AUTOMATIC GRAPHIC USER INTERFACE GENERATION FOR VTK

Wilfrid Lefer

LIUPPA - Université de Pau
B.P. 1155, 64013 Pau, France
e-mail: wilfrid.lefer@univ-pau.fr

ABSTRACT

VTK (The Visualization Toolkit) has become one of the most popular modular visualization environments. It is an open source software, which has evolved rapidly, new tools being constinuously integrated and a new (minor) release being produced daily. This rapid evolution makes it difficult to develop a graphic user interface (GUI) while maintaining software integrity, that is coherence between interface and code. In this case traditionnal GUI production tools, such as application builders, are not appropriate. This paper proposes a re-engineering approach for automatically generating GUIs for VTK and gives solutions for most of the issues that have to be addressed. We take advantage of the object-oriented feature of VTK to propose a source code analysis method that generates a software components database. Then the rich information contained in this database is used to build a GUI for VTK using a specific GUI technology. This involves a fine analysis of the components of the VTK source and the relationships between them in order to select the components that should be included in the GUI. Then the GUI is generated, which includes a run-time environment to generate and execute the code corresponding to the applications designed by the users. Although VTK has been used to implement our software, the concepts and solutions proposed in this paper are general and could be applied to any object-oriented visualization toolkit.

Keywords: Visualization Modular Environments, Graphic User Interfaces, Re-engineering Approaches, Object-Oriented Code Analysis, VTK.

1. INTRODUCTION

During the early years of scientific visualization, several toolkits appeared on the market place, each one having his characteristics and of course his fans. But not all of them have reached the same popularity because of the difficulty to integrate the most recent techniques as they are presented by researchers of this fast-growing community. In addition to new methods, the toolkits developers have had to face the major issue of making their tools be able to read data, process them and save results under a number of more or less recognized data formats. Each time a new data format can be accepted by the software, new potential users can be reached. And the problem of maintaining and upgrading those large softwares becomes rapidly a drastic issue that can only be solved by using the more sophisticated software modeling approaches raised from the software engineering research community. The object-oriented model has been a major step in this direction and object-oriented software design tools are now widely used by software developers. Modularity, extensibility, maintainability and reusability are some of the important properties of the object-oriented technology.

VTK (The Visualization Toolkit) is one of the most recent visualization softwares that appeared on the market place. It has been developed using an object-oriented design methodology, each component or group of components being designed using three different models: an object model, a dynamic model and a functional model [1]. From the user point of view though, the programming model used to design visualization applications is the usual dataflow paradigm. This approach makes VTK a rapidly evolving software, its developers team being able to produce a new release daily! VTK includes several interface languages, both interpreted ones, such as Tcl and Python, and compiled ones, such as C++ and Java. In addition to a scripting interface, the most popular visualization toolkits, such as AVS or Iris Explorer, provide the user with a graphic user interface, where the user can select modules with the

mouse, drag them into a canvas and draw links between modules to build the dataflow diagram [2]. Thus the user has a graphic representation of the diagram and can interact with that representation, for instance to change a module properties or to combine multiple diagrams as easily as are some mouse clicks. A simple click on a button of the interface updates the whole pipeline and produces the resulting image or animation. Such a friendly interface is an important feature of a visualization software because it provides an easier access to this technology for end users. VTK lacks such a graphic interface and this has motivated the work presented in this paper but we will show that the concepts and methods presented here have not been specifically designed for VTK but rather can be extended to every object-oriented visualization toolkit. The aim of this project was to develop a tool for automatically generating graphic user interfaces (GUI) for object-oriented visualization toolkits. The main features of our software are:

- the automatic or semi-automatic generation of GUIs.
- the ability to work on sources written with various popular object-oriented languages, particularly C++ and Java.
- the possibility to generate GUIs for various GUI toolkits, such as X/Motif, Tcl/Tk, Java, ...
- the possibility for the interface designer to customize the interface by selecting the modules and properties of the modules that should appear in the interface.
- an easy maintenance of the graphic interface: it is possible to generate a new GUI for each new release of the software in a minimal effort.

