions in the XML. The version 4 XML did not deal with sizes at all and the XML parsing script had to determine them. Code was added to set the property’s size based on its type which was determined from the version 3 XML files.

One benefit of the version 4 XML was added attributes that allowed properties to be excluded that would not logically appear in the Simulink mask. These attributes included the methods bitfield and methods access attributes. If the methods bitfield did not contain SET then the property was excluded since only properties that can be set should appear in the mask. Also excluded were properties whose methods access value was equal to private or protected instead of public since these properties could not be accessed to be set from the mask.

Finally, there were many minor changes that had to be made to support the
properties in the new XML. These included having to exclude properties that had already been identified as inputs or outputs, ignoring types that were currently not supported by SimITK, and identifying types that had different names in the new XML and then resolving them. These changes were predominantly made by adding exception code when parsing the properties in the XML.

4.4.7 Transform Initializer

The transform initializer, as discussed in Section 2.2.1 is a very important part of image registration since it initializes the transform to begin the registration. In the version 3 XML for this class, special XML tags were created and used that allowed four blocks to be generated from the one XML file. The reason that four blocks were generated was because four types of CenteredTransformInitializers were supported: Affine, CenteredEuler3D, CenteredRigid2D, and CenteredSimilarity2D. This was the only class in the version 3 XML that generated more than one block from a single XML file. This version 3 XML file can be seen in Appendix A.2 (Listing A.3).

When transitioning to ITK version 4 an XML file was automatically generated for the CenteredTransformInitializer as with all of the classes. This automatically generated file which can be seen in Appendix A.2 (Listing A.4) had a very similar structure to all of the generated ITK version 4 XML files and therefore only one block would have been created from this file without changes to the templating code.

To ensure that the four CenteredTransformInitializers would still be available in SimITK after transitioning to ITK version 4, changes were made to
the parsing script. These changes allowed the same version 4 XML file for the CenteredTransformInitializer to be run four times while setting the different names (Affine, CenteredEuler3D, CenteredRigid2D, CenteredSimilarity2D) and template parameters that were required for each block.

4.4.8 Exception Files

Since ITK is a very complex toolkit with many inputs, outputs, and properties that must be transformed for use in SimITK there are many of these that are not currently supported. In future, one the main goals needs to be to support more properties and types that are currently excluded. Although every effort is made to ensure that nothing essential to the classes that are currently supported is excluded, there is still a chance that an important property could be missing. In order to support further development and to ensure that the current classes are usable, code to generate exception files was added to the XML parsing script. This code creates a dump file of the properties and types that could not be translated for a particular class including the property name and a short explanation of why it was excluded. An example exception dump file can be found in Appendix A.2 for the ResampleImageFilter. There are many exceptions that should remain as such, including properties that are already included since they are inputs or outputs, and properties that are missing the SET attribute in the methods bitfield attribute; however many other exceptions could be resolved with further work.
4.5 Improved Usability

In order to begin to transition SimITK from a proof-of-concept to a usable research tool there were many modifications that needed to be made. Modifications generally came in the form of usability improvements that were suggested by a professor who had used SimITK with his class and developers of SimITK given their experience with the system; no formal usability studies have been performed at this time. These included: i) allowing the user to monitor the metric value in real-time during the registration models; ii) allowing the user to specify the size of the output image in the ResampleImageFilter; and iii) improving the usability of the code for future development. These issues are expanded upon in Sections 4.5.1-4.5.3. These changes improve the usability of SimITK in particular for the image registration pipeline, or to make the development experience smoother and more intuitive to allow further progress.

4.5.1 Metric Value output

As mentioned earlier, the metric block is a particularly important part of the image registration process. The metric allows the user to determine if the registration process is converging to a solution. In ITK, this value can be retrieved by the GetValue function in the metric classes. Previously, in SimITK, this value could not be displayed as its accessor was implemented differently from other GetValue functions.

The GetValue function for the metric classes in ITK is an exception to the general rule for the value accessor functions. Unlike the metric, most other GetValue functions do not require any parameters. For instance, this is the case in the optimizer classes which already had a value output in SimITK. The metric on the other hand
4.5. IMPROVED USABILITY

required the transform parameters to be passed to the function. In order to access these parameters changes needed to be made to all of the components of SimITK, to not only produce the output, but also to access the information to retrieve the transform parameters. Inputs had to be added for the fixed and moving image, as well as for the transform parameters which would be passed to GetValue. The fixed and moving images were fairly easy to manipulate since they were picked up automatically by the version 4 XML changes (see Section 4.4). The parameters input and value output were more complicated to implement as they needed to be represented as checkboxes in the mask of the metric block. In order to add an option such as checkboxes, changes had to be made across the system in a similar way to the example presented in 4.3.6. New template files had to be created for the metrics that would allow for checking the mask to see if the user wanted the additional input and output. Files also had to be added for changing the appearance of the block to display those ports if requested, and to get the value from the new ports.

Even though the change to add the option to have a metric value output affected every part of the SimITK templating process, the user now has access to a beneficial option.

4.5.2 Resample Image Filter Modifications

Previously, the user was unable to set the output image size that would result from the ResampleImageFilter. The system, therefore, continued to use its default, which is the size of the first input image. In the case of the ResampleImageFilter, this was the fixed image. Normally, the user would want the output image to have the size of the moving image and would change the values in the ResampleImageFilter
mask (the mask is described in Section 4.3.4) for size, spacing, and origin to match the moving image’s information. It is because of this lack of ability to set the size, that changes had to be made to the code to use the size that was specified by the user if it was available.

In order to implement this change, code was added to the S-function template file to check whether the current class being generated was the `ResampleImageFilter`. In that case, the system checked the drop down box for the size input in the mask to see if the user had specified a value. If the value existed, then it was used as the size for the output image; if not, then the default of using the first image’s size was maintained.

This change, although small, continues to improve the usability of SimITK by allowing the user to set the size of the output image.

### 4.5.3 SimITK Code Base Improvements

The code base for SimITK has existed for several years and is very large and complex as can be deduced from the templating diagram in Figure 4.3. In order to facilitate the continued development of SimITK, several usability improvements were needed. These included modifications to have the class header files mirror the process in native ITK by splitting one large function into three more representative ones, automatic initialization of variables, and general code cleanup.

The first task was to make changes in the class header template files to have their process more closely follow the process in ITK. Previously, the template files had one large function called `Run` that performed all of the initialization and updating of parameter settings. In ITK, this is performed in three groups of steps: initialization,
update information, and updating. To mirror this process the Run function was separated into three functions with similar names. This makes it easier for the developer to understand the purpose of each part of the code, and to write native C++ code for ITK if desired.

