Voreen ITK-Wrapper Manual

Mathias Dehne

Maike Dudek

Revision History

12.10.2011	1.0

Table of Contents

1. Introduction

The Insight Segmentation and Registration Toolkit (ITK)[1] is an open-source software toolkit containing an extensive suite of software tools for image analysis. It provides the developer with a large number of image-to-image filters ranging from basic filtering operations to more advanced segmentation- and registration frameworks. The ITK-Wrapper was developed as an easy to use interface for integration of these filters into Voreen[2] project. To achieve this the ITK-Wrapper provides the functionality to deserialize special XML config files containing ITK filter definitions to filter objects. These filter objects are then used to automatically generate the actual code for the corresponding Voreen processor where in- and outputs of an ITK filter are mapped to Voreen ports and the attributes of a filter are mapped to Voreen properties.

2. Getting Started

The ITK-Wrapper is located in "`apps/itk_wrapper`" in the voreen directory. It contains:

- a Qt-project-file "`itk_wrapper.pro`"
- the ITK-Wrapper `.cpp`- an `.h`- files "`itk_wrapper`", "`baseclasses`" and "`template`"
- the `xml_Files` directory, containing the XML files where the ITK filter-definitions are stored
- the `template` directory, with the general template text files for the Voreen processor generation
- the `specialFilters` folder storing filters that cannot be generated automatically

By default all XML files in the directory `xml_Files` are processed by the wrapper, generating all predefined filters. Additionally It is possible to use only a specific set of XML files. An example on the actual use of this option can be found in the main method of `itk_wrapper.cpp`. The generated Voreen processors are copied to "`module/itk_generated/processors`" in the voreen directory .

Apart from the processor files the wrapper generates the files `itk_generatedmodule.cpp`, `itk_generated_core.pri` and the `itk_generated_module.xml`.

[1] http://www.itk.org
[2] http://www.voreen.org

In order to use the generated ITK processors the `itk_generated` module needs to be included. The following line has to be added to the Voreen config.txt:

```
VRN_MODULES += itk_generated
```

Note: `itk_generated` module depends on `itk` module. Thus activation of the itk module in the `config.txt` is mandatory for using the itk_generated module.

3. Adding new ITK-filters

Now you know how to generate the Voreen processors representing the already defined ITK filters with the ITK-Wrapper. But what about adding new filters?

To add a new ITK filter you have to define the filter in its module XML file. To understand the XML filestructure it is described in the next section.

Section 3.2 shows some examples of already added filters and section 3.3 explains how to add a more difficult filter which cannot be generated automatically by the wrapper (special filter).

3.1. XML-File-Structure

In order to find a specific ITK filter in Voreen, we used the ITK-Doxygen[3] structure. All filters are subsumed under different groups which contain different modules. For example, there is the group "Filtering" which contains the modules "ITKAnisotropicSmoothing", "ITKImageFeature" and so on. Thus each XML file in the `xml_Files` directory of the ITK-Wrapper represents one module of the ITK-Doxygen structure. (The group to which a module belongs is defined in the associated XML file itself).

The inner structure of such an XML file is structured as followed:

First the **xml version** and the **VoreenData** version need to be declared:

```
<?xml version="1.0" ?>
<VoreenData version="1">
```

After that the **ITK_Module** is defined through a **name** ("itk_" followed by the name of the module, e.g. "AnisotropicSmoothing"), a **group** (e.g. "Filtering") and a **filterlist** containing the **filters** of the module:

```
  <ITK_Module name="itk_AnisotropicSmoothing" group="Filtering">
    <filterlist>
      <filter ...>
      </filter>
      ...
    </filterlist>
  </ITK_Module>
</VoreenData>
```

Each **filter** element is then described through a number of XML attributes and -child elements:

name
: Name of the filter, e.g. "CurvatureAnisotropicDiffusionImageFilter"

  Note: The name must be exactly the name of the ITK filter, otherwise it will not work!

enabled (optional)
: Boolean to enable or disable a filter. Default value is *true*.