The remaining of this paper is organized as follow. Section 2 describes the general architecture of our software. Section 3 presents the concepts used for analyzing the source code and generating the language database that is used by our GUI generation tool, which is described in section 4. Section 5 concludes and give directions for future research.

2. SOFTWARE ARCHITECTURE OVERVIEW

The graphic user interface aims at providing the user with ergonomic facilities for designing a visualization application. In a modular environment with a dataflow programming paradigm, these facilities generally include:

- a list of draggable boxes representing the modules. There are generally two types of modules: sources, which produce data, and filters, which take data as input and produce data as output. In addition, special boxes representing complete and autonomous pipelines in which several modules are linked together to achieve a given task, can be found. For instance a pipeline including a data reader, an isosurface extractor, a polygonal data mapper and a rendering window, may be available, as a commonly used visualization application.
- a canvas in which the modules are dragged and linked together to design the application pipeline. Within this area all the modifiable resources of a module can be displayed and changed as necessary. Thus the modules generally have two graphic representations in the canvas: expanded, to show their attributes, and shrunk, to have a compact representation.
- a runtime environment that controls the validity of the modules connections, generates the script corresponding to the designed pipeline and executes the script to produce the results. Indeed not every module can be connected to every other one because the data produced by the first module have to be of the same type as those taken as input by the second module.

In order to design such a graphic user interface, we need to know the following information about the visualization software:

- the name of all the modules the user may need in order to design its application.
- for every module the name of all the properties of the module that can be changed by the user, together with the name of the methods that serve as interface to these properties.
- the type of input data for filters and the type of output data for sources and filters.
- the syntax of a scripting language, which will be used to generate the code and to execute it.

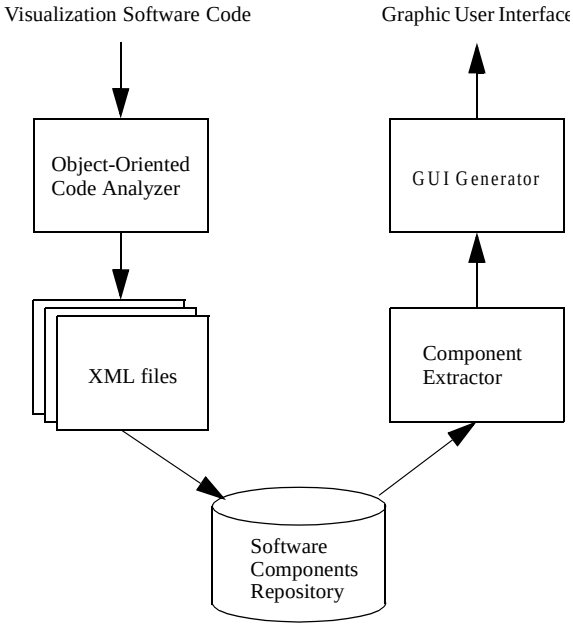

Figure 1: Architecture of our software.

Our software architecture consists of two phases: a first phase where a source code analyzer builds a structural information database related to the visualization software, and a second phase where the interface is built according to the content of the database. Figure 1 shows the different modules of our software. The source code of the object-oriented visualization software is parsed and analyzed by our *object-oriented code analyzer* described in section 3. The result is a set of files in the XML format, which

are archived in an object-oriented database management system, which acts as a software components repository. Once the database has been built, the information is extracted and filtered by a *component extractor*. This program selects the components that have to be included in the interface and retrieves the methods to be called for setting the different properties of the modules. The *GUI generator* builds the graphic interface and produces the script code generation procedures.