The second task was to automatically initialize the member variables that are added to the class header .tpp file through the virtual block generation process. This was fairly easy to do through keyword substitution since the code already existed to define these variables. The code was modified to initialize these variables in the constructor of the virtual block. This was added for good code practice and to avoid errors where a variable was not initialized and was being accessed.

Finally, some minor changes were made to the layout of the code to improve readability and modify variables names to reflect what they actually represented. This included renaming the filter variable in the S-function to be the more representative name of vBlock.

With these improvements SimITK should be easier to understand and develop.

4.6 Testing

In order to test the changes there were made to transition SimITK to ITK version 4 and to increase its usability two test suites were created.

The first was created to test the registration classes that were added (optimizers, metrics, and transforms). To create these tests a base registration model was created for 2D and 3D registration. A perl script was then used to automatically populate this base model file with the different classes that needed to be tested. These models were then run and the output was examined. If the output did not converge to a
solution this was noted and these notes will be included with the planned release.

The second test suite was created to test the filter blocks. For these models were created by hand with some reuse of input and output blocks. One model was created for each filter. These models were run and the output was saved. These outputs were examined for accuracy by the author and by a medical imaging professional. Only accurate models will be included in the planned release.

4.7 Summary

In this chapter an in depth view of the process of creating SimITK and its components was presented. Also discussed were the many changes that were made to increase the usability of SimITK to allow it to transition to a usable research tool. Changes included transitioning SimITK from supporting ITK version 3 to ITK version 4. This process required a large overhaul of the parsing process, modifications to the registration pipeline classes to allow a metric value output, and modifications to allow the user to specify the size of the resampled image. In the next chapter the results of these changes will be shown along with examples of the application of SimITK.
Chapter 5

Results

In the previous chapter an in depth overview of SimITK development was given along with detailed descriptions of its components. In this section the results of the modification to SimITK development are presented. First, an example of creating a simple model in SimITK is provided with notes on how to expand this concept to more complex models. Second, the results of moving to ITK version 4 are presented. Last, the results of the usability improvements to the registration pipeline are shown.

5.1 Available Simulink Block Libraries

The main artifacts of SimITK are the block libraries that are generated. As a result of transitioning to ITK version 4 there have been modifications to the blocks that are available. There are eleven SimITK libraries available in Simulink. Eight are for the Image Filters; one for each datatype (FLoat(FL), Short(SS), Unsigned Char(UC), Unsigned Short(US)) and dimensionality (2D, 3D) combination: FL2D, FL3D, SS2D, SS3D, UC2D, UC3D, US2D, and US3D. Two libraries are created for transforms; one for each dimensionality, and one library is created for optimizers. A total of thirty-seven image filters, twelve optimizers, and nineteen transforms that
5.2. MODEL CREATION

5.2.1 Creating a Simple Filter Model: FlipImageFilter

Creating and performing a specific medical imaging function in SimITK is a fairly simple process. The user first opens Matlab and then selects the Simulink library icon in the toolbar (shown outlined in red in Figure 5.1). Once the Simulink library has opened, the user can choose SimITK on the left hand side and then choose the appropriate library. For this example the UC2D (unsigned char, 2D) library was chosen. Once opened, the library will appear in the same way as shown in Figure 5.2, where it can be seen from the scrollbar on the right that there are many blocks available to choose from.

To create a simple model three blocks are required: an input image block, a filter, and an output image block. To add the first block to the model it is easiest to right
5.2. MODEL CREATION

Figure 5.2: Simulink UC2D block library.

click on the required block and choose “Add to a new model”. This will start an untitled canvas with the block that was right clicked on visible on it. As shown in Figure 5.3 `itkReaderUC2D` was added to an untitled model by following the above procedure.

Other blocks can be dragged to this untitled model to add them to the model. The result of dragging a filter (in this case `itkFlipImageFilterUC2D`) and an output block (`itkWriterUC2D`) can be seen in Figure 5.4. Circled in red are the input and output ports of these blocks. These ports need to be joined together to be able to successfully run the model. Of note here is the fact that each image block has two outputs; one for the image information and one for the image data. Both of these ports must be successfully connected for the model to be successful. To join the blocks together the user has to only hover over one of the output ports, this will
5.2. MODEL CREATION

Figure 5.3: Result of adding a block to an untitled model.

Figure 5.4: Input, filter, and writer blocks after being added to an untitled model.

change the cursor into a cross, they then click and drag to the appropriate input port. Figure 5.5 shows the three blocks after joining the input and output ports.

The next step is to set the input and output image names which is done by double clicking on the input or output blocks. This will bring up the mask (Section 4.3.4) as shown in Figure 5.6 and the user can set the input file name. A very similar mask
5.2. MODEL CREATION

Figure 5.5: Input, filter, and writer blocks with connections.

Figure 5.6: Mask for the input block to set the input file name.

will appear when double clicking the writer block and the user can set the output file name.

The last step before running the model is to set any properties in the mask of the filter that need to be modified. For the FlipImageFilter, the FlipAxes property must be set to indicate along which axis to flip the image. After double clicking on the FlipImageFilter block the mask will appear, the user then selects ‘As Parameter’ from the drop down list beside FlipAxes Indicator and then puts the desired value in the text box that will appear. The mask and a possible value can be seen in Figure 5.7.
Figure 5.7: FlipImageFilter mask where flip image filter properties are set.

Figure 5.8: The final model with play button to run the simulation circled in red.

The last step after all of the parameters have been set in the reader, writer, and filter is to run the model. This done by pushing the play button (circled in red in Figure 5.8). The user can then open the output file that is created in an image viewer of choice. For the example model created here the output would look like Figure 5.9(b) if the input image was as shown in Figure 5.9(a).

This is a simple example of creating a model to apply a filter to an input image.
5.2. MODEL CREATION

and write the result to an output image. If the user wanted to do the same function in pure C++ equivalent code (shown in Appendix B, (Listing B.1)) would require over 100 lines to perform the simple task of flipping an image. In comparison in SimITK, a simple model for flipping a model can be implemented by dragging and dropping those blocks, a clearly usable tool for a user with little to no C++ experience. More complicated models can be created in the same way as the simple model by dragging and dropping blocks, connecting them, and then running the resulting model.

5.2.2 A More Complex Model: Image Registration

The process to create more complex models in SimITK follows the exact same steps as were presented in Section 5.2.1. Blocks are selected from the appropriate library, dragged to the blank canvas, and then are attached by arrow connections. A 3D registration model is shown in Figure 5.10. This particular model registers a CT to an MRI. There are two image inputs for this model since registration requires a fixed image and a moving image. The CT is the fixed image and the MRI is the moving image which will be transformed to match the CT. Also in this model all
of the components of image registration that were discussed in Section 2.2.1 can be seen: the optimizer, metric, transform, and resample image filter. This model is more complex since it has more blocks but the process to create any model is always the same.