[3] http://www.itk.org/Doxygen/html/modules.html

description (optional)

If a filter is disabled the description attribute can be set to describe why the filter is disabled. Otherwise the description is taken as the processor description in Voreen. If the filter is enabled and no description is set, the wrapper will generate a link to the ITK Doxygen page of the filter (by using the name of the filter) as processor description.

autoGenerated (optional)

Boolean whether the filter can be auto generated by the wrapper or not (see "3.3 Special Filters"). Default value is *true*.

codeState (optional)

Code state of a filter.

Possible values are *EXPERIMENTAL*, *TESTING* and *STABLE*. Default value is *EXPERIMENTAL*.

inports (optional)

Child element containing the inports / inputs of a filter. See **port**.

Note: By default a filter has one inport and one outport, which support all scalar volume types. It is only necessary to set the ports of a filter if it has more or less than one in- and/or outport or if the port supports only specific volume types. If you set one port of the filter (inport or outport) you have to set all ports of the filter.As an example: If you have a filter which has volume type constraints for the inport you nevertheless have to set the outport.

outports (optional)

Child element of a **filter** containing the outports / outputs of a filter. See **port**.

Note 1: By default a filter has one inport and one outport, which allow all scalar volume types. So you only have to set the ports of a filter if it has more or less than one in- or outport or if there are only special volume types supported for the port. But if you set one port of the filter (inport or outport) you have to set all ports of the filter! So if there is for example a filter which has volume type constraints only on the inport you nevertheless have to set the outport.

Note 2: You can only set one type for the possibleTypes of an outport (e.g. *UInt8*). If there is a volume type constraint on the inport, for example *IntegerScalar,* and you want the output of the outport to be from the same type as the input of the inport you don't have to set the possibleTypes of the outport. This ensures that the output is of the same type as the input.

port

Child element of **inports** element or **outports** element containing the attributes and child elements:

name

Name of an inport or outport of a filter, e.g. "InputImage"

filterTemplate (optional)

Boolean defining whether the port is a template argument of the filter or not. Default value is *true*.

This attribute is necessary if for example a filter has two inports and one outport but only one template argument because all ports have to be from the same volume type.

nameIsSetter (optional)

Boolean defining whether the name of the port is the filter input setter of or not. Default value is *false*.

Usually the inputs of a filter are set with "SetInput1(...);", "SetInput2(...);" and so on (or only with "SetInput(..);" if there is only one input) but sometimes a special name is needed to set an input, e.g. "SetMaskImage(...)". In this case the name of the inport (here "MaskImage") can be used to set the input.

possibleTypes

Child element of a **port** containing the possible volume types of the port.

Possible values for a volume **type** are *UInt8*, *Int8*, *UInt16*, *Int16*, *UInt32*, *Int32*, *UInt64*, *Int64*, *Float*, *Double*, *4xUInt8*, *4xInt8*, *4xUInt16*, *4xInt16*, *4xFloat*, *4xDouble*, *3xUInt8*, *3xInt8*, *3xUInt16*, *3xInt16*, *3xFloat*, *3xDouble*, *2xUInt8*, *2xInt8*, *2xUInt16*, *2xInt16*, *2xFloat*, *2xDouble*, or the metaTypes *Scalar* (which contains all scalar volume types apart from *UInt64* and *Int64*), *IntegerScalar* (which contains all scalar integer volume types apart from *UInt64* and *Int64*), *RealScalar* (which contains all scalar real volume types), *Vector* (which contains all vector volume types), *IntegerVector* (which contains all vector integer volume types) and *RealVector* (which contains all vector real volume types).

attributes (optional)
Child element of a **filter** containing the attributes of a filter:

attribute
Child element of **attributes** element containing:

name
Name of the attribute, e.g. "NumberOfIterations"

argumenttype
Type of the argument.

Possible values are:

- *Int*, *Float* and *Bool*: Represented through an Int-, a Float- or a BoolProperty in Voreen.
- *KernelType*: Used if a filter has a TKernel as template-argument. This means, that it needs a structuring element as kernel.

  The ITK-Wrapper then generates a processor where you can choose one of the existing structuring elements through a StringOptionProperty.
- *SizeType* and *IndexTxpe*: Represented through an IntVec3Property in Voreen.
- *ArrayType*: Represented through a FloatVec3Property in Voreen.
- *PixelType*: Represented through a VoxelTypeProperty in Voreen.
- *SetSeedType*: Only used if a filter needs a seed point and has no AddSeed-function!