3. OBJECT-ORIENTED CODE ANALYSIS

C++ is a language that features high level, abstract concepts, such as function polymorphism, multiple inheritance, or genericity, which makes it necessary to perform a deep analysis of the source code [3]. This accurate code analysis will allow us to solve problems occuring at the GUI generation phase, such as those described in section 4.4.1. But the C++ grammar is known to be one of the more difficult to solve by a code analyzer. Details about the numerous ambiguities that a developper has to cope with when writing an analyzer are not in the scope of this paper and are skipped here. Of course we have been looking for a public domain analyzer but it turned out that we did not find anyone that works well enough for parsing the hundreds of VTK files successfully, i.e. that satisfy our needs. Several commercial C++ compilers include a reliable parser of course but the parser can not be extracted from the distribution.
So we have written a complete C++ analyzer [4], which is based on the FOG grammar (proposal of E. Willink in 1999). It includes the usual lexical, syntactic and semantic phases. This code analysis phase is clearly separated from the GUI production one. This allows us to connect the analyzer to another programme. For instance we have developped a software to automatically generate software components for online applications, which uses the results produced by our C++ code analyzer. In order to facilitate the portability of the results, our analyzer generates XML documents, which contain all information about a C++ application. Such information can be very usefull for various applications. All the XML documents are stored in a database, called the *components database*.

4. GENERATION OF THE GUI

Several independent software components are associated to generate the GUI. We present the general architecture of this part of our software and each component is detailed in the following sections.

4.1 SOFTWARE ARCHITECTURE

Figure 2 shows the software modules that interact together to generate the graphic user interface. The *component extractor* gets the software components produced by our object-oriented code analysis described in section 3. Remember the main tasks that have to be performed before the generation of the interface can take place:

- select the classes to appear as modules in the interface.
- for each class select the properties that can be changed by the user.
- determine at least one method able to modify the value of each selected property.
- build the code generation procedures that will be called to generate the script corresponding to the pipeline.

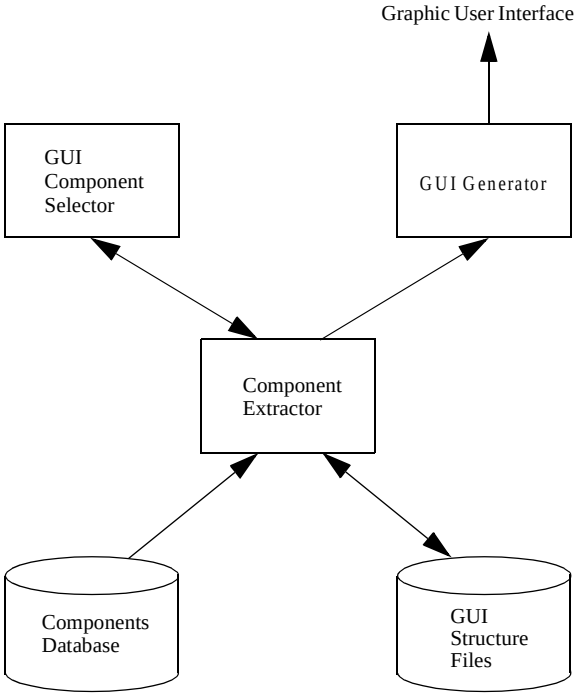

Figure 2: Graphic user interface generation.

Ideally we would like the software be able to analyze the source code through the software components database and to perform all the above mentioned tasks. It is easy to figure out that this is a real challenge. At the current state of development only certain tasks have been automated. These tasks are performed by the *component extractor*. The remaining of the tasks are performed manually by the user through the *GUI component selector*, which is a graphic tool that allows the user to select the different components to be included in the interface by simple mouse clicks. The *GUI component selector* is just a graphical user interface and selected components are toggled by the *component extractor* in response to events occurring in the *GUI component selector* interface. The components selection works as follows: the *component extractor* gets the source code information from the components database, performs all automated selection tasks and then the whole list of components is presented to the user through the *GUI component selector*. Components are toggled or not, depending on whether they have been pre-selected by the *component extractor*. Let us