To run this model the user presses the play button in the top toolbar and an output image is saved. To check that this particular model has succeeded the user would open the result in a graphical environment of their choice (shown here is OCCIViewer [29]) in Figure 5.11 the fixed image, registration result, and overlay can be seen. As with the simple model example, creating ITK C++ code to perform this function would require approximately 300 lines of code where every component has to be defined, initialized, and used correctly for the registration to succeed. A large portion of this initialization and set up is taken care of by SimITK when registration models are used, allowing the user to quickly and easily perform the registration and focus on the development of workflow and algorithms in the model.
Figure 5.10: SimITK 3D registration model.
5.3 ITK Version 4 Block Creation

When transitioning to ITK version 4 there were many new classes that were added to the SimITK block libraries. The focus was on increasing the usability of the registration pipeline by adding more classes that could be used to create different registration models. This resulted in nineteen transforms, twelve optimizers, and five metrics being available. The testing and use of these is outlined in Section 5.3.1.

Another large group of classes that are included in SimITK are the filter classes. The focus when choosing which filters to wrap for SimITK and to test was to have the most popular filters available across a broad range of functionalities. Classes were chosen from smoothing, thresholding, geometry, segmentation (edge detection), and region of interest selection for a total of thirty-two. The testing of these classes is shown in Section 5.3.2. The creation of models using filters was shown as the simple example for using SimITK in Section 5.2.

Figure 5.11: 3D registration; (a) fixed image (CT) for 3D image registration (visualized in red); (b) moving image (MRI) in registered position; (c) overlay of CT and MRI to show registration result.
5.3. ITK VERSION 4 BLOCK CREATION

5.3.1 ITK Version 4 Registration Models

As shown in Section 2.2, registration is an important concept in medical image processing. When transitioning to ITK version 4, there was a focus on increasing the options for registration models that could be generated by increasing the number of optimizer, transform, and metric classes that are available. In this section the base registration model presented in Section 5.2 (Figure 5.10) is modified with examples of other blocks that can be substituted to create different registration models. These substitutions were used not only to show the increased usability of the system but also to test the new classes that had been added. The type of optimizer, transform, or metric that is chosen depends on many aspects including the type of the image, the dimensionality of the images, and the type of registration to be performed.

5.3.1.1 Registration Expansion 1: Optimizer Blocks

After transitioning to ITK version 4 SimITK now includes twelve optimizers. The optimizer library can be seen in Figure 5.12 and includes a wide variety of optimizers to choose from. All of these optimizers were tested as described in Section 4.6. In the base registration model shown in Figure 5.10 the optimizer is the RegularStepGradientDescentOptimizer which is a choice widely used for 3D registration. Other optimizers can also be used such as the PowellOptimizer which has been substituted in as shown in Figure 5.13. When tested, this model had the same output as was shown in Figure 5.11.
5.3. ITK VERSION 4 BLOCK CREATION

Figure 5.12: The SimITK optimizer library.

Figure 5.13: SimITK registration model with Powell optimizer outlined.
5.3.1.2 Registration Substitution 2: Transform Blocks

The transforms in SimITK are divided into two libraries: a 2D library and a 3D library. There are fifteen 2D transform classes and fourteen 3D classes available. The 3D transform library can be seen in Figure 5.14. As with the optimizers there are a large number of transforms that can be used in the registration pipeline depending on the type of registration. All of the included transforms were tested as described in Section 4.6. One option to replace the CenteredEuler3DTransform that was shown in Figure 5.10, is the AffineTransform. This substitution can be seen in Figure 5.15.

![SimITK 3D transform library diagram]

Figure 5.14: The SimITK 3D transform library.
5.3.1.3 Registration Substitution 3: Metric

Metrics in SimITK are included as part of the general block libraries that are broken down by dimensionality and datatype as were described in Section 5.1. There are five metrics that were tested and are available: i) \texttt{CorrelationCoefficientHistogramImageToImageMetric}; ii) \texttt{GradientDifferenceImageToImageMetric}; iii) \texttt{MattesMutualInformationImageToImageMetric}; iv) \texttt{MeanSquaresImageToImageMetric}; and v) \texttt{NormalizedCorrelationImageToImageMetric}. The metric that was included as part of the base registration model in Figure 5.10 was the \texttt{MattesMutualInformationImageToImageMetric} but as with the transform and optimizers many different metrics can be used depending on the type of registration. For 3D registration the \texttt{NormalizedCorrelationImageToImageMetric} was substituted, as shown in Figure 5.16.
5.3.1.4 A Note about 2D Registration

All of the testing that was presented in this Section was not only performed on a 3D registration base model but was also performed on a 2D model. The base 2D model and several possible substitutions are available in Appendix B Section B.2.

5.3.2 ITK Version 4 Image Filtering Models

The supported image filtering classes can be broken down into five general subsets: smoothing, threshold, geometry, edge detection, and region of interest selection. The supported classes in each group are as follows:

All of these filters were tested on one of two input images depending on the type of image that was supported by the filter. Both of these images can be seen in Figure 5.17. The smoothing, thresholding, geometry, and region classes were run on a brain proton density slice, a 2D unsigned char image that can be seen in Figure 5.17(a). The other image which is a CT slice of a head can be seen in
Table 5.1: Smoothing classes included in SimITK.

<table>
<thead>
<tr>
<th>Smoothing</th>
</tr>
</thead>
<tbody>
<tr>
<td>MeanImageFilter</td>
</tr>
<tr>
<td>MedianImageFilter</td>
</tr>
<tr>
<td>BilateralImageFilter</td>
</tr>
<tr>
<td>BinomialBlurImageFilter</td>
</tr>
<tr>
<td>DiscreteGaussianImageFilter</td>
</tr>
<tr>
<td>RecursiveGaussianImageFilter</td>
</tr>
<tr>
<td>SmoothingRecursiveGaussianImageFilter</td>
</tr>
</tbody>
</table>

Table 5.2: Thresholding classes included in SimITK.

<table>
<thead>
<tr>
<th>Thresholding</th>
</tr>
</thead>
<tbody>
<tr>
<td>ThresholdImageFilter</td>
</tr>
<tr>
<td>OtsuThresholdImageFilter</td>
</tr>
<tr>
<td>OtsuMultipleThresholdImageFilter</td>
</tr>
<tr>
<td>BinaryContourImageFilter</td>
</tr>
<tr>
<td>LabelContourImageFilter</td>
</tr>
<tr>
<td>BinaryThresholdImageFilter</td>
</tr>
</tbody>
</table>

Table 5.3: Geometry classes included in SimITK.