  The ITK-Wrapper then generates a processor with a GeometryPort as an additional inport through which seed points can be added. (To do this the SeedpointGenerator- or the VolumePicking processor can be used.) The seed point which should be used as the filter's seed point can be chosen through an IntProperty.
- *AddSeedType*: Used if a filter needs one ore more seed point(s).

  The ITK-Wrapper then generates a processor with a GeometryPort as additional inport through which seed points can be added. (To do this the SeedpointGenerator- or the VolumePicking processor can be used.) All added seed points will be added to the filter as well but some filters only use the first added seed point.
- *VectorSeedType*: Used if a filter needs a vector of seed points and has no AddSeed-function.

  The ITK-Wrapper then generates a processor with a GeometryPort as additional inport through which seed points can be added. (To do this the SeedpointGenerator- or the VolumePicking processor can be used.)

inputArgument (optional)
Boolean defining whether the argument is an input- or an output-argument of the filter. Default value is *true*.

defaultValue
Default value of the argument. Only needed for the argumenttypes *Int* (e.g. defaultValue="1"), *Float* (e.g. defaultValue="1.0f"), *Bool* (e.g. defaultValue="false"), *SizeType* and *IndexType* (e.g. defaultValue="(1)") and *ArrayType* (e.g. defaultValue="(1.0f)"). All other argumenttypes don't have a defaultValue.

minValue
Minimum value of the argument. Only needed for the argumenttypes *Int*, *Float*, *SizeType*, *IndexType* and *Array Type* (examples see: defaultValue). For the argumenttype *PixelType* the minValue is optional (example: value 0 is not allowed).

maxValue
Maximum value of the argument. Only needed for the argumenttypes *Int*, *Float*, *SizeType*, *IndexType* and *Array Type* (examples see: defaultValue). For the argumenttype *PixelType* the maxValue is optional like the minValue.

3.2. Examples

This section shows some examples of filters which can be generated automatically by the ITK-Wrapper. Apart from these examples you can find other filter examples in the XML files of the `xml_Files` directory where over 150 filters are already defined.

3.2.1. Example 1: AbsImageFilter - A very simple filter

One of the simplest filters is the AbsImageFilter[4] which can be found in "itk_ImageIntensity.xml" in the `xml_Files` directory. This filter has one standard inport, one standard outport and no attributes. Therefore it can be simply defined like this:

```
<filter name="AbsImageFilter" codeState="STABLE">
</filter>
```

3.2.2. Example 2: AndImageFiler - A simple filter with two inports and volume type constraints

The AndImageFilter[5] also can be found in "itk_ImageIntensity.xml" in the xml_Files directory. It has two inports and one outport which only allow scalar integer volume types and no attributes.

```
<filter name="AndImageFilter" codeState="STABLE">
  <inports>
    <port name="InputImage1">
      <possibleTypes>
        <type value="IntegerScalar"/>
      </possibleTypes>
    </port>
    <port name="InputImage2">
      <possibleTypes>
        <type value="IntegerScalar"/>
      </possibleTypes>
    </port>
  </inports>
  <outports>
    <port name="OutputImage"/>
  </outports>
</filter>
```

3.2.3. Example 3: CheckerBoardImageFilter - A filter where not all ports are template-arguments of the filter

The CheckerBoardImageFilter[6] can be found in "itk_ImageCompare.xml" in the xml_Files directory. It has two inports and one outport but only one template argument because all ports have to be from the same volume type. Therefore the filterTemplate argument needs to be set to false for the second inport and the outport.

```
<filter name="CheckerBoardImageFilter" codeState="STABLE">
  <inports>
    <port name="InputImage1"/>
    <port name="InputImage2" filterTemplate="false"/>
  </inports>
  <outports>
    <port name="OutputImage" filterTemplate="false"/>
  </outports>
</filter>
```

[4] http://www.itk.org/Doxygen/html/classitk_1_1AbsImageFilter.html
[5] http://www.itk.org/Doxygen/html/classitk_1_1AndImageFilter.html
[6] http://www.itk.org/Doxygen/html/classitk_1_1CheckerBoardImageFilter.html