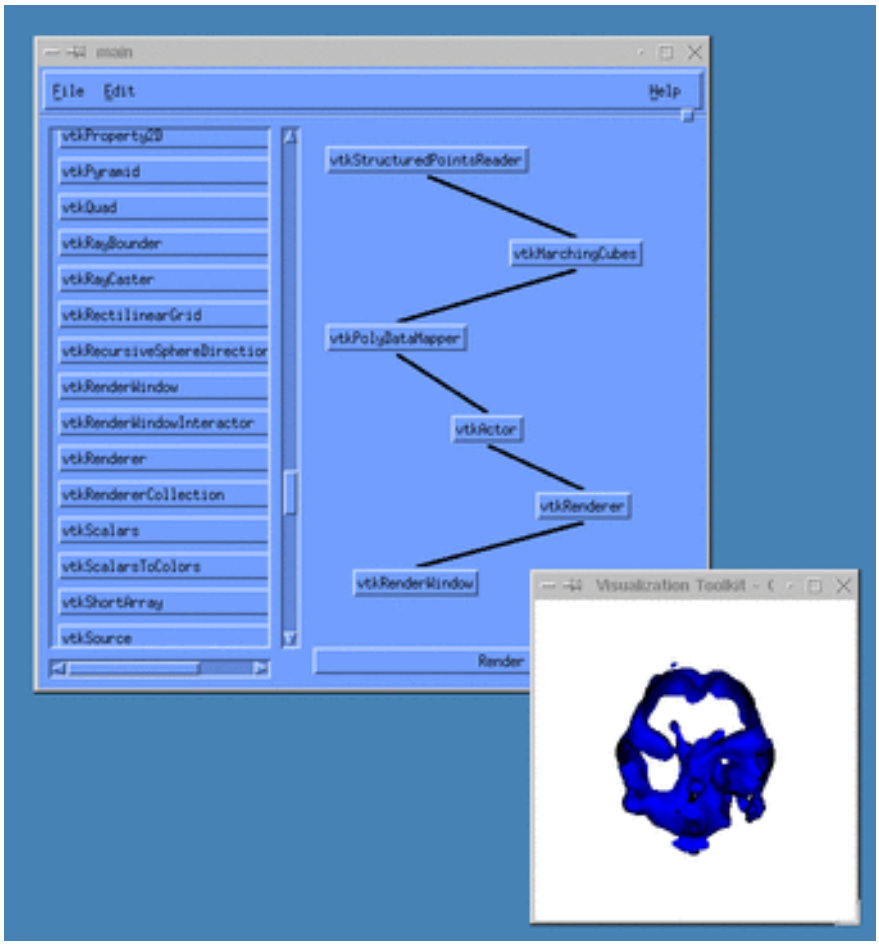

Figure 3: A snapshot of a Motif GUI for VTK, generated automatically from a reengineering approach.

note that a nice property of the separation of this process between both modules is that since more tasks will be automated in the future, no change will appear in the design of this architecture, the only impact being that more components will be pre-selected by the *component extractor*, leading to less work to do for the user. Indeed the pre-selection of components by the *component extractor* does not prevent the user to change the selection afterwards since the user task occurs next to the automated task.

4.2 COMPONENTS SELECTION

The user selects the components to appear in the GUI and the methods to be called for each selected property. Due to the object-oriented paradigm used to model the visualization software, all classes are presented in the GUI component selector, that is both abstract and concrete classes, each property or method being attached to the class in which it has been defined. This allows the user to select a property or a method for a group of selected classes once, by toggling it at the abstract class level instead of having to select it for all inherited classes separately. It means that it is possible to select a property or method for a class that has not been included in the selection of classes that will appear in the software GUI. When all components have been selected, the user validates the selection and click on the «Generate Interface» button to produce the VTK GUI. Because the number of tasks that have to be performed by the user to define a GUI for the whole VTK software can be important, the *component extractor* can save the whole configuration in a so-called GUI file, which contains the list of all software components and their respective flags (selected or not). For each new release of the software, even if there are minor changes, a new GUI has to be generated. Thus the *component extractor* can be asked to load the GUI file of the previous version of the software together with the source code description of the current version that is stored in the components database. Components of the new version of the software that match components of the previous version will be automatically toggled according to their previous status by the *component extractor*. Hence only new components will have to be treated by the user. Since the number of changes from a release to the next is generally small as compared to the size of the software, there is a significant gain in time in using this possibility.

