Software Requirements Specification

Matthias Vogelgesang, Tomas Farago

Tuesday 20th November, 2012

Contents

Chapter 1

Introduction

1.1 Purpose

This Software Requirements Specification (later just SRS) document is intended to serve as a fundamental “UFO framework” specifications document used for designing the system. It identifies the needs of an online data processing pipeline used in scientific environment, its optimization and interface to the control system of such environment. Framework verification and validation will be carried out based on this document.

The document will be used by the framework developers, control system integrators and testers responsible for verification and validation of the developed system.

1.2 Scope

“UFO framework” is a software system to be developed as a part of the BMBF *UFO project* (Ultra fast X-ray imaging of scientific processes with on-line assessment and data driven process control).

The system targets scientific community using imaging methods. The community uses standard image processing techniques (tomographic reconstruction) and develops new image processing methods for new applications. The future applications cannot be foreseen. Thus, the system to be developed needs to provide fast implementations of standard image processing routines, and simultaneously needs to be highly modular in order to provide an easy-to-use framework for developing new image processing methods. The new methods will be developed by connecting basic building blocks provided by the framework. These blocks are simple mathematical operations or image processing routines – *filters*. The filters which are connected form a data processing pipeline (a graph structure), which will be executed to perform the desired functionality.

The system is essential in order to perform high-speed imaging experiments, build a high-throughput experimental station and perform image-driven feedback for controlling the experimental station and the studied process on-line.

1.3 Definitions, Conventions, Acronyms, and Abbreviations

Let us define the following:

- A base type `T` that specifies a set of valid values
- A *digital image* `I` as an n-dimensional data structure ($n \in \mathbb{N}_0$) over a base type `T` with the following operations:
  - `create :  Integer`$^n \rightarrow$ `I`
  - `delete :  I`
  - `dimensions :  I` $\rightarrow$ `Integer` (number of dimensions)
  - `size :  I` $\rightarrow$ `Integer`^{n} (size in every dimension)
  - `get :  I` $\times$ `Integer`$^n \rightarrow$ `T` (gets pixel value)
  - `set :  I` $\times$ `Integer`$^n\times$ `T` (sets pixel value)

In the remainder of this document we will simply use *image* instead of a *digital image* for briefness. If two or more digital images appear in a definition, then their dimensions and sizes in all dimensions are considered to be the same unless explititly stated otherwise.

1.4 Overview

In section 2, general functionality description is provided with characterization of the targeted user community. Specific requirements on the system are discussed in section 3, first on a higher level describing graph and filters creation, then the essential filters for building new image processing methods are described. Software and user interfaces are discussed at the end of this document.

Chapter 2

General Description

2.1 Product Perspective

The UFO framework can be used in two ways: independently on its own or as a service from external systems. The standalone use case is mandatory for developing, debugging and testing purposes. The other use case will be a control system that integrates the UFO framework so that it can start a computation whenever experiment conditions are met. This will then depend on the actual application and cannot be defined any further.

2.2 Product Functions

2.2.1 Core Functionality

The core of the UFO framework is a set of interdependent classes that provide the basic building blocks to build *graphs* of dynamically loadable *filter nodes* and access *hardware* as well as *external components* such as kernel files. Furthermore, a replaceable *scheduler* module that orchestrates the execution of the filters according to resource availability will be provided.

It will be possible to change the graph structure and save the history of such changes, execute the graph in a "verbose" mode, providing detailed information about each of the filter inputs, outputs and execution status. Logging will be implemented for fast debugging purposes. The user will be able to specify the precision of the calculations to improve performance by setting the execution mode of the graph to *full* or *limited* precision in which case faster hardware operations will be used. It will be possible to "group" filters into a new filter and make *parameter scans* (see Sections 3.1.3 and 3.1.6), which means that users will be able to provide a set of values for every filter's parameter and execute the graph iteratively with parameter changes mapped to respective iterations. This way the users will be able to easily review changes in results based on parameter alternations.

The core objects will be mapped to a standard API that can be used stand-alone or in a third-party application through various language bindings. For interoperability between different languages, an on-disk representation of a graph instance can be stored and read.