<table>
<thead>
<tr>
<th>Geometry</th>
</tr>
</thead>
<tbody>
<tr>
<td>FlipImageFilter</td>
</tr>
<tr>
<td>ResampleImageFilter</td>
</tr>
<tr>
<td>InterpolateImageFilter</td>
</tr>
</tbody>
</table>

Table 5.4: Edge detection classes included in SimITK.

<table>
<thead>
<tr>
<th>Edge Detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>DerivativeImageFilter</td>
</tr>
<tr>
<td>LaplacianImageFilter</td>
</tr>
<tr>
<td>LaplacianSharpeningImageFilter</td>
</tr>
<tr>
<td>SobelEdgeDetectionImageFilter</td>
</tr>
<tr>
<td>CannyEdgeDetectionImageFilter</td>
</tr>
<tr>
<td>ZeroCrossingBasedEdgeDetectionImageFilter</td>
</tr>
</tbody>
</table>
Table 5.5: Region of interest classes included in SimITK.

<table>
<thead>
<tr>
<th>Region of Interest</th>
</tr>
</thead>
<tbody>
<tr>
<td>CropImageFilter</td>
</tr>
<tr>
<td>ConstantPadImageFilter</td>
</tr>
<tr>
<td>MirrorPadImageFilter</td>
</tr>
</tbody>
</table>

Figure 5.17(b). This image was used for the edge detection classes since they required an image of type float. Results from running various filters on these images are presented next with full results available in Appendix B.

Figure 5.17: Images used for testing ITK version 4 filters; (a) brain proton density slice (UC2D); (b) CT slice of a head (FL2D).
5.3.2.1 Smoothing

Two representative classes in the smoothing group are the MeanImageFilter and the DiscreteGaussianImageFilter. The output for the MeanImageFilter can be seen in Figure 5.18(a) where an averaging filter has been applied to the image. The output for DiscreteGaussianImageFilter can be seen in Figure 5.18(b) and is an example of Gaussian blurring. Full results for the smoothing filters can be found in Appendix B Section B.3.1.

![Figure 5.18: Effect of smoothing filters on brain proton density slice (Figure 5.17(a)); (a) effect of MeanImageFilter; (b) effect of DiscreteGaussianImageFilter.](image)

5.3.2.2 Thresholding

For thresholding, both BinaryContourImageFilter and LabelContourImageFilter required a binary image. One was generated from brain proton density slice (Figure 5.17(a)) and is shown in Figure 5.19. The rest of the filters were run on
Figure 5.17(a). One example from the binary group (BinaryContourImageFilter) and one from the regular group (OtsuMultipleThresholdsImageFilter) are shown in Figure 5.20. The rest of the results are available in Appendix B Section B.3.2.

BinaryContourImageFilter was run on the binary version of the brain proton density slice as shown in Figure 5.19. The result of this filter extracts the pixels on the contours of the image as can be seen in Figure 5.20(a). OtsuMultipleThresholdsImageFilter which thresholds an image by using multiple Otsu thresholds was applied to the regular brain proton density slice as seen in Figure 5.17(a) and the result can be seen in Figure 5.20(b).
5.3.2.3 Geometry

Geometry is a small group of classes with two being part of the registration pipeline. The results for InterpolateImageFilter and ResampleImageFilter can be seen as part of the registration testing that is outlined in Section 5.3.1.

The only additional filter is FlipImageFilter which flips an image along a user specified axis and an example of its use was shown in Section 5.2.1.

5.3.2.4 Edge Detection

Edge detection classes were run on a 2D float image of a CT head slice. The outputs of two filters can be seen in Figure 5.21. As can be seen both of these detect the edges of the image. The CannyEdgeDetectionImageFilter as seen in Figure 5.21(a) detects and outlines the edges of the image. DerivativeImageFilter as shown in Figure 5.21(b) computes the derivative of an image to identify its edges. Full
results for the edge detection filters can be found in Appendix B Section B.3.3.

![Figure 5.21: Effect of edge detection filters on CT head slice (Figure 5.17(b)); (a) effect of CannyEdgeDetectionImageFilter; (b) effect of DerivativeImageFilter.]

5.3.2.5 Region

Region of interest classes manipulate the region of an image by extraction or expansion (padding). These were run on the brain proton density slice image. All of these filters change the size of the image in some way whether it is through cropping or padding. The results can be seen in Figure 5.22 for the three filters that are supported. Figure 5.22(a) shows the result of cropping the image by a specified region size and Figures 5.22(b) and 5.22(c) show the results after padding an image by a constant value and by mirroring, respectively.
5.4 Usability Improvements

There were many changes that were made to increase the usability of SimITK with a particular focus on the registration pipeline. Here the results of changes to the metric block and the resample image filter are shown.

5.4.1 Metric Block Output

Previously, as shown in Figure 5.23, the only available value outputs in SimITK were from the optimizer and the transform. This needed to be improved to include the metric value output. The metric value is particularly important to image registration as was highlighted in Section 2.2.1 since it allows the user to track the progression of the registration. Figure 5.24 shows that having this metric value output is now
5.4. USABILITY IMPROVEMENTS

Figure 5.23: Old SimITK registration model showing available outputs from [9].

Figure 5.24 shows the value of the metric in the purple box outlined in red in the bottom right hand corner. This value can also be graphed using the Simulink scope block, shown here with the optimizer value in the top graph and the metric value in the bottom. This graph demonstrates that registration has converged and has therefore most likely been successful. It can be seen when comparing the two pipelines that several changes were made to support the metric value output, including having the images as inputs to the block, having parameters input that is optional (it does not appear in the top metric block), and having the value output that is optional.
5.4. USABILITY IMPROVEMENTS

5.4.2 Resample Image Filter Modified Mask Parameters

The resampling phase is a very important part of the registration pipeline as was outlined in Section 2.2.1. Previously, it was not possible to set the size of the image...
5.4. USABILITY IMPROVEMENTS

Figure 5.25: Registration model showing the ResampleImageFilter with size, scaling, and origin values set.

to be output by the ResampleImageFilter. The default setting of filters using the size of their input image as the size of their output image was used instead of the size input by the user. Changes were made to have SimITK use the user-specified size. In Figure 5.25 the mask for the ResampleImageFilter is shown with the size, origin, and scaling values set. To demonstrate that this size is now applied to the output image, Figure 5.26 shows the output image with its image information. It can be seen that the output image size under the ‘Matrix’ value at the top of the
5.5. SUMMARY

Figure 5.26: Output image from 3D registration with image information window. The information box is 256x56x33 which is also the same value that was input in the ResampleImageFilter mask showing that this size is actually being applied to the output image.