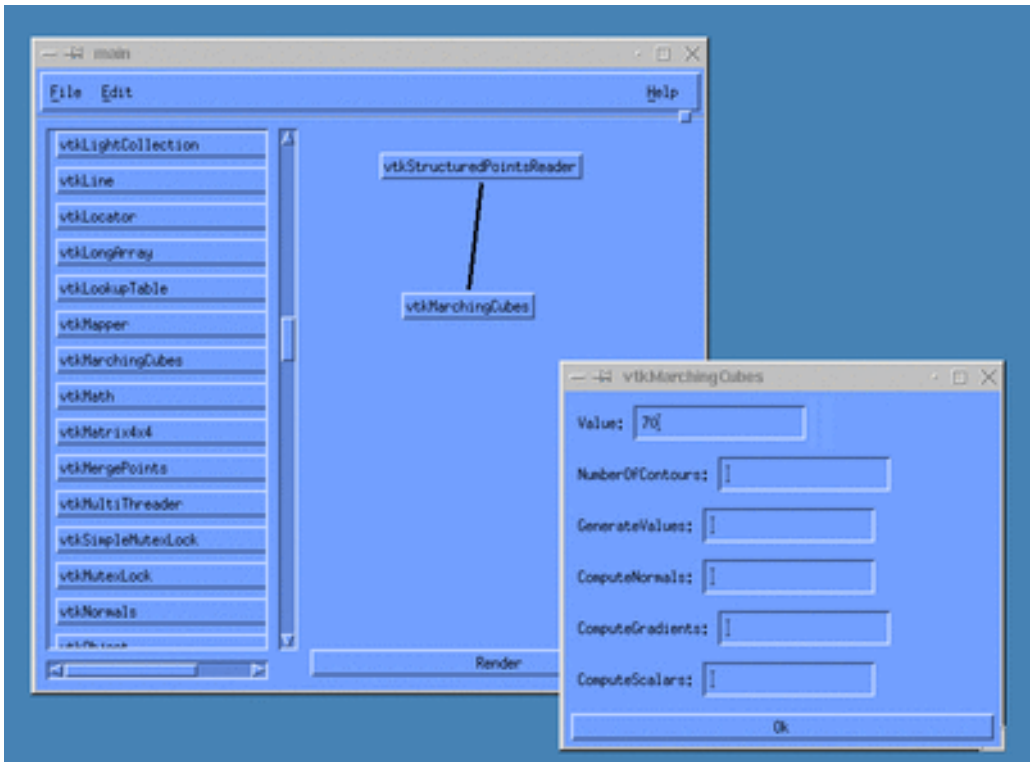

Figure 4: Object properties can be modified thanks to customized forms built on-the-fly.

4.3 GUI GENERATION

The *GUI generator* gets the list of selected components and all the necessary information from the *component extractor* and generates the source code for the GUI using a specific GUI toolkit. The code may need to be compiled if the graphic API uses a compiled language, which is the case for X/Motif or Java for instance, or it may be interpreted without any change if the API uses an interpreted language, which is the case with Tcl/Tk for instance. Each of the tasks of figure 2 have been implemented as C++ classes, except for the *GUI generator*, which has been implemented as a generic class, which realizes all common tasks of the GUI generation, and a specific class, inherited from the generic class. The specific class allows us to implement all tasks related to a specific GUI technology. Thus there is a specific class for generating X/Motif interfaces and a specific class for generating Java interfaces.

4.4 AUTOMATING COMPONENT SELECTION TASKS

The following discussion concerns the automatization of certain tasks performed during the components selection process. It will concentrate on VTK because VTK is the software on top of which our GUI has been built, but similar issues would arise with any other object-oriented visualization software and would probably lead to similar solutions. The aim of this discussion is to show some of the problem we have encountered and the solutions proposed to solve them when they exist. This part of the work is still in progress and we expect more and more automated tasks as we will get insight in the structural information of the software.