3.2.4. Example 4: GrayscaleGeodesicDilateImageFilter - A filter where the name of the inport is the name of the input-setter of the filter

The GrayscaleGeodesicDilateImageFilter[7] can be found in "itk_MathematicalMorphology.xml" in the xml_Files directory. It has two inports, one outport and two arguments of the type Bool. But this time the names of the inports are needed as setters for the inputs of the filter. Apart from this the second input is no template argument of the filter because it needs to be from the same volume type as the first input.

```
<filter name="GrayscaleGeodesicDilateImageFilter" codeState="STABLE">
  <inports>
    <port name="MarkerImage" nameIsSetter="true"/>
    <port name="MaskImage" nameIsSetter="true" filterTemplate="false"/>
  </inports>
  <outports>
    <port name="OutputImage"/>
  </outports>
  <arguments>
    <argument name="RunOneIteration" argumenttype="Bool" defaultValue="false"/>
    <argument name="FullyConnected" argumenttype="Bool" defaultValue="false"/>
  </arguments>
</filter>
```

3.2.5. Example 5: GrayscaleDilateImageFilter - A filter with a TKernel

The GrayscaleDilateImageFilter[8] can be found in "itk_MathematicalMorphology.xml" in the xml_Files directory. It has one standard inport, one standard outport and three arguments. The special thing about this filter is, that is has a TKernel as template argument. This means, that it needs a structuring element as kernel.

The wrapper therefore generates a processor with a range of different structuring elements which can be chosen by a StringOptionProperty.

```
<filter name="GrayscaleDilateImageFilter" codeState="STABLE">
  <arguments>
    <argument name="Kernel" argumenttype="KernelType"/>
    <argument name="Algorithm" argumenttype="Int" defaultValue="0"
              minValue="0" maxValue="3"/>
    <argument name="Boundary" argumenttype="PixelType"/>
  </arguments>
</filter>
```

3.2.6. Example 6: CannyEdgeDetectionImageFilter - A filter with an ArrayType-argument

The CannyEdgeDetectionImageFilter[9] can be found in "itk_ImageFeature.xml" in the xml_Files directory. It has one inport and one outport with a volume type constraint (Float) and five arguments where three are from the type *PixelType* and two are from the type *ArrayType*.

```
<filter name="CannyEdgeDetectionImageFilter" codeState="STABLE">
  <inports>
    <port name="InputImage">
      <possibleTypes>
        <type value="Float"/>
```

[7] http://www.itk.org/Doxygen/html/classitk_1_1GrayscaleGeodesicDilateImageFilter.html
[8] http://www.itk.org/Doxygen/html/classitk_1_1GrayscaleDilateImageFilter.html
[9] http://www.itk.org/Doxygen/html/classitk_1_1CannyEdgeDetectionImageFilter.html

```
      </possibleTypes>
    </port>
  </inports>
  <outports>
    <port name="OutputImage"/>
  </outports>
  <arguments>
    <argument name="LowerThreshold" argumenttype="PixelType"/>
    <argument name="UpperThreshold" argumenttype="PixelType"/>
    <argument name="OutsideValue" argumenttype="PixelType"/>
    <argument name="Variance" argumenttype="ArrayType" defaultValue="(0.1f)"
              minValue="(0.0f)" maxValue="(0.99f)"/>
    <argument name="MaximumError" argumenttype="ArrayType" defaultValue="(0.1f)"
              minValue="(0.0f)" maxValue="(0.99f)"/>
  </arguments>
</filter>
```

3.2.7. Example 7: ConfidenceConnectedImageFilter - A segmentation-filter which needs a seed point

The ConfidenceConnectedImageFilter[10] can be found in "itk_RegionGrowing.xml" in the xml_Files directory. It has one standard inport, one standard outport and five arguments. To find a segmentation of the input-image the filter needs an initial seed point. Therefore it has an argument from the type *AddSeedType*.