5.5 Summary

In this chapter the results of changes made to SimITK that were presented in Chapter 4 were outlined. This included usability improvements for the registration pipeline such as the metric value output and the ability to use a wide variety of registration components. Also outlined was a large number of filters that can be used to perform different medical imaging tasks. In the next chapter the main ideas presented in this
thesis are summarized and suggested areas for improvement are presented.
Chapter 6

Conclusions and Future Work

6.1 Summary of Conclusions

In this work, SimITK, a system that wraps ITK libraries into the Simulink MDE environment using a fully automated process was presented. SimITK is a collection of block libraries that can be quickly and easily combined together in the model driven engineering environment, Simulink, to perform medical imaging tasks. Simulink allows all programming code to be abstracted away and replaced with a graphical representation. We believe this will greatly reduce the learning curve for ITK and allow the user to focus on developing workflows and algorithms. The usability of SimITK has been the focus of this work, with changes being made to all aspects of the system to allow it to transition to a usable research tool. Increasing the usability of the registration pipeline, an important imaging task, and maintaining currency through transitioning to ITK version 4 are both goals that have been met. The author will be creating a release of this version of SimITK, which will be available along with updated installation documentation and tutorials at www.simitkvtk.com.
Thirty-seven image filers, twelve optimizers, and nineteen transforms are successfully wrapped. While transitioning to ITK version 4 a new XML structure was adopted that has allowed the process to create SimITK to be fully automated. Also implemented were many usability improvements including a metric value output for the registration pipeline. All of these changes begin the transition for SimITK to become a usable research tool. There are, however, several areas for improvement that would further enhance the usability of the system and provide benefits for users.

6.2 Future Work

Although the currently supported subset of sixty-eight classes is useful it would be very beneficial to expand them. In order to facilitate this improvement the templating process needs to be expanded to support and resolve many of the ITK types that are not currently supported.

There are many classes that could be wrapped into SimITK but would not be useful in their present state because of missing properties. Support for properties of 

\texttt{RadiusType} and \texttt{OrderType} for example would be very beneficial to increase the number of filters that could be used.

Another barrier to wrapping more classes is the set up of the block inputs and outputs that assumes all filters have one input and one output unless a special case is added. Generalizing this section would greatly increase the number of classes that could be supported. These are just a few places to start to increase the number of supported classes. The exception files that are automatically generated as part of the ITK transition work (Section 4.4.8) provide more information on the types that are not currently supported but would be beneficial to the current classes as well.
In order to determine which classes or functionalities to add, a usability study should be performed. The study could also identify parameters in the blocks where it would be beneficial to provide a default value or a different mechanism to set them.

When testing the changes that were outlined in Chapter 4 all models were made in one of two ways: Perl script generation or manually. For testing the classes that are part of the registration model, test models were created by a perl script that populated a base model template and saved a new model file for each class. For the filters, all models were created by hand with some reuse of inputs and outputs.

There are several ways that testing could be improved for SimITK. For testing the outputs of the models that are created an automatic test suite could be created. Using a Matlab script the test files could be run and their outputs could be verified against a predetermined output. The system could then report the success of failure of the run as well as the output. Another set of tests that could be added would be for the parameters themselves to ensure that they are correctly set. A dump file could be created that prints all of the parameters as they are seen in the back end of SimITK for verification of the models. Lastly, some testing could be added in the code itself to ensure that users were not entering values that were incorrect or out of bounds. Along with this, intelligent error messages could be used to inform the user of the correct parameter structure.

In SimITK’s sister project SimVTK, the documentation for each block is available to the user with a click. This information is obtained from the header file of the class in VTK and is automatically added during the SimVTK wrapping process. This is another potential addition to SimITK that would make the classes easier to use and understand by eliminating the need to search for native ITK documentation.
Bibliography


Appendix A

SimITK File Examples and Templates

A.1 ITK Process Examples

Listing A.1: Simulink mask file for ThresholdImageFilter.

```plaintext
Block {
  BlockType "S-Function"
  Name "itkThresholdImageFilterFL2D"
  Ports [2, 2]
  Position [@LEFT®, @TOP®, @RIGHT®, @BOTTOM®]
  BackgroundColor "[1, 1, 1]"
  LoadFcn "val = get_param(gcb,'MaskValues');\n  set_param(gcb,'MaskValues', val);\n  nSimITKThresholdImageFilterFL2DCB('SetPortLabels', gcb);"
  FunctionName "SimITKThresholdImageFilterFL2DMat"
  Parameters
    "OutsideValueInputIndicator,OutsideValue,LowerInputIndicator,Lower,
    UpperInputIndicator,Upper,InPlaceInputIndicator,InPlace,
    AbortGenerateDataInputIndicator,AbortGenerateData,
    ProgressInputIndicator,Progress,
    ReleaseDataFlagInputIndicator,
    ReleaseDataFlag,ReleaseDataBeforeUpdateFlagInputIndicator,
    ReleaseDataBeforeUpdateFlag,NumberOfThreadsInputIndicator,
    NumberOfThreads,DebugInputIndicator,Debug,
    ReferenceCountInputIndicator,ReferenceCount,
    GlobalWarningDisplayInputIndicator,GlobalWarningDisplay"
  MaskPromptString "OutsideValue Indicator|OutsideValue Value|Lower
    Indicator|Lower Value|Upper Indicator|Upper Value|InPlace Indicator|InPlace
    Value|AbortGenerateData Indicator|AbortGenerateData Value|Progress
    Indicator|Progress Value|ReleaseDataFlag Indicator|ReleaseDataFlag
    Value|ReleaseDataBeforeUpdateFlag Indicator|ReleaseDataBeforeUpdateFlag
    Value|NumberOfThreads Indicator|NumberOfThreads Value|Debug Indicator|Debug
    Value|ReferenceCount Indicator|ReferenceCount Value|GlobalWarningDisplay
    Indicator|GlobalWarningDisplay Value"
```
MaskStyleString "popup(Do Not Set|As Parameter),edit,popup(Do Not Set|As Parameter),edit,popup(Do Not Set|As Parameter),edit,popup(Do Not Set|As Parameter),edit,popup(Do Not Set|As Parameter),edit,popup(Do Not Set|As Parameter),edit,popup(Do Not Set|As Parameter),edit,popup(Do Not Set|As Parameter),edit,popup(Do Not Set|As Parameter),edit,popup(Do Not Set|As Parameter),edit,popup(Do Not Set|As Parameter),edit,popup(Do Not Set|As Parameter),edit,popup(Do Not Set|As Parameter),edit,popup(Do Not Set|As Parameter),edit" 

