ITK semi-automatic wrapper generation for XIP

Julien Gein, SCR-IM
October 2, 2007

1 Purpose

This document describes how to install and run the ITK wrappers generation scripts for XIP [2]. It also introduces the XML file format used to store the information about the ITK classes that need to be wrapped.

2 References

[1] ActivePerl, www.activestate.com
[2] The eXtensible Imaging Platform, https://collab01a.scr.siemens.com/xipwiki/index.php/Main_Page
[3] The Open Inventor Mentor: programming Object-oriented 3D graphics with Open Inventor, release 2, Wernecke Josie.

3 Installation

A PERL installation shall be installed on the target machine. A free version of the PERL interpreter with the PERL binaries and the most common packages, *ActivePerl* is available on the *ActiveState* website [1].

After the installation, either the PATH environment variable or PERL_PATH shall be defined and point at the binary directory of the PERL installation.

4 Running the generation scripts

The ITK wrappers for can be generated using the *wrapitk.bat* executable. It calls internally the *wrapitk.pl* Perl script with the default arguments. The first argument of the Perl script is the path to the XML configuration file (see section 5), which was created at SCR-IM and can be further extended or modified by the user. For each new release of ITK, this XML file shall be reviewed to account for the possible interface changes, potential class removal, and also to add the latest features. The second argument of the script is the output directory where the files shall be generated.
The default arguments are *itk.xml* and *../../itk/database/itk/generated/*.

5 XML configuration file format

The XML file is responsible for storing all the information about the ITK class hierarchy as well as the template configurations for each of the ITK class. The template configurations determine which combinations of template parameters can be used to instantiate one filter. For instance, the *ItkResampleImageFilter* could be instantiated with a 2D image with float representation, or a 3D image with unsigned short representation. In C++, this would translate to:

```
itkResampleImageFilter< itkImage< float, 2 >, itkImage< float, 2 > >
itkResampleImageFilter< itkImage< ushort, 3 >, itkImage< ushort, 3 > >
```

Those configurations cannot be generated automatically. It is the responsibility of the user to tell the generation scripts which configurations shall be available. The convention used in the given XML configuration file is to synchronize the output type with the input type, when possible, in order to avoid a big combinatorial. The XML file format allows the synchronization of a filter input with a given template parameter. This will be detailed in section 5.2.

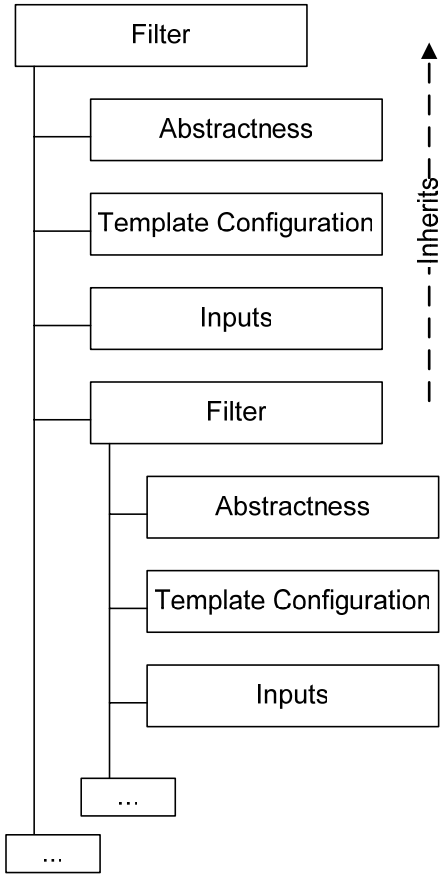

Figure 1 - Illustration of the XML tree structure

The structure of the XML tree is illustrated in Figure 1. The hierarchy of the ITK classes is preserved, meaning a deriving ITK class is declared in the body of the parent ITK

class. The deriving class inherits the inputs and the template configuration defined by the parent class.

5.1 Filter properties

The XML **`filter`** tag corresponds to the declaration of a new filter. The **`id`** attribute shall be set to the ITK class name without the itk suffix, e.g. **`id`**=*ReflectImageFilter*. The ***enabled*** attribute can be used to turn on/off the generation of the corresponding wrapper. If a parent class is disabled, all the deriving classes will be disabled too.

Under a **`filter`** tag, the user can specify the **`abstract`** tag. Wrapped abstract engines only define inputs. They do not have an *evaluate* method (user can refer to [3] for more details).