This ensures that the generated processor has a GeometryPort as additional inport through which seed points can be added. To do this for example the SeedpointGenerator- or the VolumePicking processor can be used.

```
<filter name="ConfidenceConnectedImageFilter">
  <arguments>
    <argument name="Seed" argumenttype="AddSeedType"/>
    <argument name="NumberOfIterations" argumenttype="Int" defaultValue="1"
              minValue="0" maxValue="1000"/>
    <argument name="ReplaceValue" argumenttype="PixelType"/>
    <argument name="Multiplier" argumenttype="Float" defaultValue="1.0f"
              minValue="1.0f" maxValue="1000.0f"/>
    <argument name="InitialNeighborhoodRadius" argumenttype="Int" defaultValue="1"
              minValue="0" maxValue="1000"/>
  </arguments>
</filter>
```

3.3. Special Filters

Although the ITK-Wrapper can generate a lot of filters automatically using the allready mentioned methods, there are still some filters which cannot be generated this way because they need for example an special input. This could be for example a special structure which cannot directly be represented by Voreen (e.g. a LabelMap), a special output from another filter or a special function.

This section describes how the ITK-Wrapper can help to create these special filters (see 3.3.1) and how the created filters can then s be integrated into Voreen using the ITK-Wrapper (see 3.3.2).

3.3.1. Pre Generation of a special filter with the ITK-Wrapper

As an example to explain how a special filter can be created we take a closer look at the LabelMapContourOverlayImageFilter[11]. The special thing about this filter is, that it needs a LabelMap[12] as one input which is a special representation of an

[10] http://www.itk.org/Doxygen/html/classitk_1_1ConfidenceConnectedImageFilter.html
[11] http://www.itk.org/Doxygen/html/classitk_1_1LabelMapOverlayImageFilter.html
[12] http://www.itk.org/Doxygen/html/classitk_1_1LabelMap.html

image that is not supported by Voreen. Therefore we need a second filter, e.g the BinaryImageToLabelMapFilter[13], which can convert a "normal" image to a LabelMap and take it's output as input for the LabelMapContourOverlayImageFilter. This is not directly possible with the ITK-Wrapper but we can use the wrapper to pre generate a filter which can then be enhanced.

As a first step we pre-define the filter with all needed ports and attributes in its module XML "ITKImageFusion" and let the wrapper generate a Voreen processor out of this definition:

```
<?xml version="1.0" ?>
<VoreenData version="1">
  <ITK_Module name="ITKImageFusion" group="Filtering">
    <filterlist>
      ...
      <filter name="LabelMapContourOverlayImageFilter">
        <inports>
          <port name="LabelImage">
            <possibleTypes>
              <type value="IntegerScalar"/>
            </possibleTypes>
          </port>
          <port name="FeatureImage" nameIsSetter="true">
            <possibleTypes>
              <type value="IntegerScalar"/>
            </possibleTypes>
          </port>
        </inports>
        <outports>
          <port name="OutputImage">
            <possibleTypes>
              <type value="3xInt16"/>
            </possibleTypes>
          </port>
        </outports>
        <arguments>
          <argument name="Opacity" argumenttype="Float" defaultValue="0.5f"
                    minValue="0.0f" maxValue="1.0f"/>
        </arguments>
      </filter>
      ...
    </filterlist>
  </ITK_Module>
</VoreenData
```

The generated processor-.cpp then looks like this:

- First there are some includes:

```
#include "labelmapcontouroverlayimagefilter.h"
#include "voreen/core/datastructures/volume/volume.h"
#include "voreen/core/datastructures/volume/volumehandle.h"
#include "voreen/core/datastructures/volume/volumeatomic.h"
#include "voreen/core/ports/conditions/portconditionvolumetype.h"
#include "modules/itk/utils/itkwrapper.h"
#include "voreen/core/datastructures/volume/operators/volumeoperatorconvert.h"
#include "itkImage.h"

#include "itkLabelMapContourOverlayImageFilter.h"
```

[13] http://www.itk.org/Doxygen/html/classitk_1_1BinaryImageToLabelMapFilter.html

```
#include <iostream>
```