Remote access to the computational resources will be provided. A TANGO [1] interface will be provided to integrate the framework into the experimental station's control system.

2.2.2 Filter Functionality

Filters developed as a part of the framework will have a clear description of their inputs and outputs, so the user will be able to connect them to a graph structure properly. Each of the paths in the graph will always start with a filter called *source*. It will provide an access point to either online (camera stream) or offline (data storage) data. The last filter of each path in the graph will be a *sink*, which will write the data to a permanent storage. The actual image processing will be performed by filters between source and sink filters. Users will be able to modify filters behavior by setting parameters specific for every one of them.

Each filter in the graph has an arbitrary, pre-defined number of inputs and outputs. Each output must be connected to an input of another filter in the graph. Termination of data flow is guaranteed with filters that do not provide an output. Thus the computation graph is a *directed acyclic multigraph*. For more information about filters see section 3.1.1.

2.3 User Characteristics

The targeted community is the X-ray imaging scientific community. The majority of users in this community develops new methods using Matlab® or other high-level language, meaning they are used to some for them convenient functionality of the programming language they use. Therefore, the useful functionality must be identified and provided also by the framework for easier crossover from high-level languages to the new system.

Chapter 3

Specific Requirements

3.1 Functional Requirements

3.1.1 Filters

A filter is a multi-ported node in a graph. Each node can have zero to N input *ports* and zero to M output ports to model different processing modalities. Therefore, a filter can be classified as either a *source* without inputs, a *sink* without outputs or an arbitrary processing filter.

A source filter must implement a *generate* method that produces output for each output port, a sink filter must implement a *consume* method that accepts input on each input port and an arbitrary filter must implement a *process* method that accepts input on each input port and produces output on each output port.

A special variant of a processing filter is a *reduction* filter. This filter must implement a *collect* method that accepts inputs on each input port. After a specified number of input data has been collected, it is *reduced* to one or more output data items. For example an averaging filter would produce an averaged image of the whole (finite) input stream.

Each port is restricted to accept or produce images with a specific image dimensionality n. The methods are implemented either on CPU or GPU but filters are free to implement methods on both architectures.

The primary data transfer between two filters happens through memory buffers. Another type of data are typed *properties* that belong to a filter. Users can change these before execution to parametrize the computation or bind two properties of two different filters together. A filter can then listen to changes to its own properties or wait until a specified condition is satisfied.

Starting with section 3.1.8, we will describe particular filters which are guaranteed to be implemented and integrated into the framework. We describe the filters based on their complexity. Common or simple to understand filters are described by their inputs and outputs. More complicated filters are first described in general and then, if needed for clarity, the respective parts of the filter are described (Inputs, Processing and Outputs).

3.1.2 Filter Creation

The exact set of existing filters cannot be pre-determined, therefore there must be a mechanism to construct a filter from a description.

Inputs A human-readable, unique string identifier.

Processing Set up everything that is necessary for a filter to be in a valid state.

Outputs A new filter.

3.1.3 Filter Composition

Because several filters are very basic building blocks, restating them over and over again when defining the graph is time-consuming and error prone. Therefore, there will be a mechanism that allows users to compose a list of filters into a larger single filter block.

Inputs A list of filters.

Outputs A new filter that contains the input filters.

3.1.4 Graph Creation

A graph consists of nodes that are connected by directed edge relations. Each edge specifies the data flow from one node's output port to another node's input port.

Inputs Producing filter, producer's output port, consuming filter and consumer's input port.

Processing Store the connection information.

3.1.5 Graph Serialization

To exchange graph information between different languages and keep configuration stored for a longer term, the graph must be serialized and written to disk.

Inputs A graph object

Processing Save nodes and their current property values as well as the relationship between them.

Outputs A JSON compliant configuration[1]

3.1.6 Graph Execution

The graph must be executed by the user when it is completely defined.

In order to evaluate the effect of different parameters values assigned to the individual filters, it must be possible to execute it repeatedly with a different parameter combination in a parameter scan fashion. It is up to the user and filters to define metrics and to interpret outputs.