MaskTunableValueString "on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on," 

MaskCallbackString 
"SimITKThresholdImageFilterFL2DCB('OutsideValueInputIndicatorCallback',gcb)
|SimITKThresholdImageFilterFL2DCB('OutsideValueCallback',gcb)
|SimITKThresholdImageFilterFL2DCB('LowerInputIndicatorCallback',gcb)
|SimITKThresholdImageFilterFL2DCB('LowerCallback',gcb)
|SimITKThresholdImageFilterFL2DCB('UpperInputIndicatorCallback',gcb)
|SimITKThresholdImageFilterFL2DCB('UpperCallback',gcb)
|SimITKThresholdImageFilterFL2DCB('InPlaceInputIndicatorCallback',gcb)
|SimITKThresholdImageFilterFL2DCB('InPlaceCallback',gcb)
|SimITKThresholdImageFilterFL2DCB('AbortGenerateDataInputIndicatorCallback',gcb)
|SimITKThresholdImageFilterFL2DCB('AbortGenerateDataCallback',gcb)
|SimITKThresholdImageFilterFL2DCB('ProgressInputIndicatorCallback',gcb)
|SimITKThresholdImageFilterFL2DCB('ProgressCallback',gcb)
|SimITKThresholdImageFilterFL2DCB('ReleaseDataFlagInputIndicatorCallback',gcb)
|SimITKThresholdImageFilterFL2DCB('ReleaseDataFlagCallback',gcb)
|SimITKThresholdImageFilterFL2DCB('ReleaseDataBeforeUpdateFlagInputIndicatorCallback',gcb)
|SimITKThresholdImageFilterFL2DCB('ReleaseDataBeforeUpdateFlagCallback',gcb)
|SimITKThresholdImageFilterFL2DCB('NumberOfThreadsInputIndicatorCallback',gcb)
|SimITKThresholdImageFilterFL2DCB('NumberOfThreadsCallback',gcb)
|SimITKThresholdImageFilterFL2DCB('DebugInputIndicatorCallback',gcb)
|SimITKThresholdImageFilterFL2DCB('DebugCallback',gcb)
|SimITKThresholdImageFilterFL2DCB('ReferenceCountInputIndicatorCallback',gcb)
|SimITKThresholdImageFilterFL2DCB('ReferenceCountCallback',gcb)
|SimITKThresholdImageFilterFL2DCB('GlobalWarningDisplayInputIndicatorCallback',gcb)
|SimITKThresholdImageFilterFL2DCB('GlobalWarningDisplayCallback',gcb)"

MaskEnableString "on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on," 

MaskVisibilityString "on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on," 

MaskToolTipString "on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on,on," 

MaskVarAliasString "" 

MaskVariables "OutsideValueInputIndicator=@1;OutsideValue=@2;LowerInputIndicator=@3;Lower=@4;UpperInputIndicator=@5;Upper=@6;InPlaceInputIndicator=@7;InPlace=@8;AbortGenerateDataInputIndicator=@9;AbortGenerateData=@10;ProgressInputIndicator=@11;Progress=@12;ReleaseDataFlagInputIndicator=@13;ReleaseDataFlag=@14;ReleaseDataBeforeUpdateFlagInputIndicator=@15;ReleaseDataBeforeUpdateFlag=@16;NumberOfThreadsInputIndicator=@17;NumberOfThreads=@18;DebugInputIndicator=@19;Debug=@20;ReferenceCountInputIndicator=@21;ReferenceCount=@22;GlobalWarningDisplayInputIndicator=@23;GlobalWarningDisplay=@24;"

MaskDisplay "port_label('input',1,'Input [info]');\nport_label('input',2,'Input [data]');\nport_label('output',1,'Output [info]');\nport_label('output',2,'Output [data]');\n" 

MaskSelModifiable on 

MaskIconFrame on 

MaskIconOpaque on
MaskIconRotate "none"
MaskIconUnits "autoscale"
MaskValueString "Do Not Set|0|Do Not Set|0|Do Not Set|0|Do Not Set|0|Do Not Set|0|Do Not Set|0|Do Not Set|0|Do Not Set|0|Do Not Set|0|Do Not Set|0|Do Not Set|0|Do Not Set|0|Do Not Set|0|Do Not Set|0|Do Not Set|0|Do Not Set|0"
MaskTabNameString ",,,,,,,,,,,,,,,,,,,,,"
A.2 ITK Version 4 Conversion Examples

Listing A.2: Exception dump file for ResampleImageFilter class.