- Then the two inputs and the output, represented through ports, and the attribute, represented through a property, are defined:

```
namespace voreen {

const std::string LabelMapContourOverlayImageFilterITK::
                  loggerCat_("voreen.LabelMapContourOverlayImageFilterITK");

LabelMapContourOverlayImageFilterITK::LabelMapContourOverlayImageFilterITK()
    : ITKProcessor(),
    inport1_(Port::INPORT, "LabelImage"),
    inport2_(Port::INPORT, "FeatureImage"),
    outport1_(Port::OUTPORT, "OutputImage"),
    opacity_("opacity", "Opacity", 0.5f, 0.0f, 1.0f)
{
    addPort(inport1_);
    PortConditionLogicalOr* orCondition1 = new PortConditionLogicalOr();
    orCondition1->addLinkedCondition(new PortConditionVolumeTypeUInt8());
    orCondition1->addLinkedCondition(new PortConditionVolumeTypeInt8());
    orCondition1->addLinkedCondition(new PortConditionVolumeTypeUInt16());
    orCondition1->addLinkedCondition(new PortConditionVolumeTypeInt16());
    orCondition1->addLinkedCondition(new PortConditionVolumeTypeUInt32());
    orCondition1->addLinkedCondition(new PortConditionVolumeTypeInt32());
    inport1_.addCondition(orCondition1);
    addPort(inport2_);
    PortConditionLogicalOr* orCondition2 = new PortConditionLogicalOr();
    orCondition2->addLinkedCondition(new PortConditionVolumeTypeUInt8());
    orCondition2->addLinkedCondition(new PortConditionVolumeTypeInt8());
    orCondition2->addLinkedCondition(new PortConditionVolumeTypeUInt16());
    orCondition2->addLinkedCondition(new PortConditionVolumeTypeInt16());
    orCondition2->addLinkedCondition(new PortConditionVolumeTypeUInt32());
    orCondition2->addLinkedCondition(new PortConditionVolumeTypeInt32());
    inport2_.addCondition(orCondition2);
    addPort(outport1_);

    addProperty(opacity_);
}

Processor* LabelMapContourOverlayImageFilterITK::create() const {
    return new LabelMapContourOverlayImageFilterITK();
}
```

- After that the most important part follows, the labelMapContourOverlayImageFilterITK method which defines and performs the filter with the transferred template-arguments:

  Here first of all the image types of the two inputs and the output and two ITK SmartPointers which point to the input images are defined.

```
template<class T, class S>
void LabelMapContourOverlayImageFilterITK::labelMapContourOverlayImageFilterITK() {

    typedef itk::Image<T, 3> InputImageType1;
    typedef itk::Image<S, 3> InputImageType2;
    typedef itk::Image<itk::CovariantVector<int16_t,3>, 3> OutputImageType1;

    typename InputImageType1::Pointer p1 = voreenToITK<T>(inport1_.getData());
    typename InputImageType2::Pointer p2 = voreenToITK<S>(inport2_.getData());
```

Then the filter definition follows which will not work like this because the first template argument of the filter needs to be a TLabelMap and not a TImage. Therefore the following SetInput(p1)-command also won't work whereas the second input and the argument of the filter are set correctly.

```
    //Filter define
    typedef itk::LabelMapContourOverlayImageFilter<InputImageType1, InputImageType2,
                                                   OutputImageType1> FilterType;
    FilterType::Pointer filter = FilterType::New();

    filter->SetInput(p1);
    filter->SetFeatureImage(p2);

    filter->SetOpacity(opacity_.get());
```

In the last part the filter is performed and the output is set.

```
    observe(filter.GetPointer());

    try
    {
        filter->Update();
    }
    catch (itk::ExceptionObject &e)
    {
        LERROR(e);
    }

    VolumeHandle* outputVolume1 = 0;
    outputVolume1 = ITKVec3ToVoreenVec3Copy<int16_t>(filter->GetOutput());

    if (outputVolume1)
        outport1_.setData(outputVolume1);
    else
        outport1_.setData(0);
}
```