Mandatory inputs A graph object

Optional inputs Ranges of valid parameter values

Outputs No immediate outputs apart from what the filter chain defines.

3.1.7 Logging

All events concerning the operation must be logged to a user-specified location such as a file or the terminal. The output must be classified and routed according to the log messages' severance:

- Mandatory messages
- Optional information
- Debug messages
- Warnings

[1]Specification of the format here?

- Errors

Moreover, the output must be classified according to its origin:

- Core component
- Specific filter
- External, third-party dependency

3.1.8 Basic Arithmetic

Basic arithmetic will be implemented to provide pointwise elementary operations on images. *Operation* is one of the $+, -, \cdot, \div, x^n, \sqrt[n]{x}$. Two variants for all operations will provided:

- `I` $\times$ `I` $\rightarrow$ `I`, in this case an *operation* is applied to all respective pixels in the images
- `I` $\times$ `T` $\rightarrow$ `I`, an *operation* is applied to all pixels in the image and a base type value

3.1.9 Relational Operations

Relational operations are defined pointwise with result `True` or `False` for every pixel in an image, an *operation* is one of the $=, <, \leq, >, \geq$. Two variants will be provided:

- `I` $\times$ `I` $\rightarrow$ `I`, in this case an *operation* is applied to all respective pixels in the images
- `I` $\times$ `T` $\rightarrow$ `I`, an *operation* is applied to all pixels in the image and a base type value

The resulting image has as its base type `Bool`.

3.1.10 Bitwise Operations

Bitwise *operations* NOT, AND, OR, XOR on images will be provided. Two variants will be implemented:

- `I` $\times$ `I` $\rightarrow$ `I`, in this case an *operation* is applied to all respective pixels in the images
- `I` $\times$ `T` $\rightarrow$ `I`, an *operation* is applied to all pixels in the image and a base type value

3.1.11 Standard Statistical Operations

The following standard statistical operations will be implemented as standalone filters:

- mean
- median
- min
- max
- variance
- standard deviation

All of the above mentioned operations are defined as `I` $\rightarrow$ `T`.

3.1.12 Histogram

Histogram is a filter which counts number of repetitions of grey values in an image. The output takes into account number of “bins” into which we want to divide the intensities. The filter is defined as `I` $\times$ `Integer` $\rightarrow$ `A`.

Inputs An image, an integer representing the number of bins.

Processing The input image intensities are split into intervals depending on the number of bins. All pixels with intensities in a particular interval are then counted and stored in the output array.

Output Let $e_{min} \in T$ be the smallest grey value in the input image, $e_{max} \in T$ the maximum one. The interval width is then defined as $w = \frac{e_{max}-e_{min}}{b}$. The output is then an array ordered by intensity intervals $A = \{x_0, ..., x_{b-1} \mid i \in \texttt{Integer}, x_i \in \texttt{Integer}$ is the number of pixels with grey values in interval $\langle e_{min} + wi, e_{min} + w(i+1))$ if $i \neq b-1$, $\langle e_{min} + wi, e_{min} + w(i+1)\rangle$ otherwise$\}$.

3.1.13 Determination of the Rotation Center

Tomographic data sets consist of images taken from different sample angles. The sample can be misaligned from the center of the FOV. For some reconstruction algorithms to work, the center of rotation must be computed prior to the reconstruction itself. The filter will compute the center of rotation from a set of images and is defined as $\mathtt{I}^n \to \mathtt{Integer}^2$.

Inputs A sequence of images, where all of them are 2-dimensional.

Outputs An (x, y) tuple representing the position of the center of rotation.

3.1.14 Padding

Symmetric, periodic and constant padding will be supported. Symmetric and periodic padding are described as $\mathtt{I} \times \mathtt{Integer}^n \to \mathtt{I}$. Constant padding is described as $\mathtt{I} \times \mathtt{Integer}^n \times \mathtt{T} \to \mathtt{I}$.

Inputs An image, new sizes of the image in every dimension, fill value (only constant case).