4.4.1 IDENTIFYING MODULES NAMES

Depending on the software structure, finding the classes implementing a given module could be difficult, due to the various forms of inheritance implemented by various object-oriented languages. In VTK the names of the modules are also the names of the corresponding classes and are familiar to the users of the toolkit. Many classes are abstract classes though, which should not be part of the GUI because they should not be instantiated themselves but rather one of their subclasses. For instance the *vtkStructuredPointsToPolyDataFilter* class should be unselected but its subclasses *vtkStructuredPointsToGeometryFilter* and *vtkMarchingCubes* should be part of the GUI. Our object-oriented code analyzer allows us to retrieve all inheritance relationships between classes. Hence we are able to withdraw abstract classes from the selection. A particular case concerns abstract classes that can be instantiated though. Actually they are not instantiated themselves but rather one of their subclasses. If the user instantiates such a class directly, a procedure in the VTK source determines which of its subclasses should be instantiated instead, as a function of the system

configuration. An example of this is the class *vtkRenderWindow*, which is the class for windows into which rendering can take place. If an application program instantiates an object of type *vtkRenderWindow*, a portion of code in the VTK source determines which graphic library is used by the system and instantiates the right subclass accordingly, such as *vtkOpenGLRenderWindow* for OpenGL rendering under X11. Today this particular case of inheritance is not treated properly by our software. In order to avoid a wrong selection that the user may not be aware of, we maintain a list of such classes and the component extractor does not process the hierarchies whose root is in the list. Thus *vtkRenderWindow* has been put into the list.

4.4.2 MATCHING PROPERTIES AND METHODS

Automatically selecting the properties to appear in the interface is a difficult task because there is no structural information that can give reasonable hints to make the right choice. But this is not a drastic limitation because of the possibility to save the configuration for a given release of the software and to reload it for the next version, so that properties will be selected according to the history. Once a property has been selected it is necessary to find at least one method able to change the value of this property. It is not rare that the name of the method can not be directly inferred from the name of the property itself. In this case the user is asked to make the right choice through the *GUI component selector*. Fortunately in VTK, in most of the cases, the methods are named according to the name of the property that they get or change value. Indeed many methods to access class properties are automatically built by C macro-functions, so that we have been able to automatically determine the name of the methods to invoke for a given property in most of the cases. Another problem is that there is generally more than one method that can be used to modify a given property. For instance in VTK there are at least three methods that allow to set the *ColorMode* property of objects of class *vtkMapper*: *SetColorMode(int)*, *SetColorModeToMapScalars()* and *SetColorModeToLuminance()*. In order to determine the right method to use and how to use it, we need to know the list of parameters of the methods, the type of each parameter and the type of the value returned by the method. These information is sufficient to produce the script generation procedures for each method invocation. But it may not be enough to determine whether a method is able to modify a property value or not. In this case we need to have information on the code of the method. This is a part of our object-oriented code analysis that is still under development.

4.4.3 CONNECTING MODULES

The generation of the script corresponding to the visualization pipeline designed by the user thanks to the GUI requires to connect the classes together. Each connection represents a path followed by some data, according to the dataflow paradigm. In VTK all source objects and filters have a method called *GetOutput()*, which is used to connect an object to the next object. In the same way all filter objects have a method called *SetInput()*, so that connecting an *object* filter with an object *reader* in C++ is achieved by the following instruction:

filter->SetInput(reader->GetOutput());

Any attempt to connect two modules by the user should be validated by the GUI software. This is possible because we know for each class the type of the data returned by the *GetOutput()* method and the type of the parameter of the *SetInput()* method.

4.4.4 RUN-TIME ENVIRONMENT

By now we have generated a GUI for VTK using the Motif toolkit, probably the most powerful and popular GUI technology in the Unix world. The GUI generation process is handled by several classes: *viscGUIGenerator*, which provides common functionalities for all toolkits, and classes implementing specific functionalities, such as the *viscMotifGUIGenerator* class, which innherits from *viscGUIGenerator*. Both classes have access to the components database through an interface class, named *viscExtractVTKInfo*, which provides routines to browse classes, methods and arguments of the VTK software structure.