- After that the processing-methods follow which validate the allowed volume types for the inputs and call the labelMapContourOverlayImageFilterITK method with the appropriate template-arguments.

```
void LabelMapContourOverlayImageFilterITK::process() {
    const VolumeHandleBase* inputHandle1 = inport1_.getData();
    const Volume* inputVolume1 = inputHandle1->getRepresentation<Volume>();

    if (dynamic_cast<const VolumeUInt8*>(inputVolume1))  {
        volumeTypeSwitch1<uint8_t>();
    }
    else if (dynamic_cast<const VolumeInt8*>(inputVolume1))  {
        volumeTypeSwitch1<int8_t>();
    }
    else if (dynamic_cast<const VolumeUInt16*>(inputVolume1))  {
        volumeTypeSwitch1<uint16_t>();
    }
    else if (dynamic_cast<const VolumeInt16*>(inputVolume1))  {
        volumeTypeSwitch1<int16_t>();
    }
    else if (dynamic_cast<const VolumeUInt32*>(inputVolume1))  {
        volumeTypeSwitch1<uint32_t>();
    }
    else if (dynamic_cast<const VolumeInt32*>(inputVolume1))  {
        volumeTypeSwitch1<int32_t>();
    }
    else {
        LERROR("Inputformat of Volume 1 is not supported!");
    }
}

template <class T>
void LabelMapContourOverlayImageFilterITK::volumeTypeSwitch1() {
    const VolumeHandleBase* inputHandle2 = inport2_.getData();
    const Volume* inputVolume2 = inputHandle2->getRepresentation<Volume>();

    if (dynamic_cast<const VolumeUInt8*>(inputVolume2))  {
        labelMapContourOverlayImageFilterITK<T, uint8_t>();
    }
    else if (dynamic_cast<const VolumeInt8*>(inputVolume2))  {
        labelMapContourOverlayImageFilterITK<T, int8_t>();
    }
    else if (dynamic_cast<const VolumeUInt16*>(inputVolume2))  {
        labelMapContourOverlayImageFilterITK<T, uint16_t>();
    }
    else if (dynamic_cast<const VolumeInt16*>(inputVolume2))  {
        labelMapContourOverlayImageFilterITK<T, int16_t>();
    }
    else if (dynamic_cast<const VolumeUInt32*>(inputVolume2))  {
        labelMapContourOverlayImageFilterITK<T, uint32_t>();
    }
    else if (dynamic_cast<const VolumeInt32*>(inputVolume2))  {
        labelMapContourOverlayImageFilterITK<T, int32_t>();
    }
    else {
        LERROR("Inputformat of Volume 2 is not supported!");
    }
}
}   // namespace
```

And the generated processor.h looks like this:

```
#ifndef VRN_LABELMAPCONTOUROVERLAYIMAGEFILTER_H
#define VRN_LABELMAPCONTOUROVERLAYIMAGEFILTER_H

#include "modules/itk/processors/itkprocessor.h"
#include "voreen/core/processors/volumeprocessor.h"
#include "voreen/core/ports/allports.h"
#include <string>

#include "voreen/core/properties/floatproperty.h"

namespace voreen {

class VolumeHandleBase;

class LabelMapContourOverlayImageFilterITK : public ITKProcessor {
public:
    LabelMapContourOverlayImageFilterITK();

    virtual Processor* create() const;
    virtual std::string getCategory() const   {
        return "Volume Processing/Filtering/ImageFusion";
    }
    virtual std::string getClassName() const  {
        return "LabelMapContourOverlayImageFilterITK";
    }
    virtual CodeState getCodeState() const    {
        return CODE_STATE_EXPERIMENTAL;
    }

protected:
    template<class T, class S>
    void labelMapContourOverlayImageFilterITK();

    virtual void process();
    template<class T>
    void volumeTypeSwitch1();

private:
    VolumePort inport1_;
    VolumePort inport2_;
    VolumePort outport1_;

    FloatProperty opacity_;

    static const std::string loggerCat_;
};
}
#endif
```

Now we enhance the processor.cpp by the following lines to make the filter work:

- As mentioned above we need for example the BinaryImageToLabelMapFilter[14] to convert the first input image to a LabelMap. Therefore we need to include this filter first:

```
...
#include "itkLabelMapContourOverlayImageFilter.h"
#include "itkBinaryImageToLabelMapFilter.h"
...
```