Processing Symmetric and periodic padding use patches of the original image as fill values while the constant case uses a defined base type value.

Outputs If the input image is I_0, output image is I_1, $\boldsymbol{n}_0 = \mathtt{size}(I_0)$, $\boldsymbol{n}_1 = \mathtt{size}(I_1) \Rightarrow \forall i \in \mathtt{Integer}, i < \mathtt{dimensions}(I_1) : \boldsymbol{n}_{0_i} \leq \boldsymbol{n}_{1_i}$

3.1.15 Cropping

Image cropping is described as $\mathtt{I} \times \mathtt{Integer}^n \to \mathtt{I}$.

Inputs An image, new sizes of the image in every dimension.

Outputs If the input image is I_0, output image is I_1, $\boldsymbol{n}_0 = \mathtt{size}(I_0)$, $\boldsymbol{n}_1 = \mathtt{size}(I_1) \Rightarrow \forall i \in \mathtt{Integer}, i < \mathtt{dimensions}(I_1) : \boldsymbol{n}_{0_i} \geq \boldsymbol{n}_{1_i}$

3.1.16 Binning

Image binning is a form of quantization and is defined as $\texttt{I} \times \texttt{Integer}^n \rightarrow \texttt{I}$.

Inputs An image, number by which the respective axis length will be divided.

Processing The resulting image I_{res} is composed of the averages of respective ranges from the input image.

Outputs If the input image is I_0, the second argument is $\boldsymbol{b} = (b_0, ..., b_{n-1})$, output image is I_1, $\boldsymbol{n}_0 = \texttt{size(}I_0\texttt{)}$, $\boldsymbol{n}_1 = \texttt{size(}I_1\texttt{)} \Rightarrow \forall i \in \texttt{Integer}, i < \texttt{dimensions(}I_1\texttt{)} : \boldsymbol{n}_{1_i} = \frac{\boldsymbol{n}_{0_i}}{\boldsymbol{b}_i}$.

3.1.17 Affine Transformations

Affine transformation is defined as $\texttt{I} \times \texttt{I} \rightarrow \texttt{I}$.

Inputs The first argument is a d-dimensional image to be transformed and the second argument is a 2-dimensional image representing the transformation matrix A and $\texttt{size(}A\texttt{)} = (d+1, d+1)$

Processing the transformation matrix is applied to the input's image coordinates.

Outputs The transformed image.

3.1.18 Fourier Transformation

Fourier transformation is an important tool in image processing. Therefore, three filters providing forward, inverse and shift will be implemented.

The filters are defined as follows:

- forward transform – $\texttt{I} \rightarrow \mathcal{F}(\texttt{I})$
- inverse transform – $\mathcal{F}(\texttt{I}) \rightarrow \texttt{I}$
- shift – $\mathcal{F}(\texttt{I}) \rightarrow \mathcal{F}(\texttt{I})$ which swaps quadrants of the transformation so that low frequencies are located in the center of the image

The $\mathcal{F}(\texttt{I})$ notation is used in this section for clarity reasons. In practice, an image in Fourier domain is an image as defined in chapter 1.3 with complex base type.

3.1.19 Bandpass Filter

Bandpass filter operates in the fourier domain. It processes some frequencies of the image and is defined as $\texttt{I} \times \texttt{Float} \times \texttt{Enum} \rightarrow \texttt{I}$

Inputs An image, attenuation coefficient, filter profile $\texttt{E} = \{$Step, Gaussian, Sine, Polynomial, Butterworth$\}$.

Outputs Image with removed specified frequencies.

where the third argument is an enumeration E of possible filter profiles, $\texttt{E} = \{$Step, Gaussian, Sine, Polynomial, Butterworth$\}$.