NameOfClass : Exception 4 (no SET in the bitfield attribute)
Extrapolator : Exception 15 (has one of many ITK types that has been excluded)
OutputSpacing : Exception 6 (Has resample or fastMarching in the name and the name is output and the ITKParameterTypeValue is not defined)
OutputOrigin : Exception 6 (Has resample or fastMarching in the name and the name is output and the ITKParameterTypeValue is not defined)
OutputParametersFromImage : Exception 15 (has one of many ITK types that has been excluded)
ReferenceImage : Exception 15 (has one of many ITK types that has been excluded)
MTime : Exception 4 (no SET in the bitfield attribute)
Input : Exception 1 (is an input/output)
Output : Exception 1 (is an input/output)
InputNames : Exception 4 (no SET in the bitfield attribute)
RequiredInputNames : Exception 4 (no SET in the bitfield attribute)
Inputs : Exception 4 (no SET in the bitfield attribute)
NumberOfInputs : Exception 4 (no SET in the bitfield attribute)
NumberOfOutputs : Exception 4 (no SET in the bitfield attribute)
OutputNames : Exception 4 (no SET in the bitfield attribute)
Outputs : Exception 4 (no SET in the bitfield attribute)
IndexedInputs : Exception 4 (no SET in the bitfield attribute)
NumberOfIndexedInputs : Exception 4 (no SET in the bitfield attribute)
NumberOfValidRequiredInputs : Exception 4 (no SET in the bitfield attribute)
IndexedOutputs : Exception 4 (no SET in the bitfield attribute)
NumberOfIndexedOutputs : Exception 4 (no SET in the bitfield attribute)
MultiThreader : Exception 4 (no SET in the bitfield attribute)
PrimaryInput : Exception 4 (no SET in the bitfield attribute)
PrimaryOutput : Exception 4 (no SET in the bitfield attribute)
NumberOfRequiredInputs : Exception 4 (no SET in the bitfield attribute)
NumberOfRequiredOutputs : Exception 4 (no SET in the bitfield attribute)
TimeStamp : Exception 4 (no SET in the bitfield attribute)
MetaDataDictionary : Exception 15 (has one of many ITK types that has been excluded)

```xml
<Filter_Descriptions>
  <TransformInitializer>
    <Name>CenteredTransformInitializer</Name>
    <TransformType>
      <TransformType_Name>CenteredRigid2DTransform</TransformType_Name>
      <TransformType_Template_Parameters>
        <TransformType_Template_Parameter>TScalarType</TransformType_Template_Parameter>
      </TransformType_Template_Parameters>
    </TransformType>
    <Template_Parameters>
      <Template_Parameter>TTransform</Template_Parameter>
      <Template_Parameter>TFixedImage</Template_Parameter>
      <Template_Parameter>TMovingImage</Template_Parameter>
    </Template_Parameters>
    <Allowed_Dimensionalities>
      <Dimensionality>2</Dimensionality>
    </Allowed_Dimensionalities>
    <Inputs>
      <Input>
        <Input_Name>FixedImage</Input_Name>
      </Input>
      <Input>
        <Input_Name>MovingImage</Input_Name>
      </Input>
    </Inputs>
    <Parameters>
      <Parameter>
        <Parameter_Name>Transform</Parameter_Name>
        <ITK_Parameter_Type>TransformType</ITK_Parameter_Type>
        <Parameter_Size>1,1</Parameter_Size>
      </Parameter>
    </Parameters>
  </TransformInitializer>
  <TransformInitializer>
    <Name>CenteredTransformInitializer</Name>
    <TransformType>
      <TransformType_Name>CenteredSimilarity2DTransform</TransformType_Name>
      <TransformType_Template_Parameters>
        <TransformType_Template_Parameter>TScalarType</TransformType_Template_Parameter>
      </TransformType_Template_Parameters>
    </TransformType>
    <Template_Parameters>
      <Template_Parameter>TTransform</Template_Parameter>
      <Template_Parameter>TFixedImage</Template_Parameter>
      <Template_Parameter>TMovingImage</Template_Parameter>
    </Template_Parameters>
    <Allowed_Dimensionalities>
      <Dimensionality>2</Dimensionality>
    </Allowed_Dimensionalities>
    <Inputs>
      <Input>
        <Input_Name>FixedImage</Input_Name>
      </Input>
      <Input>
        <Input_Name>MovingImage</Input_Name>
      </Input>
    </Inputs>
    <Parameters>
      <Parameter>
        <Parameter_Name>Transform</Parameter_Name>
        <ITK_Parameter_Type>TransformType</ITK_Parameter_Type>
        <Parameter_Size>1,1</Parameter_Size>
      </Parameter>
    </Parameters>
  </TransformInitializer>
</Filter_Descriptions>
```
A.2. ITK VERSION 4 CONVERSION EXAMPLES

```xml

<Parameter>
  <Parameter_Name>Transform</Parameter_Name>
  <ITK_Parameter_Type>TransformType</ITK_Parameter_Type>
  <Parameter_Size>1,1</Parameter_Size>
</Parameter>

<Parameters>
</Parameters>
</TransformInitializer>

<TransformInitializer>
  <Name>CenteredTransformInitializer</Name>
  <TransformType>
    <TransformType_Name>CenteredEuler3DTransform</TransformType_Name>
    <TransformType_Template_Parameters>
      <TransformType_Template_Parameter>TScalarType</TransformType_Template_Parameter>
    </TransformType_Template_Parameters>
  </TransformType>
  <Template_Parameters>
    <Template_Parameter>TTransform</Template_Parameter>
    <Template_Parameter>TFixedImage</Template_Parameter>
    <Template_Parameter>TMovingImage</Template_Parameter>
  </Template_Parameters>
  <Allowed_Dimensionalities>
    <Dimensionality>3</Dimensionality>
  </Allowed_Dimensionalities>
  <Inputs>
    <Input>
      <Input_Name>FixedImage</Input_Name>
    </Input>
    <Input>
      <Input_Name>MovingImage</Input_Name>
    </Input>
  </Inputs>
  <Parameters>
    <Parameter>
      <Parameter_Name>Transform</Parameter_Name>
      <ITK_Parameter_Type>TransformType</ITK_Parameter_Type>
      <Parameter_Size>1,1</Parameter_Size>
    </Parameter>
  </Parameters>
</TransformInitializer>

<TransformInitializer>
  <Name>CenteredTransformInitializer</Name>
  <TransformType>
    <TransformType_Name>AffineTransform</TransformType_Name>
    <TransformType_Template_Parameters>
      <TransformType_Template_Parameter>TScalarType</TransformType_Template_Parameter>
      <TransformType_Template_Parameter>NDimensions</TransformType_Template_Parameter>
    </TransformType_Template_Parameters>
  </TransformType>
  <Template_Parameters>
    <Template_Parameter>TTransform</Template_Parameter>
    <Template_Parameter>TFixedImage</Template_Parameter>
    <Template_Parameter>TMovingImage</Template_Parameter>
  </Template_Parameters>
  <Allowed_Dimensionalities>
    <Dimensionality>3</Dimensionality>
  </Allowed_Dimensionalities>
  <Inputs>
    <Input>
      <Input_Name>FixedImage</Input_Name>
    </Input>
    <Input>
      <Input_Name>MovingImage</Input_Name>
    </Input>
  </Inputs>
  <Parameters>
    <Parameter>
      <Parameter_Name>Transform</Parameter_Name>
      <ITK_Parameter_Type>TransformType</ITK_Parameter_Type>
      <Parameter_Size>1,1</Parameter_Size>
    </Parameter>
  </Parameters>
</TransformInitializer>
```
<Template_Parameters>
  <Allowed_Dimensionalities>
    <Dimensionality>2</Dimensionality>
    <Dimensionality>3</Dimensionality>
  </Allowed_Dimensionalities>
  <Inputs>
    <Input>
      <Input_Name>FixedImage</Input_Name>
    </Input>
    <Input>
      <Input_Name>MovingImage</Input_Name>
    </Input>
  </Inputs>
  <Parameters>
    <Parameter>
      <Parameter_Name>Transform</Parameter_Name>
      <ITK_Parameter_Type>TransformType</ITK_Parameter_Type>
      <Parameter_Size>1,1</Parameter_Size>
    </Parameter>
  </Parameters>
</TransformInitializer>

```xml
<file name=" itkCenteredTransformInitializer.h">
<namespace name=" itk">
<class name=" CenteredTransformInitializer" template=" 1">
<tparam name=" TTransform" type=" typename" />
<tparam name=" TFixedImage" type=" typename" />
<tparam name=" TMovingImage" type=" typename" />
<base name=" Object" access=" public" />
<inheritance>
<context name=" Object" access=" public" />
<context name=" LightObject" access=" public" />
</inheritance>

<property name=" NameOfClass" access=" public" type=" char" pointer=" *">
<methods bitfield=" GET" access=" public" />
</property>

<property name=" Transform" access=" public" type=" TransformType" pointer=" *">
<methods bitfield=" SET" access=" public" />
</property>

<property name=" FixedImage" access=" public" type=" FixedImageType" pointer=" *">
<methods bitfield=" SET" access=" public" />
</property>

<property name=" MovingImage" access=" public" type=" MovingImageType" pointer=" *">
<methods bitfield=" SET" access=" public" />
</property>

<property name=" FixedCalculator" access=" public" type=" FixedImageCalculatorType" pointer=" *">
<methods bitfield=" GET" access=" public" />
</property>

<property name=" MovingCalculator" access=" public" type=" MovingImageCalculatorType" pointer=" *">
<methods bitfield=" GET" access=" public" />
</property>

<property name=" Debug" context=" Object" access=" public" type=" bool">
<methods bitfield=" GET|SET|SET_BOOL" access=" public" />
</property>

<property name=" MTime" context=" Object" access=" public" type=" unsigned long">
<methods bitfield=" GET" access=" public" />
</property>

<property name=" TimeStamp" context=" Object" access=" public" type=" TimeStamp">
<methods bitfield=" GET" access=" public" />
</property>

<property name=" ReferenceCount" context=" Object" access=" public" type=" int">
<methods bitfield=" GET|SET" access=" public" />
</property>
</class>
</namespace>
</file>
```
<property name="GlobalWarningDisplay" context="Object" access="public" static="1"
type="bool">
<methods bitfield="GET|SET|SET_BOOL" access="public" />
</property>