- Then this filter must be defined and convert the first input image to a LabelMap:

```
...
    typedef itk::Image<T, 3> InputImageType1;
    typedef itk::Image<S, 3> InputImageType2;
    typedef itk::Image<itk::CovariantVector<T,3>, 3> OutputImageType1;

    typename InputImageType1::Pointer p1 = voreenToITK<T>(inport1_.getData());
    typename InputImageType2::Pointer p2 = voreenToITK<S>(inport2_.getData());

    typedef itk::BinaryImageToLabelMapFilter<InputImageType1> LabelMapType;

                     LabelMapType::Pointer labelMap = LabelMapType::New();
    labelMap->SetInput(p1);
    labelMap->Update();
```

[14] http://www.itk.org/Doxygen/html/classitk_1_1BinaryImageToLabelMapFilter.html

- After that the output of this filter can be used as input for the LabelMapContourOverlayImageFilter:

```
    //Filter define
    typedef itk::LabelMapContourOverlayImageFilter<LabelMapType::OutputImageType,
                                                   InputImageType2,
                                                   OutputImageType1> FilterType;
    FilterType::Pointer filter = FilterType::New();

    filter->SetInput(labelMap->GetOutput());
    filter->SetFeatureImage(p2);

...
```

After these changes the LabelMapContourOverlayImageFilter will work but it is also possible to add more things like for example a read only property that shows how many objects the filter has found in the image for a more comfortable analysis of the filter.

Therefore additionally the following lines can be added to the .cpp-file:

- First the new property-definition:

```
LabelMapContourOverlayImageFilterITK::LabelMapContourOverlayImageFilterITK()
    : ITKProcessor(),
    inport1_(Port::INPORT, "LabelMap"),
    inport2_(Port::INPORT, "FeatureImage"),
    outport1_(Port::OUTPORT, "OutputImage"),
    numObjects_("numObjects", "Number of Objects", 0, 0, 1000),
    opacity_("opacity", "Opacity", 0.0f, 0.f, 1.f)
{
    ...

    addProperty(numObjects_);
    numObjects_.setWidgetsEnabled(false);
    addProperty(opacity_);
}
```

- And then the setting of this property:

```
    ...
    LabelMapType::Pointer labelMap = LabelMapType::New();
    labelMap->SetInput(p1);
    labelMap->Update();
    numObjects_.set(labelMap->GetOutput()->GetNumberOfLabelObjects());
    ...
```

... and to the .h-file:

- The new IntProperty:

```
...

#include "voreen/core/properties/floatproperty.h"
#include "voreen/core/properties/intproperty.h"

namespace voreen {
```

```
class VolumeHandleBase;

class LabelMapContourOverlayImageFilterITK : public ITKProcessor {
...

private:
    VolumePort inport1_;
    VolumePort inport2_;
    VolumePort outport1_;

    IntProperty numObjects_;
    FloatProperty opacity_;

    static const std::string loggerCat_;
};
}
#endif
```

Now the LabelMapContourOverlayImageFilter is complete and works for the moment. But there is still a problem:

- The filter will be overwritten after running the ITK-Wrapper again!

The next section describes how to solve this problem.

3.3.2. Integrating a special filter to Voreen with the ITK-Wrapper

As mentioned in the last chapter we can use the ITK-Wrapper to pre generate a special filter and then enhance this filter. But if we run the wrapper again these enhancements will be lost. To prevent this you have to copy the filter's .cpp- and .h-file to the `specialFilters` directory (see section 2) where all special filters are stored.

Apart from this you have to change the filter's definition in the "ITKImageFusion" xml like this:

```
...
      <filter name="LabelMapContourOverlayImageFilter" autoGenerated="false"

               codeState ="STABLE">
      </filter>
...
```

This has the effect that the ITK-Wrapper copies the processor-files of the filter from the specialFilters directory to it's module directory under "`module/itk_generated/processors`" and adds the filter to the `itk_generatedmodule.cpp`, the `itk_generated_core.pri` and the `itk_generated_module.xml`.

Note: The *codeState* will not be set by the wrapper if the filter's attribute *autoGenerated* is set to *false*. It is just set to have an overview of the code states of a module. So you have to set the codeState in the .h-file of the filter in the specialFilters directory on your own.

After that you can run the ITK-Wrapper again without overwriting the new filter.