3.1.20 Phase Retrieval

Phase retrieval filter will be used to deal with the phase problem. It is defined on 2D images as $\texttt{I}^m \rightarrow \texttt{I}^n, m, n \in \mathbb{N}, m \geq n$. The following algorithms for this filter will be implemented:

- Paganin
- Moosmann/Hoffmann

3.1.21 Grating Interferometry

Grating interferometry is used for imaging in multiple contrasts, such as absorption, phase and dark-filed imaging (scattered radiation). It is based on diffraction by X-ray grating, its self-imaging effect and Moire pattern recognition. So-called phase-stepping technique is applied when all three contrast functions shall be obtained. The method is based on acquisition of images at the different gratings (lateral) position with and without the sample. With equidistant steps analysis of the data can be performed via FFT. The algorithm is defined as $\texttt{I}^m \times \texttt{I}^n \rightarrow \texttt{I}^3, m, n \in \mathbb{N}, n > 3, m = n$, $\texttt{I}^m$ are flat fields and $\texttt{I}^n$ are side steps. All images used by this method are 2D images.

3.1.22 Phase Unwrapping

Phase extracted by the phase retrieval filter suffers from 2π-based jumps (wrapped phase) which causes problems in further image processing. Therefore, an "unwrapping" technique which resolves the continuous phase must be employed before further processing. The filter is defined as $\texttt{I} \rightarrow \texttt{I}$ for 2D images.

3.1.23 Convolution

Convolution will be implemented and is defined on parameters $\texttt{I} \times \texttt{I} \rightarrow \texttt{I}$, where number of dimensions and dimension sizes of the two input images may be different. The resulting image dimensions are equal to the biggest of the inputs.

3.1.24 Hot Pixel Filter

"Hot" pixels are the ones with intensities much different from other pixels in a small neighbouring area, thus are considered defective in the respective frame. Filter for removing will be provided and is defined as $\texttt{I} \rightarrow \texttt{I}$.

3.1.25 Reduction

Reduction is an operation which is applied to all values in an image along a specified axis. The filter which will perform such operation is defined as $\texttt{I} \times op \times \texttt{Integer} \rightarrow \texttt{I}$.

Inputs A digital image, operation and axis number. Operation is defined as $op : \texttt{I} \rightarrow T$, where I is a d-dimensional input image.

Processing Apply operation op along a specified axis.

Outputs If I is an n-dimensional image and $n > 0$, the result is an $(n-1)$-dimensional image, 0-dimensional image otherwise.

3.1.26 Interpolation

Various interpolation techniques will be implemented, they are defined as $\texttt{I} \rightarrow \texttt{I}$ and at least the following methods will be provided:

- Nearest neighbour
- Linear
- Quadratic
- Cubic
- Spline

3.1.27 Noise Reduction

Noise is a serious issue in high-speed experiments due to extremely low recording time. The filter for its reduction operates on $\mathtt{I} \rightarrow \mathtt{I}$. Two techniques will be implemented, namely:

- Average
- Mean
- Non-local means
- Anisotropic diffusion

3.1.28 Tomographic Reconstruction

Tomographic reconstruction is one of the key filters in the framework. Several techniques will be implemented. The filter is defined as $\mathtt{I}^n \rightarrow \mathtt{I}$.

Inputs A sequence of 2-dimensional images (radiographs).

Processing Transformation of the input sequence of radiographs into a 3D volume.

Outputs A 3-dimensional image.

At least the following techniques will be implemented:

- Filtered backprojection
- Direct Fourier inversion
- Linogram method
- Algebraic reconstruction

3.1.29 Ring Artifact Removal

Ring artifacts are caused by detector faults and are a common problem for tomographic reconstruction. Therefore we need to provide a way to remove them. The filter for their removal is defined as $\mathtt{I} \rightarrow \mathtt{I}$. Both input and output images are 2-dimensional.

3.1.30 Stripes Removal

This filter will be used to remove stripe-like patterns from images. It is defined as $\mathtt{I} \rightarrow \mathtt{I}$.

3.1.31 Change Detection

Change detection is needed for recognizing events in which the users are interested. It is defined as $\mathtt{I} \times \mathtt{I} \rightarrow \mathtt{Bool}$. If a change is detected, the result is `True`, `False` otherwise.

3.1.32 Object Tracking

Object tracking enables us to track one or more objects in the scene at the same time. It is defined as $\mathtt{I}^n \rightarrow \mathtt{M}^{n-1}$. `M` is an object-sorted set over d-tuples, which represent position changes of all tracked objects.