<property name="MetaDataDictionary" context="Object" access="public"
type="MetaDataDictionary">
<methods bitfield="GET|SET" access="public" />
</property>
</class>
</namespace>
</file>
Appendix B

Transitioning to ITK Version 4: Models and Testing

B.1 Creating Models in Simulink with SimITK

Listing B.1: FlipImageFilter C++ code from [18].

```cpp
#include "itkFixedArray.h"
#include "itkFlipImageFilter.h"
#include "itkImage.h"
#include "itkImageFileReader.h"
#include "itkRGBPixel.h"

#include "QuickView.h"

typedef itk::RGBPixel<unsigned char> PixelType;
typedef itk::Image<PixelType, 2> ImageType;

static void CreateImage(ImageType::Pointer image);

int main(int argc, char *argv[]) {
    ImageType::Pointer image = ImageType::New();
    std::stringstream desc;

    itk::FixedArray<bool, 2> flipAxes;
    flipAxes[0] = false;
    flipAxes[1] = false;
    if (argc > 1) {
        typedef itk::ImageFileReader<ImageType> ReaderType;
        ReaderType::Pointer reader = ReaderType::New();
        reader->SetFileName( argv[1] );
        reader->Update();
        image = reader->GetOutput();
        desc << itk::sys::SystemTools::GetFilenameName(argv[1]);
    }
    desc << itk::sys::SystemTools::GetFilenameName(argv[1]);
```
if (argc > 2)
{
    flipAxes[atoi(argv[2])] = true;
}
if (argc > 3)
{
    flipAxes[atoi(argv[3])] = true;
}
else
{
    CreateImage(image);
    desc << "Synthetic image";
    flipAxes[1] = true;
}

typedef itk::FlipImageFilter <ImageType> FlipImageFilterType;
FlipImageFilterType::Pointer flipFilter = FlipImageFilterType::New ();
flipFilter->SetInput(image);
flipFilter->SetFlipAxes(flipAxes);

QuickView viewer;
viewer.AddImage(
    image.GetPointer(),
    true,
    desc.str());
std::stringstream desc2;
desc2 << "Flip, flipAxes= " << flipAxes;
viewer.AddImage(
    flipFilter->GetOutput(),
    true,
    desc2.str());
viewer.ShareCameraOff();
viewer.Visualize();
return EXIT_SUCCESS;

void CreateImage(ImageType::Pointer image)
{
    // Create an image with 2 connected components
    ImageType::RegionType region;
    ImageType::IndexType start;
start[0] = 0;
start[1] = 0;

ImageType::SizeType size;
unsigned int NumRows = 200;
unsigned int NumCols = 300;
size[0] = NumRows;
size[1] = NumCols;

region.SetSize(size);
region.SetIndex(start);
image->SetRegions(region);
image->Allocate();

// Make a square
for(unsigned int r = 20; r < 80; r++)
{
    for(unsigned int c = 20; c < 80; c++)
    {
        ImageType::IndexType pixelIndex;
        pixelIndex[0] = r;
        pixelIndex[1] = c;
        image->SetPixel(pixelIndex, 255);
    }
}
B.2 2D Registration Testing

The base model for 2D registration along with possible optimizer, metric, and transform substitutions will be shown here.

Figure B.1: Base model for 2D image registration.
Figure B.2: 2D image registration with amoeba optimizer outlined.

Figure B.3: 2D image registration with centered similarity 2D transform outlined.
Figure B.4: 2D image registration with gradient difference metric outlined.
B.3 Filter Testing

B.3.1 Smoothing

Here all of the outputs when applying the smoothing filters to Figure 5.17(a) will be shown.

Figure B.5: Smoothing filters applied to brain proton density slice (Figure 5.17(a)); (a) effect of DiscreteGaussianImageFilter; (b) effect of MeanImageFilter; (c) effect of MedianImageFilter; (d) effect of BilateralImageFilter; (e) effect of BinomialBlueImageFilter; (f) effect of RecursiveGaussianImageFilter.
B.3. FILTER TESTING

B.3.2 Thresholding

Here all of the outputs when applying the thresholding filters to Figure 5.17(a) and Figure 5.19 will be shown.

Figure B.6: Thresholding filters applied to applied to Figure 5.19 ((a) and (b)) and applied to Figure 5.17(a) ((c) through (f)); (a) effect of BinaryContourImageFilter; (b) effect of LabelContourImageFilter; (c) effect of OtsuThresholdImageFilter; (d) effect of OtsuMultipleThresholdsImageFilter; (e) effect of ThresholdImageFilter; (f) effect of BinaryThresholdImageFilter.
B.3.3 Edge Detection

Figure B.7: Edge Detection filters applied to CT head slice (Figure 5.17(b)); (a) effect of DerivativeImageFilter; (b) effect of LaplacianImageFilter; (c) effect of LaplacianSharpeningImageFilter; (d) effect of SobelEdgeDetectionImageFilter; (e) effect of CannyEdgeDetectionImageFilter.