Under a **`filter`** tag, the user can add as many inputs as required. The **`input`** tag is used to declare an input field, which usually corresponds to a *Set*...method in ITK. It has the following attributes:

- **`id`**, name of the input field corresponding to the suffix of the ITK *Set*...method (e.g. *SetIsoSurfaceValue* becomes *IsoSurfaceValue*)
- **`type`**, type of the input field. It can either be a built-in type (float, unsigned short, string, etc.) or a type corresponding to a template argument (see section 5.2 for more details). In this case the syntax is the following:
  `type`=ARG(TemplateParameterName)
- **`enabled`**, can be used to turn on/off the wrapping of this input field.
- **`value`**, is the default value taken by this input field.
- **`num`**, determines the size of a multiple field.
- **`multiple`**, is deprecated.
- **`access`,** is used to access an internal value of an advanced data type (usually derived from *SoXipData*).
  e.g.

```
<input id = "Input" type = "ARG(InputImageType)" />
<input id = "Input" access = "getType" type = "ARG(0)"/>
<input id = "Input" access = "getNumDimension" type = "ARG(1)"/>
```

Two accessors are defined for the image input field “Input” to allow the wrappers to access the image dimension and image pixel data type using the given methods *getType*, and *getNumDimension*. The pixel type corresponds to the first argument of the input templated type and the dimension corresponds to the second argument, as stated by the **`type`** attribute. This information is used to determine if an input image provided by the user to the wrapped engine is supported. If so that image information will be used to determine which template configuration for the ITK filter to be used.

Filters also have one or more outputs, which can be defined using the **output** tag. The name of the output corresponds to the suffix of an ITK *Get*... method. As the **input** tag, the **output** tag has a **type** attribute that determines the type of the output field. Most of the time the type of the output corresponds to the type of the input.

5.2 Template configurations

The handling of the template configurations is the most difficult part of the wrapping process.

By default, classes inherit the template configuration from their parent class. Thus, only a subset of classes needs to go through this process. Derived classes can redefine the template configuration, and reuse a partial configuration defined previously by the parent class. The tricky part is that an input field can be defined in the parent class and associated to a template argument of the parent class that will be redefined later on by the derived class!

To start a template configuration, the user shall use the **arguments** tag. The nested **argument** type will define sequentially the template parameter of the class. The name of an argument is specified with the attribute **id**. It shall be followed by a **type** tag, which can either be a terminal type (a built-in) or another templated type! In that case, this type will have sub-arguments. Input field will refer to the name of the argument.

```
// Parent filter
<filter id = "MeshSource">

      // Template Configurations
      <arguments>
            <argument id = "OutputMeshType">
                  <type id = "Mesh">
                        <argument id = "MeshPixelType">
                              <type id = "float"/>
                        </argument>
                        <argument id = "MeshDimension">
                              <type id = "2"/>
                              <type id = "3"/>
                        </argument>
                  </type>
            </argument>
      </arguments>

      // Output Definition
      <output id = "Output" type = "ARG( OutputMeshType )"/>

      // Derived Filter
      <filter id = "ImageToMeshFilter">
            // Template Configurations
            <arguments>
                  <argument id = "InputImageType">
                        <type id = "Image">
                              <use_argument ref = "MeshPixelType"/>
```

```
                        <use_argument ref = "MeshDimension"/>
                    </type>
                </argument>
                <use_argument ref = "OutputMeshType"/>
            <arguments/>

            // Input Definition
            <input id = "Input" type = "ARG( InputImageType )"/>
            <input id = "Input" access = "getType" type = "ARG(0)"/>
            <input id = "Input" access = "getNumDim" type = "ARG(1)"/>
            ...
        </filter>
        ...
</filter>
```

5.2.1 Inheritance and redefinition

The previous XML file defines the template configuration for the parent class *MeshSource* and the derived class *ImageToMeshFilter*. The template definitions of the parent class expose the following configurations.

```
itkMeshSource< itkMesh< float, 2 > >
itkMeshSource< itkMesh< float, 3 > >
```

The derived class *ImageToMeshFilter* reuses the template configurations defined by the parent class for its second argument (corresponding to the output mesh). The **use_argument** tag allows the user to reuse a previously defined argument. The *ImageToMeshFilter* also requires an additional template for the input image. The first argument is defined as an *Image,* templated by two arguments that are reused from sub-arguments defined by the parent class. This basically means that the image pixel type and image dimension shall be synchronized with the mesh pixel type and dimension. This leads to the following configurations:

```
itkImageToMeshFilter< itkImage< float, 2 >, itkMesh< float, 2 > >
itkImageToMeshFilter< itkImage< float, 3 >, itkMesh< float, 3 > >
```

As mentioned earlier, the type synchronization helps reducing the big type combinatorial.

The XML file also defines one output field for the parent class and one input field for the derived class. The interface between the user and the class are the two accessors defined for the input *Input*. Depending on the type and dimension of the image, extracted with the methods *getType* and *getNumDimension*, the template configuration of the filter as well as the configuration of the output are deduced.

5.2.2 Declarations

The declarations **tag** is somehow equivalent to the **arguments** tag. It can be followed by the **type** and **argument** tags, and defines as well potential template configurations. While the **argument** tag defines the template configurations for the class itself, the **declarations** tag can be used to define the type of inputs/outputs synchronized with one of the class template. The following XML sample illustrates how to define an ITK metric based on the previous declarations of two image types.

```
<declarations>
      <argument id = "MetricType">
            <type id = "ImageToImageMetric">
                  <use_argument ref = "InputImageType"/>
                  <use_argument ref = "OutputImageType"/>
            </type>
      </argument>
</declarations>
```