Inputs A time-ordered image sequence.

Processing Let $\forall i \in \mathtt{Integer}, i \geq 0 \ \wedge \ i < n-1$ be indices into the input sequence, $\boldsymbol{p}_i = (p_0, ..., p_{d-1})$ be a position of an object in image I_i, position $\boldsymbol{p}_{i+1} = (p'_0, ..., p'_{d-1})$ an object's position in image I_{i+1}. Position change for object o is then defined as $\boldsymbol{m} = (p'_0 - p_0, ..., p'_{d-1} - p_{d-1})$ and stored in the position changes set.

Outputs A time-ordered sequence of position changes sets.

3.1.33 Optical Flow

Optical flow is very similar to the object tracking filter but instead of looking for object's position changes, we look for position change for every pixel in an image. The filter is defined as $\mathtt{I}^n \rightarrow \mathtt{M}^{n-1}$. `M` is a digital image over a base type d-tuple representing the position change, which we will call a *dense motion field*.

Inputs A time-ordered image sequence.

Processing Let $\forall i \in \texttt{Integer}, i \geq 0 \ \wedge \ i < n-1$ be indices into the input sequence, I_i and I_{i+1} two consequent input images and $\boldsymbol{x} = (x_0, ..., x_{d-1})$ an index into the image I_i. Optical flow then finds an index $\boldsymbol{x}' = (x'_0, ..., x'_{d-1})$ into the image I_{i+1}, such that $\texttt{get(}I_i, \boldsymbol{x}\texttt{)} = \texttt{get(}I_{i+1}, \boldsymbol{x}'\texttt{)}$. The algorithm then performs a $\texttt{set(}M_i, \boldsymbol{x}, \boldsymbol{x}' - \boldsymbol{x}\texttt{)}$.

Outputs A time-ordered sequence of dense motion fields.

3.2 Performance Requirements

Performance requirements on the described system arise from the experiments which will be conducted within the UFO project. High sample throughput and high data throughput experiments both require fast image processing in order not to cause a time bottleneck during an X-ray experiment.

High sample throughput experiments require fast image analysis which is later used by control mechanisms moving the experimental station's hardware setup. Thus the required limit on the computational time of the image analysis algorithms developed within the framework is to be less or equal to the control mechanisms response time plus the movement of hardware equipment.

High data throughput experiments require fast reconstruction and analysis algorithms in order to provide a preview shortly after data acquisition. From this point of view we require image analysis algorithms to finalize computation before the next dataset is downloaded from a camera. As an example may serve a tomographic experiment where we can reconstruct tomogram i while tomogram $i+1$ is being recorded by the camera. When the data with tomogram $i+1$ is downloaded the recontruction of tomogram i must be finished in order to be able to start with reconstruction of tomogram $i+1$.

3.3 External Interface Requirements

3.3.1 User Interfaces

A GUI for graph an filters manipulation will be provided. Functionality covered by the GUI is the following:

- Graph manipulation (create, delete)
- Filters manipulation (add to graph, delete from graph, connect to others, group more filters to a new "compound" filter)
- Graph execution (run, pause, resume, stop)
- Filter parameters scans (graph run for different filter parameters), users will provide parameter ranges for particular runs
- History access
- Data visualization

3.3.2 Hardware Interfaces

The UFO framework must run on commodity PC hardware with standard x86 (32- or 64-bit) CPUs and OpenCL compliant GPUs from NVIDIA and AMD. Each filter executes its code on a designated CPU or GPU.

3.3.3 Software Interfaces

TANGO interface will be provided in order to integrate the system into an existing control system of an experimental station. The interface specification will be made in collaboration with IT department of ANKA synchrotron which needs to provide requirements on such an interface. The implementation will be based on that specification.

Bibliography

[1] *The TANGO Control System Manual Version 8.0*. The TANGO Team. 2012. URL: http://www.tango-controls.org/Documents/tango-kernel/copy_of_index_html.