The Motif GUI includes a list of all classes that can be instanciated and a canvas in which the user designs the application, as show on figure 3. Instances are dragged from the button list and dropped into the canvas. Once the user has built a valid pipeline he just have to click on the «Render» button to obtain the image. For simplicity reasons, we use the Tcl interface for generating the code implementing a given pipeline. Due to the technique used in VTK to encapsulate the compiled code as Tcl commands, the Tcl implementation is as efficient as the compiled one. The code generated for the pipeline that is shown on figure 4 follows:

```
# object instanciations
vtkStructuredPointsReader obj0
vtkMarchingCubes obj1
vtkPolyDataMapper obj2
vtkActor obj3
vtkRenderer obj4
vtkRenderWindow obj5

# object properties
obj0 SetFileName brain.vtk
obj1 SetValue 0 70
obj4 SetBackground 1 1 1
obj5 SetSize 250 250

# pipeline links
obj1 SetInput [obj0 GetOutput]
obj2 SetInput [obj1 GetOutput]
obj3 SetMapper obj2
obj4 AddActor obj3
obj5 AddRenderer obj4

# now run it
obj5 Render
```

The user has the possibility of saving this code in a file for future reuse. Note that some object properties have been changed in this example. Object properties can be changed by clicking on the modules in the canvas with the right mouse button. Then a form window is generated on-the-fly, which allows the user to set the selected properties, as shown on figure 4. Note that some object methods require several arguments, such as the *SetValue()* method of the *vtkMarchingCube* class. A single text field widget is enough though because we keep the value in text mode and write it to the Tcl file without any format transformation, as a sequence:

```
<object> <method> <value>
```

Later one can imagine adding a property checking procedure to validate the user entries. The forms are generated on-the-fly for obvious reasons: there are several hundreds classes in the VTK toolkit and making all the forms at a time would require a huge amount of memory. Indeed, since we have convenient interface routines provided by the *viscExtractVTKInfo* class to browse classes and their methods, including through inheritance relationships, this procedure was easy to realize.

5. CONCLUSION AND FUTURE WORK

VTK has become one of the most popular and widely used visualization toolkits on the market place. Although the former reason of this success was probably due to its open source status, the variety of interface languages available and its ability to continuously integrate new algorithms have largely contributed to this situation. As compared to other similar products, VTK still lacks a graphic user interface. The size of the software and its rapid evolution make it difficult to maintain a bug-free GUI using traditional approaches, such as application builders. Even with table driven interface generators, the developer has the entire responsability of checking by hand the integrity of the interface according to the code. We have proposed an automatic approach, which allows us to generate a new GUI in a few minutes of computing. We have presented a number of concepts together with a re-engineering approach to analyze the source code and build user interfaces for VTK. Thus there can not be any inconsistency between the interface and the code.

It should be noted that, although the current implementation concerns VTK, the approach is general and all the concepts developed in this paper could be applied to any object-oriented visualization toolkit. Because the different tasks have been separated in independent software components, only the specific *GUI Generator* component has to be rewritten if a new GUI technology is chosen. Thus it should be easy to generate GUIs for various technologies with a minimal development time overhead. It would allow VTK users to choose the look-and-feel of their visualization toolkit and to make it matching the look-and-feel of their familiar environments. The software component repository produced by our object-oriented code analysis is stored in a database with the XML format. XML being now recognized as a standard for data exchange over the networks, one can imagine partitioning our software in two parts: the code analysis part, which will be located on a machine of the VTK development team, and the GUI generation part, which would be available for various operating systems and could be downloaded by users from various repositories. Thus the VTK team would just have to run the code analyzer and make the components database available online. Then the users would have to run the GUI generation software and ask for their favorite GUI technology. The GUI generation software would act as a client of the components database server and download the code components and generate the GUI using the requested technology. We feel that this work could have other application areas. In the context of another project under development by our team, we have started to use it to generate automatically Corba envelopes to build visualization components for the Web. How this work could be used to integrate visualization and other technologies, such as Internet and databases, is also matter of investigation.

REFERENCES

[1] Schroeder, W., K. Martin and W. Lorenzen, The Visualization Toolkit - An Object-Oriented Approach to 3D Graphics - 2nd edition. Prentice Hall, 1998.

[2] Cameron, G. Modular Visualization Environments: Past, Present and Future. Computer Graphics, 29(2), pages 3-4, 1995.

[3] Stroustrup, B. C++. Addison Wesley, 1997.

[4] Gahide, P. Création Automatique de Composants Logiciels: du C++ vers CORBA. DEA Report, University of Lille, France, 2001 (in french).